Using Coq to Write Fast and Correct Haskell

John Wiegley
BAE Systems
USA
john.wiegley@baesystems.com

Benjamin Delaware
Purdue University
USA
bendy@purdue.edu

Abstract

Correctness and performance are often at odds in the field of systems engineering, either because correct programs are too costly to write or impractical to execute, or because well-performing code involves so many tricks of the trade that formal analysis is unable to isolate the main properties of the algorithm.

As a prime example of this tension, Coq is an established proof environment that allows writing correct, dependently-typed code, but it has been criticized for exorbitant development times, forcing the developer to choose between optimal code or tractable proofs. On the other side of the divide, Haskell has proven itself to be a capable, well-typed programming environment, yet easy-to-read, straightforward code must all too often be replaced by highly optimized variants that obscure the author's original intention.

This paper builds on the existing Fiat refinement framework to bridge this divide, demonstrating how to derive a correct-by-construction implementation that meets (or exceeds) the performance characteristics of highly optimized Haskell, starting from a high-level Coq specification. To achieve this goal, we extend Fiat with a stateful notion of refinement of abstract data types and add support for extracting stateful code via a free monad equipped with an algebra of heap-manipulating operations. As a case study, we reimplement a subset of the popular bytestring library, with little to no loss of performance, while retaining a high guarantee of program correctness.

CCS Concepts •**Software and its engineering** → *Functional languages;*

Keywords Performant Certified Software, Stepwise Refinement

ACM Reference format:
John Wiegley and Benjamin Delaware. 2017. Using Coq to Write Fast and Correct Haskell. In *Proceedings of Haskell'17, Oxford, United Kingdom, September 7–8, 2017,* 11 pages.
DOI: 10.1145/3122955.3122962

1 Introduction

Everyone loves the benefits of dependent types, but few people like programming in Coq. On the other hand, many more like programming in Haskell. How can we bridge the divide between the two? Recently there has been increased interest in shrinking this gap by enriching Haskell's type system, either via refinement types (LiquidHaskell [20]), or by bolting dependent types onto Haskell [7], enabling certification inside the language itself. Other work has tackled the problem of formally verifying Haskell programs from a different angle by lifting programs written in system FC into Agda [1].

This paper advocates another approach: the development of certified code in a dependently-typed surface specification language which is then extracted to Haskell. Coq already supports the extraction of Haskell programs from Gallina, but users have to tangle with eccentricities like termination checking. Our observation is that rather than writing executable, dependently-typed *programs* in Coq, why not leverage its power as a proving language to embed a more pleasant declarative language, one which defers programming idioms such as general recursion and foreign function calls to an eventual Haskell implementation? We argue that the specification features and data refinement mechanisms of the existing Fiat refinement framework [5] provide a natural environment for specifying such high-level programs *and* deriving efficient Haskell implementations from them.

The existing Fiat framework was already expressive enough to capture specifications from a wide variety of domains, allowing clients to specify library behaviors in terms of high-level algebraic data types, as shown by the straightforward specification of the popular `ByteString` library in Figure 1. This paper extends the framework to support the derivation of even more efficient correct-by-construction heap-manipulating implementations using external function calls. These extensions allow users to, for example, correctly derive the implementation of the `pack` method from the `ByteString` library shown in Figure 2. In order to produce low-level heap-manipulating implementations, we extend Fiat's notion of refinement to incorporate an explicit view of the heap at later stages of refinement. We introduce a lightweight mechanism for capturing foreign function calls via a translation from shallowly-embedded Gallina functions in the nondeterminism monad to programs in a variant of the free monad, where the foreign function calls are the algebraic operations. We demonstrate our ability to generate reasonable code by deriving an implementation of the `ByteString` library specified in Figure 1 which we extract to Haskell for benchmarking.

2 A Motivating Example

To examine the problem in more detail, consider the specification of Haskell's bytestring library as an abstract data type (ADT). The methods of this type allow clients to: create empty `ByteString`s, or build them from lists of bytes; add bytes to, and remove them from, the front of a `ByteString`; concatenate two `ByteString`s together; and other operations.

Figure 1 presents a naïve, functional implementation of this interface using an algebraic datatype for lists as the internal representation. While appealing from a specification standpoint, this implementation is much too inefficient: in Haskell, for example,

This work was sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007.

Haskell'17, Oxford, United Kingdom

DOI: 10.1145/3122955.3122962

```
Definition ByteStringSpec := Def ADT {
  rep := list Word,
  Def Constructor empty : rep := ret [],
  Def Constructor pack (xs : list Word) : rep := ret xs,
  Def Method unpack (r : rep) : rep * (list Word) :=
    ret (r, r),
  Def Method cons (r : rep) (w : Word) : rep :=
    ret (cons w r),
  Def Method uncons (r : rep) : rep * (option Word) :=
    ret (match r with
         | nil => (r, None)
         | cons x xs => (xs, Some x)
         end),
  Def Method append (r1 : rep) (r2 : rep) : rep :=
    ret (r1 ++ r2)
}.
```

Figure 1. Naïve specification of a ByteString

```
haskell_bs_pack :: [Word8] -> ByteString
haskell_bs_pack xs = unsafeDupablePerformIO $ do
    let len = length xs in
    p <- mallocPlainForeignPtrBytes len
    withForeignPtr p $ \ptr -> pokeArray ptr xs
    return $ PS p 0 len
```

Figure 2. Optimized implementation of "pack"

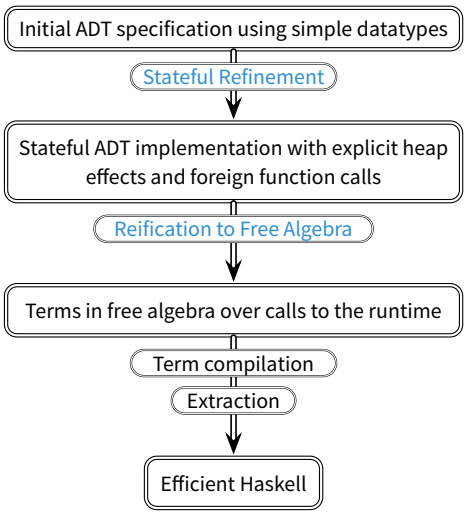


Figure 3. A pictorial representation of the derivation process, with transformations new to this work highlighted in blue

assuming a 64-bit system, a list of bytes requires that each byte be referenced from a cons cell, using 40 allocated heap bytes per byte stored; whereas the optimized ByteString implemented in the Haskell standard library allocates only one byte per byte stored, plus 80 bytes of overhead per ByteString for maintenance. This efficiency is achieved by explicitly managing heap memory via foreign function calls to library functions written in C. Figure 2 shows a representative implementation of pack from this library (about which more is said at the end of Section 4).

Verifying that such a complex implementation meets the high-level specification written in terms of lists would be a natural application of dependent types, were they available in Haskell. However, even with such a facility, the nature of the optimized implementation does not lend itself readily to proof: it relies on several details concerning the semantics of the runtime environment, such as IO and explicitly managed heaps, that are orthogonal to the semantics of ByteStrings themselves. Any proofs written in Haskell using dependent types would need to handle these details, conflated with the underlying proof of functional correctness.

