# Correct-by-Construction Program Derivation from Specifications to Assembly Language

## Abstract

We present a Coq-based system to certify the entire process of implementing declarative mathematical specifications with efficient assembly code. That is, we produce formal assembly-code libraries with proofs, in the style of Hoare logic, demonstrating compatibility with relational specifications in higher-order logic. Most code-generation paths from high-level languages involve the introduction of garbage collection and other runtime support for source-level abstractions, but we generate code suitable for resource-constrained embedded systems, using manual memory management and in-place updating of heap-allocated data structures. We start from very high-level source code, applying the Fiat framework to refine set-theory expressions into functional programs; then we further apply Fiat's refinement tools to translate functional programs into Facade, a simple imperative language without a heap or aliasing; and finally we plug into the assembly-generation features of the Bedrock framework, where we link with handwritten data-structure implementations and their associated proofs. Each program refinement leads to a proved Hoare-logic specification for an assembly function, with no trust dependencies on any aspect of our synthesis process, which is highly automated.

## 1. Introduction

One of the fundamental sources of progress in programmer productivity is the introduction of new abstractions. SQL databases, for instance, are commonly applied to manage persistent relational data via high-level notation for queries and updates. Unfortunately, the challenges of efficiently implementing a database engine, e.g. the selection of low-level data structures, concurrency management, etc., are so daunting that most programmers would never consider extending an engine with new features for fear of breaking it.

The situation for compilers is broadly similar. The average programmer dares not modify a compiler for languages like C or Java. It is too easy to introduce bugs that generate wrong code for programs that used to be compiled correctly. Still, often one very specific, custom optimization could make a large performance difference, and we might wish for a compiler-extension methodology that guarantees soundness. For instance, the usual strategy of a Java compiler may be less suitable for embedded systems with significant resource constraints. We might prefer to have the programmer explain clever rules for implementing particular high-level code patterns with low-level code, if we can be sure that no unsound rules are admitted into the system.

These two examples fall within a general pattern. We have some source language, more or less declarative, but always with a formal semantics. Some tool implements efficient translation to a different language, itself with a semantics. In a pipeline of several such tools, the semantics provide a theoretical basis for reasoning about end-to-end translation correctness. A variety of tools make sense from that perspective, including generators for parsers and pretty-printers for programming languages and network-protocol wire formats. The overall picture gets more interesting when we move beyond a traditional pipeline of black-box C compiler, parser generator, etc., and aim to make each translation tool *extensible*. Programmers should be able to safely teach compilers new tricks when they can provide evidence that the tricks are sound.

In this paper, we present **an end-to-end methodology for automatic derivation of efficient assembly code from relational specifications, where each of our several key stages supports correct-by-construction extension with new derivation strategies.** Our framework is implemented inside the **Coq proof assistant**, and our final theorems about assembly programs have very small trusted bases that completely exclude all synthesis tools and compilers. We use Coq's Turing-complete tactic (i.e., proof procedure) language Ltac [Delahaye 2000] as a unified setting for describing correct-by-construction program transformations; we provide libraries of both verified program refinement rules and heuristics for applying them in sequence, and any user of our framework may (soundly) add new rules and heuristics, while reusing ours as building blocks. As a case study, we demonstrate a unified tactic for refining set-theory specifications of list functions into efficient assembly code, completely automatically.

This paper combines three prior Coq libraries to implement the methodology described above. The initial stages