One of the pleasant lies we tell ourselves about data abstraction is that it frees clients from worrying about implementation details of the library code: any type-safe program written against a particular interface will behave correctly at run-time. Unfortunately this ignores the obvious impact that an implementation's choice of data structures and algorithms can have. Consider the dizzying array of options presented on one GHC developer's webblog [22], where no fewer than eleven string representations are given, each varying in the runtime characteristics offered or the underlying list element type.

While there exist systems to help select optimal implementations from a library of existing, manually written implementations [18], the ideal scenario is for a client to *specify* the desired functionality of a library and to have an optimized implementation synthesized for them automatically. There have been a few realizations of this idea for libraries with specifications from a targeted domain, particularly query-like operations [8, 16, 17]. The ultimate goal of the work we present here is to generalize this approach, enabling high-level algorithmic and data structure optimizations while also allowing precise specification of the in-memory layout of data structures and tailoring the algorithms to these choices, all in a way that is opaque to library clients.

This is achieved by ensuring that proofs involving ByteString semantics occur separately, in the domain of lists, such that the validity of these proofs is preserved for any implementation in the domain of managed heaps and IO operations that fulfill certain criteria. To accomplish this, we suggest an alternative approach, using the venerable ideas of refinement advocated by Djikstra [6] and Hoare [10], and more recently realized in systems like Fiat [5] and the Isabelle Collections Framework [13]. Starting from the specification in Figure 1, we refine this into an ADT with an explicit view of the runtime environment. Next, we ensure that unimplemented bits of the ADT can be implemented via calls to a set of functions provided by a fixed interface, in this case those provided by a heap-manipulating library. We observe that such operations are basically functions in the free monad, whose operations are those of the interface, and lift the functions to these operations. From here, we use Coq's extraction mechanism to produce reasonably good Haskell code. This process is summarized in Figure 3.

The rest of the paper proceeds as follows: we present a core language for declarative ADT specifications and their clients before formalizing our notion of refinement for heap-manipulating programs. We next describe our implementation of this calculus in Fiat and our experience extracting performant code from these partial implementations. We then present an empirical evaluation of the extracted code and discuss related work before tackling future directions.

3 A Core Calculus for Data Refinement

We begin by introducing a core calculus which includes the key features needed for data refinement by our source language. Figure 4 presents the syntax of this calculus, which is a variant of PCF extended with an arbitrary set of algebraic data types T, abstract data

$\tau ::= \tau \to \tau \mid X \mid T$ (* Algebraic Data Types *)

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

$I ::= ADT\ \{\ \textbf{rep} := \tau\ ;$
$op_1(r_1 ... r_n : \textbf{rep})\ (x_1 : \tau_1) ... (x_n : \tau_n) : \textbf{rep} \times \tau := e;$
$...$
$op_n(r_1 ... r_n : \textbf{rep})\ (x_1 : \tau_1) ... (x_n : \tau_n) : \textbf{rep} \times \tau := e\ \}$

$p ::= \textbf{let}\ X_1 := I_1\ \textbf{in}\ ...$
$\textbf{let}\ X_n := I_n\ \textbf{in}\ e$

Figure 4. Core syntax of Fiat

types (ADTs), and, most importantly, a nondeterministic choice operator, $\{\ x : \tau \mid P\ (x, e_1, ..., e_n)\}$. Intuitively, this operator represents a "hole" in an expression which can evaluate to any value of type τ that satisfies the predicate $P\ (x, e_1, ..., e_n)$. The language of these predicates is a parameter of the calculus, and there is no requirement that they be decideable, in contrast to liquid types [20][1]. As an example, consider an expression representing a sorted version of a list l:

$$\{l' : \text{List } \mathbb{N} \mid \forall m\ n.\ m \le n < |l'| \to l'[m] \le l'[n] \wedge \text{Permutation}(l, l')\} \quad (1)$$

While this expression precisely spells out *what* is to be computed, in that this predicate holds only for a uniquely sorted list, the operational semantics of the choice operator, given in ChoiceR, do not specify *how* this list is to be computed. Programs in this calculus consist of an initial sequence of ADT definitions followed by a client program that can utilize the operations of those ADTs. The semantics of these operations is defined with respect to a distinguished representation type, **rep**. Well-typed clients of an ADT are oblivious to its choice of representation type. The full operational semantics and type system for this calculus are given in Figure 5 and Figure 6, respectively.

This calculus captures the mixed-language model we pursue here, with the initial sequence of ADT definitions providing libraries whose semantics are given in Coq, and the client expression representing a Haskell program using those libraries. Our goal is to start with the operational description of an ADT in Coq, modeling its internal state via the simplest algebraic data type possible, as in Figure 1, and to transform these naïve library specifications into efficient implementations which satisfy the initial specification and can be extracted to performant Haskell implementations to be linked with Haskell clients.

This transformation is carried out within Coq via stepwise refinement [6]. We say that an expression e_2 *refines* another expression e_1 when the possible evaluations of the former are a subset of the latter:

$$e_1 \supseteq e_2 \triangleq \forall v.\ e_2 \longrightarrow v \ \to\ e_1 \longrightarrow v$$

Thus, any implementation of a sorting algorithm is a refinement of Equation 1:

$$\{l' : \text{List } \mathbb{N} \mid \forall m\ n.\ m \le n < |l'| \to l'[m] \le l'[n] \wedge \text{Permutation}(l,\ l')\} \ \supseteq\ \text{quicksort}(l)$$

[1] As a Coq library, the Fiat framework used in our case study uses the Calculus of Inductive Constructions for these predicates.

By applying a series of refinement rules, users can derive a correct-by-construction implementation of a declarative program. An implementation in this context is a fully refined program, in the sense that it either evaluates to a single value or fails to terminate.

Definition 3.1. A program e is *fully refined* when it can evaluate to a unique value:

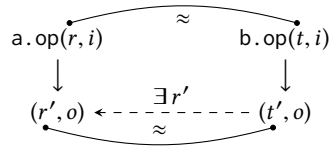
$$\forall v\ v'.\ e \longrightarrow v \wedge e \longrightarrow v' \to v = v'$$

As ChoiceR is the only source of nondeterminism in the evaluation rules of Figure 5, a program that does not have any occurrences of the choice operator is fully refined. Recalling our original goal of deriving ADTs implementations for Haskell clients, we also introduce a more nuanced notion of an ADT implementation:

Definition 3.2 (Fully Refined)**.** An ADT I_o is *fully refined* when all of its operations evaluate to a unique value, assuming that any ADTs mentioned in those operations are fully refined.

This definition captures those ADT implementations that can be extracted to code which can both call and be called from Haskell code, using an interface for Haskell operations which are themselves expressed as an ADT. Subsection 3.1 discusses this idea in more detail.

In addition to implementing any nondeterministic choices, we also want to support more efficient implementations of the representation type of ADTs. As noted in Section 2, while lists offer a clean specification of the behavior of ByteString, they are neither fast nor memory efficient enough for most clients (indeed, this is an important motivation for the existence of the bytestring library). To this end, we lift our previous definition of refinement on expressions to ADTs modulo an *abstraction relation* [9] on representation types. Intuitively, the implementation of ADT I_n refines another implementation I_o under some abstraction relation $\approx$ if every operation of I_n produces a subset of the concrete values produced by the same operation in I_o and results in related values of their respective representation types, when run in similar representation states:



A key property of this notion of data refinement is that the client program is protected from any changes to the representation type of a "library" ADT by the data abstraction boundary. Thus, a client program should be completely oblivious to the data types and abstraction relation used to produce a derived implementation of the library. In contrast to other refinement frameworks [4, 13] this means there is no need to modify a client program when using a derived implementation.

Definition 3.3 (Soundness of Data Refinement)**.** We say that an ADT refinement is *sound* when replacing an ADT I_o in a well-formed client program with a refined implementation $I_o \supseteq_\approx I'_o$ is a valid refinement:

$$\begin{array}{lcl} \textbf{let}\ X_1 := I_1\ \textbf{in}\ ... & & \textbf{let}\ X_1 := I_1\ \textbf{in}\ ... \\ \textbf{let}\ X\_i := I\_o\ \textbf{in}\ ... & \supseteq & \textbf{let}\ X\_i := I\_o'\ \textbf{in}\ ... \\ \textbf{let}\ X_n := I_n\ \textbf{in}\ e & & \textbf{let}\ X_n := I_n\ \textbf{in}\ e \end{array}$$

Thus, the goal of stepwise refinement of a program e in this calculus is to find a valid sequence of ADT refinements $I_0 \subseteq_\approx I_1 \subseteq_\approx ... \subseteq_\approx I_n$ that produce a fully refined target program.

$$\frac{\Gamma \vdash e_i \longrightarrow e'_i}{\Gamma \vdash C\ (v_1, ..., e_i, ..., x_n) \longrightarrow C\ (v_1, ..., e'_i, ..., x_n)}\ (\textsc{ConstrR})$$

$$\frac{\Gamma \vdash e_1 \longrightarrow e'_1}{\Gamma \vdash e_1 e_2 \longrightarrow e'_1 e_2}\ (\textsc{CAppLeftR}) \qquad \frac{\Gamma \vdash e_2 \longrightarrow e'_2}{\Gamma \vdash v_1 e_2 \longrightarrow v_1 e'_2}\ (\textsc{CAppRightR})$$

$$\frac{}{\Gamma \vdash (\textbf{fix}\ f\ (x : \tau_1) : \tau_2 := e)\ v \longrightarrow e\ [x \mapsto v, f \mapsto \textbf{fix}\ f\ (x : \tau_1) : \tau_2 := e]}\ (\textsc{CFixR})$$

$$\frac{\Gamma \vdash e \longrightarrow e'}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ \ ...\ \textbf{end} \longrightarrow \textbf{match}\ e'\ \textbf{with}\ \ ...\ \textbf{end}}\ (\textsc{CMatchR})$$

$$\frac{\Gamma \vdash e_i \longrightarrow e'_i}{\Gamma \vdash X.op\ (v_1, ..., e_i, ..., x_n) \longrightarrow X.op\ (v_1, ..., e'_i, ..., x_n)}\ (\textsc{CCallR})$$

$$\frac{\Gamma \vdash e_i \longrightarrow e'_i}{\Gamma \vdash \{x : \tau\ \mid P\ (x, v_1, ..., e_i, ..., x_n)\} \longrightarrow \{x : \tau\ \mid P\ (x, v_1, ..., e'_i, ..., x_n)\}}\ (\textsc{RChoiceR})$$

$$\frac{[\overline{x \mapsto v}] \vdash e \longrightarrow e'}{\vdash \textbf{let}\ X_1 := I_1\ \textbf{in}\ ...\ \textbf{let}\ X_n := I_n\ \textbf{in}\ e \longrightarrow \textbf{let}\ X_1 := I_1\ \textbf{in}\ ...\ \textbf{let}\ X_n := I_n\ \textbf{in}\ e'}\ (\textsc{ProgR})$$

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

$$\frac{\Gamma(X, op) = e_i}{\Gamma \vdash X.op_i(v_1, ..., v_n) \longrightarrow \Gamma \vdash e_i[\overline{x \mapsto v}]}\ (\textsc{CallR}) \qquad \frac{\Gamma \vdash P\ (v, v_1, ..., v_n)}{\Gamma \vdash \{x : \tau\ \mid P\ (x, v_1, ..., v_n)\} \longrightarrow v}\ (\textsc{ChoiceR})$$

Figure 5. Core Operational Semantics of Fiat

$$\frac{\Delta;\ \Gamma(x) = T}{\Delta;\ \Gamma \vdash x : T}\ (\textsc{VarT})$$

$$\frac{\overline{\Delta;\ \Gamma \vdash e_i : \tau_i} \qquad \vdash\ C : \overline{\tau_i} \to T}{\Delta;\ \Gamma \vdash C\ (e_1, ..., e_n) : T}\ (\textsc{ConstrT})$$

$$\frac{\Delta;\ \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Delta;\ \Gamma \vdash e_2 : \tau_2}{\Delta;\ \Gamma \vdash e_1 e_2 : T_1}\ (\textsc{AppT})$$

$$\frac{\Delta;\ \Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Delta;\ \Gamma \vdash \textbf{fix}\ f\ (x : \tau_1) : \tau_2 := e : \tau_1 \to \tau_2}\ (\textsc{FixT})$$

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

$$\frac{\Delta\ (X,op) = \tau_1 \to ... \to \tau_n \to \tau \qquad \overline{\Delta;\ \Gamma \vdash e_i : \tau_i}}{\Delta;\ \Gamma \vdash X.op\ (e_1, ..., e_n) : \tau}\ (\textsc{CallT})$$

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

$$\frac{I = \text{ADT}\{\textbf{rep} := \tau\ ;\ \overline{op\ (r_1 ... r_n : \textbf{rep})(x_1 : \tau_1) ... (x_n : \tau_n) : \textbf{rep} \times \tau_i := e_i}\} \qquad \overline{\Delta; [\overline{r \mapsto X}, \overline{x \mapsto \tau}] \vdash e_i : \textbf{rep} \times \tau_i} \qquad \Delta, [\overline{(X, op) \mapsto \overline{X} \to \overline{\tau} \to \textbf{rep} \times \tau_i}] \vdash e : \tau}{\Delta \vdash \textbf{let}\ X := I\ \textbf{in}\ e\ :\ \tau}\ (\textsc{ProgT})$$