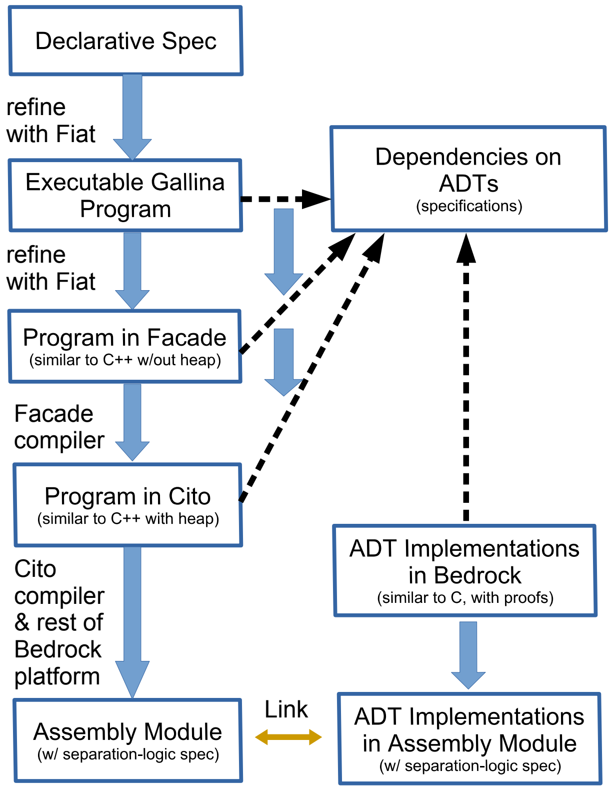

Declarative Spec
refine with Fiat
Executable Gallina Program
refine with Fiat
Program in Facade (similar to C++ w/out heap)
Facade compiler
Program in Cito (similar to C++ with heap)
Cito compiler & rest of Bedrock platform
Assembly Module (w/ separation-logic spec)
Link
Dependencies on ADTs (specifications)
ADT Implementations in Bedrock (similar to C, with proofs)
ADT Implementations in Assembly Module (w/ separation-logic spec)

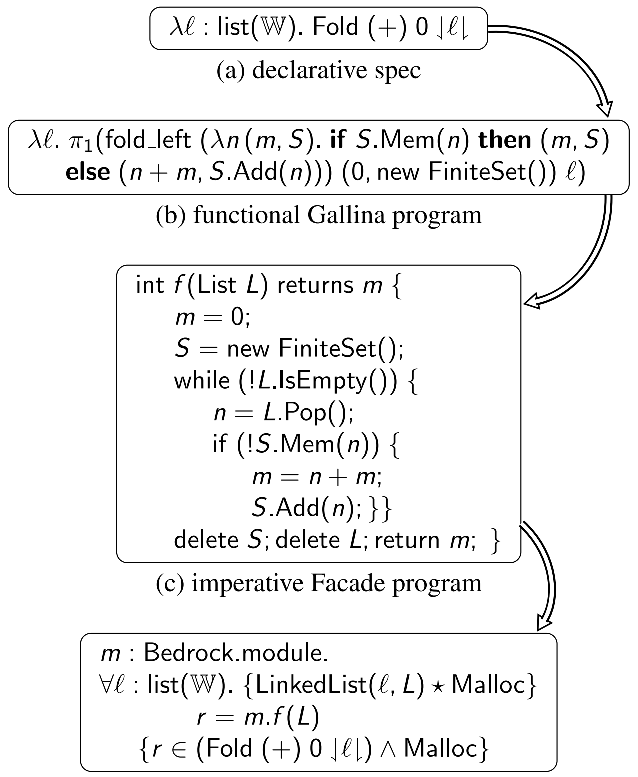
**Figure 1.** Synthesis and compilation pipeline

of this pipeline use **Fiat** [Delaware et al. 2015], a library for correct-by-construction program derivation by stepwise refinement. Starting from concise specifications in set theory, Fiat synthesizes source-code programs in Gallina, Coq's functional programming language. The **Cito** [Wang et al. 2014] programming language is the foundation for the intermediate stage of this pipeline. Cito is a programming language in the style of C++, with explicit support for data abstraction when linking against assembly code compiled from multiple programming languages. The target of the Cito compiler is the **Bedrock** framework [Chlipala 2011, 2013; Malecha et al. 2014], a Coq library for orchestrating implementation and verification of multilanguage programs, bottoming out in assembly language with a common module interface formalism. This paper demonstrates how to connect these pieces to form a certified pipeline from high-level specifications to assembly language implementations, where we are able to link our synthesized program modules (and their proofs) with modules written (and verified) using a variety of languages.

Figure 1 sketches the high-level structure of our certified pipeline. We overview the elements here and then return to them in detail in dedicated sections.

1. We begin with *declarative specifications in the succinct style of set theory*. Our concrete case study works with functions over lists (i.e., mathematical sequences).
2. The first compilation step is to *refine specifications into functional programs* (see Section 2). These functional programs are written in Gallina, the pure ML-like language that doubles as Coq's logic. While the original specs are "written in math" and so not directly executable, the functional programs we generate are efficient, modulo availability of good implementations of key abstract data types (ADTs), like finite sets. Outputs of this stage record their ADT dependencies explicitly, via abstract specifications in the Fiat style [Delaware et al. 2015].
3. Next, we *refine Gallina programs into imperative programs* (see Section 4) in a novel intermediate language, **Facade**. Facade is a C++-style language with no explicit heap. Rather, all program state lives within the local variables of dynamic function calls. Local variables are mutable and may be passed by reference in function calls. This refinement process uses libraries of hints to implement various functional-programming constructs in an imperative language. Crucially, the results do not rely on garbage collection, instead synthesizing correct-by-construction code that does manual memory management and in-place updating of data structures.
4. Composing our new Facade compiler (see Section 5) with the existing Cito compiler [Wang et al. 2014], we *generate Bedrock assembly modules* from the Facade code output by the prior step. The major hurdle in verifying this compiler stems from the semantic gap in how the two languages represent memory. Cito exposes a heap with pointers and aliasing, while Facade has a more disciplined world where state is pointer-free and lives only in local variables. We found this staged introduction of mutable state supports a convenient decomposition of proof effort.
5. Throughout the last three stages, dependencies have been maintained on arbitrary implementations of ADTs. We may implement ADTs in Fiat, but it is also pragmatic to *implement them directly in the C-like language Bedrock provides* [Chlipala 2013] and verify them using Bedrock's low-level proof automation [Malecha et al. 2014]. It is possible to achieve better performance with clever manual implementation than via our compilation pipeline.
6. Finally, we *link the assembly modules produced by synthesis and by semi-automatic verification* (see Section 6), leading to closed programs with small-trusted-base Coq correctness theorems that do not mention any of our compilation, synthesis, or verification machinery.