Figure 6. Typing Rules of Fiat

3.1 Stateful Refinements

The refinement methodology presented so far features a significant obstacle to generating efficient code: all of the datatypes available to refinement are under the control of the garbage collector, preventing the derivation of explicitly managed heap-allocated data structures. Supporting such a refinement is necessary to obtain a performant implementation of the bytestring library. As previously discussed, an ADT implementation is said to be fully refined modulo a set of ADTs which it is a client of. Thus, a natural solution to this problem is to specify the heap via an ADT whose representation type is a model of the heap and which features methods that enable clients to manipulate this model. Figure 7 shows the interface of such an ADT. The semantics of these heaps is given using a representation type of two sets of mappings, one from addresses to allocation sizes, and the other from addresses to byte values. No relationship is implied between the two maps, meaning that proper use is considered a property of the client, not the heap *per se*. Such constraints may be introduced later by refinement, or proven as a theorem against clients of the heap. For instance, while it is expected that a proper client will never access unallocated bytes, Poke is only specified to mean "changes value at memory position", and not "verifies address is within allocated region", since the latter would impose either a proof burden at every use, or a performance-impairing runtime check. Alloc features nondeterminism, using the choice operator to select *some* block of free memory, without specifying *which*. Given such a specification, a client ADT can be refined using our existing notion of data refinement to explicitly include the heap within its representation type, and its operations can manipulate this reference through the public interface of the heap ADT. As an example, using the following abstraction relation[2],

$$r_o : \text{list Byte} \approx r_n : \langle \text{addr} : \mathbb{N}; \text{len} : \mathbb{N} \rangle \times \textbf{rep}\ \text{heap} \triangleq r_o = \text{Unpack}\ r_n.\pi_2\ r_n.\text{addr}$$

it is straightforward to prove the following refinement lemma about the cons method from Figure 1:

$$\forall r_o\ r_n\ w.\ r_o \approx r_n \to w :: r_o \approx (\langle \text{addr} := r_n.\text{addr}; \text{len} := r_n.\text{len} + 1 \rangle, \text{poke}\ r_n.\text{addr}\ r_n.\text{len}\ w)$$

While this approach suffices for unary methods, it is insufficient for multi-arity methods. When refining append, for example, there is no explicit guarantee that the method will be called with the same model of the heap. We cannot simply pick one heap to modify, as

[2] Unpack h addr len uses Peek to read len bytes from h starting at address a

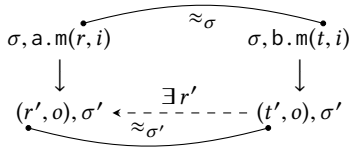
```
ADT Heap := {
rep := (ℕ → ℕ_opt ) × (ℕ → Byte_opt ),

Empty : rep := (λ _. None, λ_. None);

Alloc (r : rep) (len : ℕ) : rep × ℕ:=
  let addr := { addr : ℕ | ∀ addr' sz. r_π2 addr' = sz →
                    ¬ (addr < addr' + sz) ∧ ¬ (addr' < addr + len)} in
   ((add addr len r_π1, r_π2), addr);

Free (r : rep) (addr : ℕ) : rep := (remove addr r_π1, r_π2);

Peek (r : rep) (addr : ℕ) (offset : Size) : rep × Byte :=
  (r, {p : Byte | r_π2 (addr + offset) = p ∨ (r_π2 (addr + offset) = None ∧ p = Zero)});

Poke (r : rep) (addr : ℕ) (offset : Size) (w : Byte) : rep :=
  (r_π1, add (addr + off) w r_π2) }.
```

Figure 7. Specification of a heap as an ADT.

this would have the effect of "forgetting" the bytestrings stored in the other heap! To resolve this problem, we will first refine our core calculus into a stateful variant, presented in Figure 8, that treats the heap appropriately. We next introduce a stronger notion of data refinement that can explicitly relate the representation type of an ADT to the heap.

To begin, we augment the syntax with a new, distinguished type of ADTs S whose representation type is implicitly threaded through a program; any stateless program can be directly lifted to a stateful program. The existing semantics from Figure 5 are updated to explicitly thread state through the rules, and a new rule is added which explicitly passes the state value in calls to the distinguished ADTs. Finally, the typing rules are extended with a rule for these specialized ADTs which ensures that operations have at most one **rep** argument.

We can now augment our definition of ADT refinement to include these state parameters:



Intuitively, abstraction relations are lifted to a ternary relation, relating the representation type arguments of the ADT to a stateful ADT implementation that is threaded through the execution. Armed with an updated abstraction relation:

$$r_o : \text{list Byte} \approx_\sigma r_n : \langle \text{addr} : \mathbb{N}; \text{len} : \mathbb{N} \rangle \triangleq r_o = \text{Unpack } \sigma \ \ r_n.\text{addr}$$

we can now prove the following refinement fact for append[3]:

```
∀r_o r_n r'_o r'_n σ σ' σ" a'.
    r_o ≈_σ r_n →
    r'_o ≈_σ r'_n →
    (σ' , a') = Alloc (Free(Free σ r_n.addr) r'_n.addr) (r_n.len + r'_n.len) →
    σ" = Pack σ' ((Unpack σ r_n.addr r_n.len)+ (Unpack σ r'_n.addr r'_n.len)) →
    r_o+ r'_o ≈_σ" ⟨addr := a'; len := r_n.len + r_n.len'⟩
```

Any program in the base calculus can be lifted to one in this extended stateful calculus in a straightforward manner. The previously defined notions of fully refined and valid ADT refinement can be similarly lifted. Our final approach to stepwise refinement

[3] Pack h l places a list of bytes l into heap h using poke.

$$\tau ::= \ldots \mid \text{SX}$$

$$e ::= \ldots \mid \text{SX.op } (e_1, \ldots, e_n)$$

$$p ::= \textbf{let SX} := I \textbf{ in} \quad \textbf{let } X_1 := I_1 \textbf{ in} \ldots \quad \textbf{let } X_n := I_n \textbf{ in } e$$

$$\frac{\Gamma \vdash \ \sigma, \ e_i \longrightarrow e'_i, \ \sigma'}{\Gamma \vdash \ \sigma, \ \text{SX.op } (v_1, \ldots, e_i, \ldots, x_n) \longrightarrow \text{SX.op } (v_1, \ldots, e'_i, \ldots, x_n), \ \sigma'} \ (\textsc{SCCallR})$$

$$\frac{\Gamma(\text{SX, op}) = e_i \qquad \Gamma \vdash e_i[\overline{r \mapsto \sigma}, \overline{x \mapsto v}] \longrightarrow^* (\sigma', v)}{\Gamma \vdash \ \sigma, \ \text{SX.op}_i(v_1, \ldots, v_n) \longrightarrow v, \ \sigma'} \ (\textsc{SCallR})$$

Figure 8. Updated syntax and semantics of stateful Fiat

of a program e is thus to find a valid sequence of ADT refinements in the pure calculus: $I_0 \subseteq_\approx I_1 \subseteq_\approx \ldots \subseteq_\approx I_i$, followed by a valid sequence of stateful ADT refinements which produce a fully refined target program: $\text{lift}_\sigma(I_i) \subseteq_{\approx\sigma} \subseteq_{\approx\sigma} \ldots \subseteq_{\approx\sigma} I_i$.

The state argument used in the abstraction relation introduces a new wrinkle to ADT refinement, since the validity of a refined program now depends on the client's usage of the state value. Whereas before we could rely on the data abstraction boundary to ensure that a refined client would never call an ADT operation with a representation argument that was not related to the original state under the abstraction relation, a client could now call a heap operation between method calls that invalidates this relationship, by deallocating a pointer between calls to cons, for example. Our current implementation of this refinement calculus relies on clients being well-behaved; we leave statically ensuring this property for future work.

4 Implementation

We now turn to how we utilize the above ideas to derive an Haskell implementation of the bytestring library. This derivation is carried out in Coq on top of the Fiat [5] framework, extended with stateful refinements. Fiat contains a shallow embedding of the calculus of Figure 4 using the nondeterminism monad encoded as mathematical sets and an implementation of ADTs as sigma types. The use of mathematical sets frees the framework from many implementation concerns that programmers typically find constraining in dependently-typed programming. As an example, fixpoints are encoded as the intersection of all the sets closed under the body of the fixpoint, freeing specifications from termination considerations.

Our initial specification of the semantics of ByteString is given in its entirety in Figure 1, stating formally that bytestrings are, as the name implies, a string (or list) of bytes, with all its operations specified directly in terms of such lists. This specification is already computable, and could be refined to an implementation automatically by Fiat as is. The performance of this implementation would be abysmal, however.

Looking to the Haskell bytestring library for inspiration, we find that after several generations they have settled on a simulation of lists of bytes using memory buffers allocated directly on the heap. This allows for highly optimal concatenation behavior when sufficient space remains in the buffer, for example, and frees the

implementation to choose a size by which the buffer is grown whenever more space is needed. The downside to this freedom is that the underlying list semantics are somewhat obscured when reading the code, requiring extensive testing to ensure that all behaviors are as expected.

In order to join the simplicity of the formally specified semantics in Coq with the optimized representation of byte lists using heap buffers, we use Fiat's refinement calculus to transform the simple specification into a direct equivalent of the optimized version found in Haskell. Each refinement step carries with it a proof of correctness, as we move through the various stages of abstraction leading to the final result.

As noted previously in Section 2, a major benefit of the stepwise refinement method is that theorems established against the initial specification may be mechanically transported to cover the final implementation, meaning we need only prove interesting properties of ByteString against its simplest form, rather than confound those proofs with the complexities of buffer management. The overall flow of refinement is presented in Figure 14, and described throughout the rest of this section.

The first step of this proof refines the initial, list-based specification of ByteString is refined from lists to use heap buffers directly, relying on the Fiat specification of heaps shown in Figure 9. This step relies on the notion of stateful refinement introduced in Subsection 3.1; to implement such refinements, we have extended Fiat with a notion of stateful refinement. The ternary abstraction relation used relates the list representation type used in the specification to a pointer representation type that references the shared heap. To implement the stateful semantics from Figure 8, we have defined a lifting function that augments the operations of an ADT with an additional parameter representing the initial heap and an extra return value standing for the final state of the heap. Such ADTs are "stateful", in the sense that each method now lives in the state monad.

As an example, the signature of the "stateful" variant of the cons method in Figure 1 is $\text{cons} :: \text{pointer} \to \text{heap} \to \text{Word} \to \text{pointer} \times \text{heap}$. The abstraction relation used to connect the two operations states uses an auxiliary function, getbytes, to relate a sequence of bytes in the shared heap σ to the list used in the original specification:

$$\text{l} \approx_\sigma \text{p} \triangleq \qquad\qquad \text{l} = \text{getbytes}(\sigma,\ \text{p})$$

This is done by tracking allocated sections of the heap, plus an offset and length within that section, using the type defined in Figure 10. The complexity induced by this transformation is verified against proof requirements generated by Fiat, ensuring that the resulting implementation exactly satisfies the original semantics. As an example of the level of complexity involved, the cons operation in Figure 11, a mere single constructor call in the list-based version, becomes forty-five lines of code in the buffer-based version, some of which is shown in Figure 12, involving three decision points that one of the authors failed to write correctly on two separate attempts. Lastly, the final extracted function for this code is shown in Figure 13, using similar tricks to what the hand-coded version relies on. The remainder of this section covers the details of how these refinements are used to implement the bytestring library.

The first ADT refinement, named ByteStringHeap, refines ByteString in terms of abstract heaps, but it is not yet computable since certain details have yet to be decided: namely, how free addresses on the heap should be allocated. We prove this is achievable by refining

```
Definition HeapSpec := Def ADT {
  (* Two FMaps, one for allocations the other for values
     on the heap. *)
  rep := M.t Size * M.t Word,

  Def Constructor0 empty : rep := ret newHeapState,

  Def Method1 alloc (r : rep) (len : Size | 0 < len) :
      rep * Ptr Word :=
    addr <- find_free_block (` len) (fst r);
    ret ((M.add addr (` len) (fst r), snd r), addr),

  Def Method1 free (r : rep) (addr : Ptr Word) : rep :=
    ret (M.remove addr (fst r), snd r),

  Def Method2 peek (r : rep) (addr : Ptr Word) (off : Size) :
      rep * Word :=
    let addr' := plusPtr addr off in
    p <- { p : Word
         | M.MapsTo addr' p (snd r)
             \/ (~ M.In addr' (snd r) /\ p = Zero) };
    ret (r, p),

  Def Method3 poke (r : rep) (addr : Ptr Word)
                   (off : Size) (w : Word) : rep :=
    ret (fst r, M.add (plusPtr addr off) w (snd r)),

  (* And other methods... *) }.
```

Figure 9. Basic structure of the Fiat Heap ADT.

```
Record PS := makePS {
  psBuffer : Ptr Word;   (* address of allocation *)
  psBufLen : Size;       (* total space allocated *)
  psOffset : Size;       (* offset of byte data *)
  psLength : Size        (* length of byte data *)
}.
```

Figure 10. Internal representation of ByteStrings

```
(* All we do is call List.cons, there is no other behavior. *)
Def Method1 cons (r : rep) (w : Word) : rep :=
  ret (cons w r),
```

Figure 11. Simple specification of cons

```
(* Relies on four helper functions, not shown here. *)
Program Definition buffer_cons (r : bsrep) (d : Word) :
  Comp (Rep HeapSpec * PS) :=
  let h  := fst r in
  let ps := snd r in
  `(h, ps) <-
    If 0 <? psOffset ps
    Then ret (h, simply_widen_region ps 1)
    Else
      If psLength ps + 1 <=? psBufLen ps
      Then make_room_by_shifting_up h ps 1
      Else
        If 0 <? psBufLen ps
        Then make_room_by_growing_buffer h ps 1
        Else allocate_buffer h 1;
  poke_at_offset h ps d.
```

Figure 12. Implementation of cons using heaps

Heap to HeapCanon, establishing the notion of a moving free pointer, although this refinement is unused in the final result[4]. However, we adopt the same techniques to build ByteStringCanon, resulting in a functional implementation of ByteString. Although this refinement