The middle stages of this pipeline constitute an alternative approach to compiling functional languages, especially relevant to resource-constrained platforms, and applicable outside the context of synthesis from specifications. The same phases are also of interest as an alternative to the *program extraction* feature of Coq [Letouzey 2003] and other proof assistants, where functional programs within a logic may be translated into more mainstream functional languages and compiled with their compilers. Our approach supports extraction-like functionality that is *more trustworthy*, since we produce Coq



$\lambda \ell : \mathsf{list}(\mathbb{W}).\ \mathsf{Fold}\ (+)\ 0\ \lfloor \ell \rfloor$

(a) declarative spec

$\lambda \ell.\ \pi_1(\mathsf{fold\_left}\ (\lambda n\,(m, S).\ \textbf{if}\ S.\mathsf{Mem}(n)\ \textbf{then}\ (m, S)$
$\textbf{else}\ (n + m, S.\mathsf{Add}(n)))\ (0, \mathsf{new\ FiniteSet}())\ \ell)$

(b) functional Gallina program

```
int f(List L) returns m {
    m = 0;
    S = new FiniteSet();
    while (!L.IsEmpty()) {
        n = L.Pop();
        if (!S.Mem(n)) {
            m = n + m;
            S.Add(n); }}
    delete S; delete L; return m; }
```

(c) imperative Facade program

$m$ : Bedrock.module.
$\forall \ell : \mathsf{list}(\mathbb{W}).\ \{\mathsf{LinkedList}(\ell, L) \star \mathsf{Malloc}\}$
$r = m.f(L)$
$\{r \in (\mathsf{Fold}\ (+)\ 0\ \lfloor \ell \rfloor) \wedge \mathsf{Malloc}\}$

(d) Bedrock assembly module (with separation-logic spec)

**Figure 2.** Translating an example spec to assembly

refinement proofs connecting programs to assembly code; and *amenable to producing higher-performance code*, since programmers may extend the system with specialized compilation strategies as proved-correct refinement rules.

As a more concrete example, consider Figure 2, showing a program at four key stages during its refinement into assembly code. Figure 2a shows the relational specification that we start with, formalizing the idea of computing the sum of all *unique* elements of a list of numbers. For convenience throughout the paper, we work with numbers in the domain $\mathbb{W}$ of 32-bit machine words. We write $\lfloor \ell \rfloor$ for the set of elements appearing in list $\ell$, and we write Fold for folding an associative-commutative operator over all elements of a set. The latter will be undefined for infinite sets. (While the code in Figure 2a looks like a normal functional program, it actually expands into set-theory notation denoting a set, which in this case always has exactly one element, but in general in Fiat may have zero or many elements.)

Skipping to the end in Figure 2d, we see the final product: a module of assembly code within Bedrock's specification formalism. In particular, the final module $m$ contains a code label $f$, and within the module is a proof of a specification for this label, in the style of Hoare logic. In particular, we follow separation logic [Reynolds 2002], a flavor of Hoare logic with good support for reasoning about heap-allocated data structures with potential aliasing. The spec in Figure 2d gives a *precondition* and *postcondition* for an arbitrary call to the function $f$ on an (untyped) argument pointer $L$. The precondition asserts that $L$ is the root of some singly linked list, representing mathematical list $\ell$, which we quantify over and connect to the pointer using an *abstract predicate* [Parkinson and Bierman 2005] that encapsulates the details of some pointer-based, imperative representation of lists with associated methods. We use separation logic's *separating conjunction* $\star$ to express that, when calling the function, in some part of memory disjoint from the list $L$, there must appear the data structures of a standard malloc library. This particular synthesized code will use malloc and free both to allocate a finite set as a temporary data structure and to free the memory associated with the input list. Finally, the postcondition says that the function return value $r$ contains exactly the answer prescribed by the original spec of Figure 2a, and some valid malloc state remains in memory (though the list has been deallocated).