[4]Presently there is no support in Fiat for composing separate lines of refinement—that is, if an ADT A is refined to make use of ADT B, and B is refined into C, then automatically refine A in terms of C—although this is currently under development.

```
extracted_bs_cons :: PS0 -> Word8 -> PS0
extracted_bs_cons p w = unsafeDupablePerformIO $
    if 0 < psLength0 p
    then do
        cod <- mallocPlainForeignPtrBytes (psLength0 p + 1)
        withForeignPtr (psBuffer0 p) $ \ptr1 ->
            withForeignPtr cod $ \ptr2 ->
                copyBytes (plusPtr ptr2 1)
                          (plusPtr ptr1 (psOffset0 p))
                          (psLength0 p)
        withForeignPtr cod $ \ptr -> pokeByteOff ptr 0 w
        return $ MakePS0 cod (psLength0 p + 1)
                         0   (psLength0 p + 1)
    else do
        cod <- mallocPlainForeignPtrBytes 1
        withForeignPtr cod (\ptr -> pokeByteOff ptr 0 w)
        return $ MakePS0 cod 1 0 1
```

Figure 13. Extracted cons Haskell function

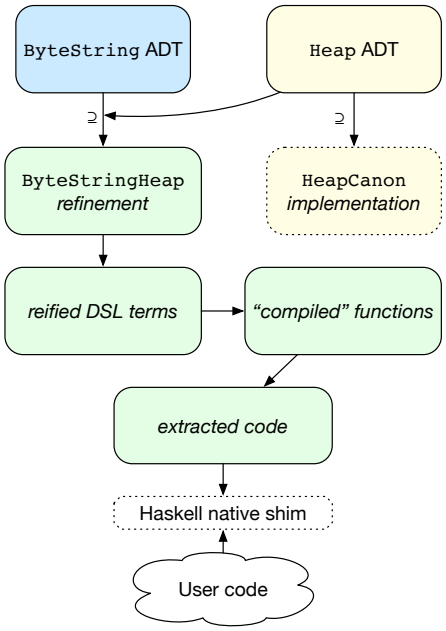


Figure 14. Relationship of abstract data types

was made for an earlier version of the project, before moving to stateful refinements, we make note of the results here since they helped guide the course of development.

Although ByteStringCanon gave us a definition that could be extracted to Haskell and used correctly, the implementation was still woefully inadequate. The heap it uses is based on a private, map-based construction, and not the runtime heap used by the GHC compiler; its notion of free pointers is too naïve (they only move forward); and it treats pointer addresses as raw integers, meaning any association with the garbage collector is indirect, whereas full performance mandates we integrate with it directly.

To apply these optimizations to ByteStringHeap, we had to step back and assess what the refinement represents: An implementation of ByteString that manipulates heaps using an interface defined by the Heap abstract data type. Since the final heap we wish to use is not defined in the proof environment—being an entity known only to the GHC runtime—we cannot use refinement to inject its method calls in a principled way. Although we could use axioms to define

```
Definition MyFunction
    (r : rep) (addr : Ptr Word) (val : Word) : rep :=
  let heap := fst r in
  heap <- poke heap addr 0 val;
  heap <- poke heap addr 1 val;
  ret heap.
```

Figure 15. Basic heap client using the nondeterminism monad

```
fun (r : rep) (addr : Ptr Word) (val : Word) =>
  Join (fun (heap : Rep HeapSpec) =>
    Join (fun (heap : Rep HeapSpec) => Pure heap)
      (Call <index of poke method> [heap; addr; 1; val]))
    (Call <index of poke method> [fst r; addr; 0; val])
```

Figure 16. Basic heap client, reified as free algebra term

the various heap methods, direct use of those axioms would be arbitrary, causing us to lose all the properties we had established thus far.

Since the GHC heap's own semantics are not known with certainty, we simply cannot refine programs that use it within the realm of proof. What we can achieve, however, is a more principled refinement by reducing the amount of trusted code. In the case of a direct refinement from ByteStringHeap to GHC's heap, the entire refinement would need to be trusted, owing to the degree of reliance on axiomatized definitions. We propose instead a different approach that relies on the key insight that ByteStringHeap only relies on the Heap ADT's public interface and semantics. This allowed us to reify the definition of ByteStringHeap into a free algebra over calls made to the Heap interface.

To refresh the reader on the notion of free monad algebras: A monad in functional programming can, with some caution, be viewed as something that "computes" when a term of type $m\ (m\ a)$ is collapsed to $m\ a$. This is how monads carry context through a sequential chain of computations: because at each point in the series, the context from the previous call is collapsed with the next.

The *free monad* [2] is a construction satisfying the monad laws but nothing more: it never collapses, or performs computation; it simply builds up a nested set of functor-shaped layers. That is, for a given functor f, a value in the free monad over that functor effectively has type $f(f(\ldots(f\ a)))$. In the case of a regular monad, these multiple f layers would be collapsed immediately during composition, by calls to *join*, but under the free monad this structure is preserved for later analysis and reduction. This allows the *meaning* of the construction to be deferred.

For the present case, the basic transformation is to change the representative snippet of code shown in Figure 15 into the value term shown in Figure 16. The advantage of this approach is that while the former could be reflected on using Ltac pattern matching, the latter is a deeply embedded term that we can evaluate directly. It also clearly delineates calls made to the Heap interface from any computations specific to the client code and isolates all uses of nondeterminism to such calls.

This reified ByteStringHeap is guaranteed to be a "proper client", that is, it never mutates the internal representation of the heap directly, since we have abstracted away all knowledge of a particular heap implementation. This representation is more abstract even than the implementation of ByteStringHeap using the nondeterminism monad, since during the creation of that refinement the representation type of Heap was made visible to the implementation.

With the reified term in hand, representing a pure functional program written using a deeply-embedded heap DSL, we can evaluate the term and render each abstract heap Call into its equivalent GHC heap call, referring to an axiomatic stub in each case. Although the GHC heap must still be axiomatized to be referenced in Coq, the mapping of DSL calls to GHC calls is now one-to-one and onto, removing any possibility of complexities introduced by the DSL term itself. Thus, every GHC heap method, such as malloc, is paired with its corresponding heap DSL construction, with no extra logic applied to any of its arguments.

Finally, this GHC-specific function compiled from the DSL term may now be extracted to Haskell code that is able to match—modulo alpha renaming and syntactic conventions—what a trained engineer would have written. And while the extraction process in Coq is not a verified subsystem (although, see future work in Subsection 7.3), the algebraic mapping from these compiled functions to Haskell means that any opportunity for error must be due to the extraction process, and not the function being supplied to the extractor.

A word should also be said on why certain functions were chosen during the extraction process. Some involve obscure choices, such as mapping malloc to mallocPlainForeignPtrBytes, rather than simply malloc. These choices were made after reading the existing bytestring source code to determine which tricks it is playing to achieve its speed, and then modifying the extraction process to map our interface onto those same functions. This results in code that is quite close to the original version, but with the major difference that we only need to do this fine-tuning in one place, the extraction mapping, rather than implement all of ByteStringHeap in terms of these specialized choices.

For example, in the case of pack (shown in Figure 2), there are several details taken directly from the existing bytestring library for the sake of efficiency, but implemented by way of almost direct correspondence with the simpler functions used in the Coq definition:

- unsafeDupablePerformIO requests execution of an IO action in an otherwise pure context, but is more efficient than unsafePerformIO because it omits the check that the IO is only being performed by a single thread;
- mallocPlainForeignPtrBytes returns a pointer to pinned memory—memory not moved by the garbage collector, meaning pointer references remain stable–that can be reclaimed by the garbage collector directly, without calling a function to release ownership of the memory block;
- withForeignPtr simply gives access to the underlying memory pointer;
- pokeArray directly writes the list of bytes into the memory region;
- the returned structure maintains a foreign pointer to the memory block, allowing it to be reclaimed when no longer referenced, and a note that the data begins at offset zero within the block, and extends for the length of the input list.

5 Evaluation

In order to evaluate the performance of the extracted program, a benchmarking program was written to compare the methods from the bytestring library to those of our extracted code. The tests proceed as follows:

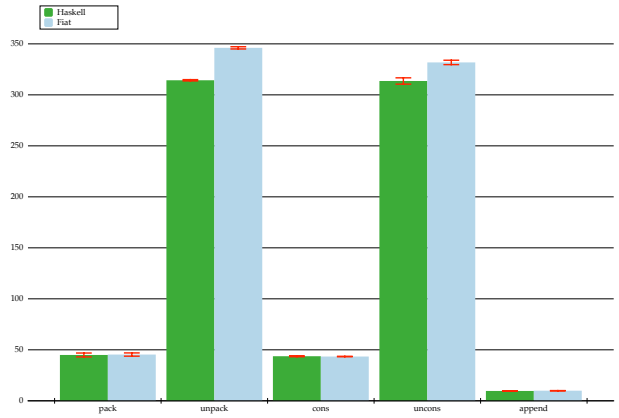


Table 1. Benchmark comparing time with Haskell's ByteString; scale is in seconds, smaller bars are better

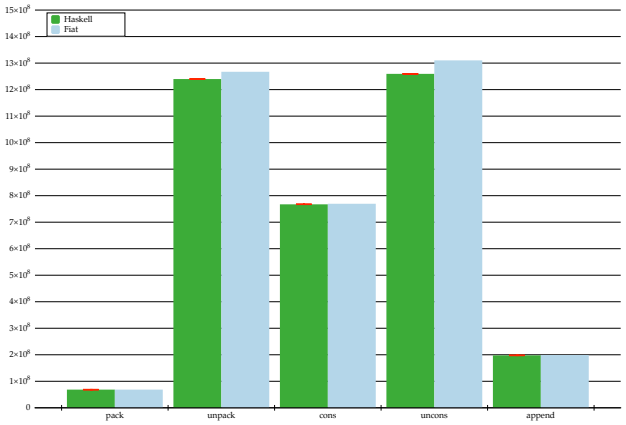


Table 2. Benchmark comparing allocated memory with Haskell's ByteString; scale is in bytes allocated (not resident), smaller bars are better

1. Convert each integer in the range 1 to 10^6 into its string representation;
2. Concatenate these strings, converting each digit character to a Word8 byte value;
3. Use these bytes to either iteratively construct a large ByteString or perform manipulations on such a ByteString after it is constructed.

Benchmarking results are given in Table 1 comparing time, and Table 2 comparing memory usage. The whole test series was run twice within the same process, to ensure the runtime was sufficiently primed (doing so made a significant impact on the first test). Note that only the pack method has been tuned thus far, leaving room for improvement in the other numbers, though the numbers for cons are already performing above expectation. The main reason for the discrepancy in the unpack and uncons numbers is that Haskell's bytestring library uses a special function called accursedUnutterablePerformIO. We could likewise use this method during extraction, but for the time being only unsafeDupablePerformIO is used, until the semantics of the former are better understood.

6 Related Work

Data Refinement Frameworks Hoare [10] 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 [9] later extended this approach to use relations and nondeterminism, which has been realized in implementations in both Coq [4] and Isabelle [11, 13]. Both frameworks allow for *arbitrary* refinement of data types, requiring the transport of data type refinements across the entire program. In contrast, Fiat's approach restricts the data refinement to the representation types of ADTs, allowing clients to use derived implementations without change. This restricted style of data refinement is particularly well-suited for our approach to mixed-language development. Additionally, both of these frameworks target pure functional programs and do not support fine-grained control over the layout of a data structure on the heap. Recent work [12] has extended the Isabelle Refinement Framework with support for refining to an embedded imperative language; this approach relies on the use of garbage-collected heap objects, however.

Extraction of Formally Verified Programs Coq has long supported proof-erasing extraction of functions [15] to both OCaml and Haskell to build executable binaries of formally verified code, a feature we use to produce our Haskell implementations. Extraction to OCaml is the final step of many certified developments in the Coq proof assistant, including CompCert [14] and the FSCQ file system [3]. An important novelty of this work is our support for heap-manipulating foreign function calls by specifying the foreign function interface as an ADT. Another approach to program extraction is found in CakeML [19], which extracts ML programs from pure HOL4 functions and supports foreign function calls via an interface that models how foreign function calls can manipulate the environment. More recently, the SpaceSearch project [21] certifies programs written against an ADT interface, which are converted to calls to a solver implementing that interface in the extracted program, akin to the heap ADT interface used here.

Haskell Refinement Types As an aid to Haskell programmers, the LiquidHaskell [20] project provides an optional type checker that uses code annotations to assert properties of types, such that LiquidHaskell can analyze program source code to determine whether these assertions are maintained. LiquidHaskell's first case study also verified the correctness of the bytestring library, where the authors also mention "[bytestring's] pervasive intermingling of high level abstractions like higher-order loops, folds, and fusion, with low-level pointer manipulations in order to achieve high-performance".

A key difference between the refinement types approach and this work is that LiquidHaskell's correctness properties are applied to the optimized code, requiring a complete understanding of the code's semantics during the annotation process. For example, annotations are applied to the pointer math performed in the bytestring library, rather than at the higher level of ByteString's list-like semantics.

7 Discussion and Future Work

Specifying critical code in a proof environment and then deriving high-performance implementations presents novel cognitive burdens on the would-be systems designer. Since this library constitutes a first attempt by the authors to refine Coq specifications into highly optimized Haskell, we would like to review some of the hurdles encountered during the process, and also some future directions for improving the utility of this methodology.

7.1 The Formalism Gap

We make a number of assumptions that could impact the correctness of the implementation of ByteString we derive here. First, we assume that the ADT we use to model the heap is a faithful model of the heap provided by the Haskell runtime. While we believe this to be the case, we could increase our confidence by extracting a canonical implementation of the heap and rigorously testing its fidelity. Another gap exists between the stateful semantics presented in Subsection 3.1, which implicitly threads state through the execution, and the implementation in Fiat, which uses an explicit state value. At present, there is nothing preventing our implementation from reusing the same state value twice or disregarding effects. While we have reviewed the extracted code to determine this is not the case, we are investigating the development of a linear type system for Fiat that would shield clients from such harmful behaviors. Finally, the correctness of the Haskell implementation depends on both Coq's proof checker and its Haskell extraction mechanism.

7.2 Lessons learned

Choice of representation type The choice of internal representation type used by the top-level specifications is crucial, since this type influences all subsequent proofs and the course of the refinement process. Choosing a type with broader semantics than the abstract data type being implemented necessitates pinning down exactly which part of those semantics is necessary for correctness in the refinement relation. For example, we initially used mathematical sets of indexed bytes for the first iteration, resulting in significant time lost proving characteristics of this formulation, until it was discovered that inductively defined lists already present exactly the semantic content needed.

Canonical refinements Although not used in the final product, providing canonical refinements of supporting abstract data types can serve as a check against unimplementable specifications, giving assurance that at least one implementation exists. This step might be considered the "proof of soundness" for a specification, and can largely be automated if the representation type's theories are well-established. In the first iteration of the library, such proofs for Heap were not done, resulting in several difficulties during later refinement when it was hard to recognize that the impossibility of certain proofs was due to the misspecification of the Heap abstract data type.

7.3 Future directions

Client and ADT co-refinement The present work refines a set of ADTs to a final implementation for use by Haskell clients, limiting the optimizations we may perform to those that are valid for any such client. For example, the composition uncons . cons x must extract to code that creates a temporary value, in the hopes that GHC might realize the equivalence with Just . (x,).

If certain client functions were also specified using Fiat, and co-refined against ADT definitions known to Coq, these fusion opportunities could be tuned for specifically, without relying on

generalized mechanisms within the Haskell compiler, or the necessity of setting up rewrite rules in the hopes they might apply during compilation. Such client functions would represent "supercompilation by refinement", where opportunities like fusion can be applied with as fine a granularity as needed, independent of generalized mechanisms. Especially where both correctness and performance are needed in key situations, this introduces yet another advantage to having formally specified one's library code in the manner given above.

Compilation to GHC Core In future work, rather than extract from a compiled function using an axiomatized GHC heap to Haskell, we hope to compile to the same GHC Core language Haskell itself compiles to, using a formal model of that language so that the possibility for error is reduced to whether our understanding of GHC's runtime heap semantics, as offered by its methods, corresponds to the semantics defined by the `Heap` ADT. This approach would bypass Coq's informal extraction mechanism, ensuring the final GHC Core program is precisely what is expected.

Multiple optimization strategies We hope to support configurable optimization strategies, allowing for the production of multiple bytestring libraries from a single specification tuned to varying preferences of CPU or memory performance. Going one step further, we should also be able to replace other string-like libraries, such as the text library, since all of these represent identical semantics to the bytestring library, differing primarily in the element type they range over, internal representation and optimization strategies; yet the underlying semantics remains "a list of elements" in all these cases.

Improved automation Fiat's automation mechanisms are covered in detail in prior work [5] and establish that code generation from a rigorous proof environment need not be as labor intensive as constructing correct programs whole cloth using dependent types. Once a particular programming domain, e.g. the theory of lists, has been well-established by a proof engineer, this domain can be used to automate much of the process of refinement. Already, the algebraic DSL terms compiled during our case study were created with almost no human involvement at all. It is intended for future work that this should characterize much of the refinement process, with the only manual steps being the creation of the initial specification, the choice of final representation type and how best to optimize for it, and the association of DSL terms with their external, Haskell counterparts.

Completing the library For our initial case study, only the essential methods of `ByteString` were implemented and put through the compilation process. We would like to map out the rest of the library, while at the same time providing tools for writing library functions in terms of core abstract data types, in such a way that these library functions can take advantage of the same type of optimization tuning, but without any knowledge of the representation type used in the primary specification.

8 Conclusion

We have shown that an ADT specified by a high-level semantics in Coq can be related to an optimized Haskell program in a way that demonstrates preservation of those semantics in the final implementation—with the caveat that the operational semantics must still be well understood if runtime effects are relied on.

Further, the benefit of separating semantics from execution in this way leads to clearer and simpler specifications of core program behavior. It provides a clean separation between the abstract, mathematical description of what a program does, and the low-level details of how this is realized in a high performance setting. One can imagine a division of labor between the programming language theorist who dwells mostly in the realm of the abstract, and the practitioner who is intimately familiar with the resource requirements of specific platforms. Whereas previously the information communicated between these two was *ad hoc*, if at all, the Fiat system provides a formal setting where the efforts of both can serve as inputs to a common product.

In sum, we have shown the use of Fiat to construct a subset of the Haskell bytestring library that closely matches the performance of hand-optimized code, while formally connecting that implementation to correctness guarantees and proof results developed in Coq. This demonstrates that adding dependent types to Haskell is not strictly necessary to leverage the power of such types, and that Coq can be used for what it does best to fill that gap. This represents a feasible methodology for producing high assurance yet high performance code, without placing undo cognitive burden on one person to master every technique involved. Rather, the work can be safely divided between proof engineers and system engineers, knowing that the formal guarantees of proof connect the two.

9 Acknowledgements

We would like to thank the other members of the MIT Fiat team (Adam Chlipala, Clément Pit--Claudel and Jason Gross) for their ongoing assistance and encouragement: from the initial idea of building a Haskell library based on refinement, to its present form. We are also grateful to the anonymous reviewers for their helpful and clarifying comments.

This work was sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007.

10 Implementation

To browse the source code for this project, or build and use it on your own system, the current version is located on Github at: https://github.com/jwiegley/bytestring-fiat.

References

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 62–73. DOI: http://dx.doi.org/10.1145/1088348.1088355

[2] S. Awodey. 2006. *Category Theory*. Ebsco Publishing. https://books.google.com/books?id=IK_sIDI2TCwC

[3] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. DOI: http://dx.doi.org/10.1145/2815400.2815402

[4] Cyril Cohen, Maxime Dns, and Anders Mrtberg. 2013. Refinements for Free! In *Certified Programs and Proofs*. Springer International Publishing.

[5] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. *Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant*. Association for Computing Machinery. http://dspace.mit.edu/handle/1721.1/91993

[6] 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.

[7] Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. *CoRR* abs/1610.07978 (2016). https://www.cis.upenn.edu/~sweirich/papers/eisenberg-thesis.pdf

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

[9] 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.

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

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

[12] 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. DOI: http://dx.doi.org/10.1007/978-3-319-22102-1_17

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

[14] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. DOI: http://dx.doi.org/10.1145/1538788.1538814

[15] Pierre Letouzey. 2003. A New Extraction for Coq. In *Proc. TYPES*. Springer-Verlag.

[16] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast Synthesis of Fast Collections. *SIGPLAN Not.* 51, 6 (June 2016), 355–368. DOI: http://dx.doi.org/10.1145/2980983.2908122

[17] Robert Paige and Shaye Koenig. 1982. Finite Differencing of Computable Expressions. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982).

[18] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. *SIGPLAN Not.* 44, 6 (June 2009), 408–418.

[19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 60–73. DOI: http://dx.doi.org/10.1145/2951913.2951924

[20] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. *SIGPLAN Not.* 49, 12 (Sept. 2014), 39–51. DOI: http://dx.doi.org/10.1145/2775050.2633366

[21] Konstantin Weitz, Steven S. Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. In *Proc. of the ACM Program. Lang. (ICFP '17)*, Vol. 1. ACM.

[22] Edward Yang. 2010. How to pick your string library in Haskell. http://blog.ezyang.com/2010/08/strings-in-haskell/. (2010).