The figure also highlights some key intermediate steps in the derivation. Figure 2b shows a first executable version, as a pure functional program. The key idea is to introduce an intermediate data structure, a finite set recording which numbers have already been encountered as we traverse the input list with a fold_left. Rather than hardcoding use of a particular implementation of finite sets, we parameterize the program over an arbitrary correct implementation of a Fiat ADT spec. Later, we plug in an implementation in a C-like language within Bedrock.

Another interesting step connects Figures 2b and 2c, where we have produced code in Facade, an imperative language with manual memory management. The expanded code more or less follows the structure of Figure 2b, but we must account explicitly for object lifetimes and mutation. For instance, we traverse the list $L$ with a loop, where we call the list method Pop() to remove the first element of the list, returning that element as the method result. At the end of the function, we must deallocate the list with delete. Our derivation process guarantees that we manage memory correctly while preserving the behavior of the original functional program and, transitively, the relational spec.

We will present the techniques that allow us to refine from the beginning of the figure to the end, *automatically*, while still generating rigorous Coq proofs from first principles, where the guarantee about the final assembly program may be understood without reference to any of the programming or proving tools we have developed for this project in particular. The dependencies are just on the syntax and semantics of an assembly language (the Bedrock IL [Chlipala 2013]).

To summarize the contributions of this paper:

- We provide **the first mechanically certified translation pipeline from declarative specifications to assembly-language libraries**.
- We introduce **a proof-generating approach to deriving imperative programs from functional programs within a general stepwise refinement framework**.

- We present a novel **imperative intermediate language with a formal semantics and verified compiler**, supporting gradual bridging of the gap between pure functional code and pointer-based imperative code.
- The pipeline overall supports **abstraction over ADT implementations, compatible with satisfaction via handwritten (and hand-verified) low-level code**.

The remainder of the paper introduces the stages of our pipeline in order and then discusses our empirical evaluation.

## 2. Correct-by-Construction Functional Code

Our goal is to synthesize assembly-level code from high-level specifications of set-manipulating programs. These specifications are written using the FiatL DSL:

$$S \coloneqq x \mid S \cup S \mid S \cap S \mid S - S \mid \{x \mid P\,x\} \mid \upharpoonleft S \upharpoonright \mid \downharpoonleft S \downharpoonright \mid \textbf{Fold}\,f\,i\,S$$

FiatL includes the standard set operators plus operators for mapping lists to sets and vice-versa, respectively $\downharpoonleft \_ \downharpoonright$ and $\upharpoonleft \_ \upharpoonright$, and an operator Fold that folds an associative-commutative operator over all elements of a set. This "DSL" is actually just a set of combinators defined in Gallina, Coq's functional-programming language, so all the usual Gallina features are available implicitly, for instance for doing arithmetic or defining Fold functions anonymously.

The initial stage of compilation is one of *refinement*: a high-level specification in this DSL that admits a number of different implementations is iteratively refined to produce a single (and hopefully efficient) implementation in a high-level functional language.

***Foundations of Computational Refinement*** Fiat represents high-level specifications of set-manipulating functions as nondeterministic *computations*, or sets of values satisfying some defining property. We have the usual set-comprehension notation $\{\_ \mid \_\}$ available, along with two monad-style [Wadler 1992] combinators **ret** for building a singleton set and "bind", written "$\_ \leftarrow \_; \_$", combining two computations.

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

Consider the following specification of a function that filters out any words in a list $ls$ above a threshold $x$:

$$\mathsf{LtUniqueSpec}\ ls\ x \equiv \{l \mid \downharpoonleft l \downharpoonright = \downharpoonleft ls \downharpoonright - \ \{w \mid w < x\}\}$$

This specification admits a number of implementations by not fixing the order of the filtered list and by using the nonexecutable set-minus operator to removes elements from $ls$. Thus, a function implementing LtUniqueSpec could choose to implement the set subtraction as a filter over $ls$ or $\mathsf{rev}(ls)$. Either choice represents a more refined version of LtUniqueSpec, with refinement defined by the subset relation $\subseteq$ on the set of implementations allowed by each specification. Intuitively, a computation $c'$ is a refinement of a computation $c$ if $c'$ only "computes" to values that $c$ can "compute" to.

***Deductive Synthesis*** The end result of synthesis is a *sharpened* or fully deterministic computation that computes to precisely one value, coupled with a proof that it is a valid refinement of such a specification:

$$\mathsf{sharpened\ Spec} \ \equiv\ \Sigma\,\mathsf{Impl}.\ \mathsf{Spec} \supseteq \mathsf{Impl}$$

A derivation is simply a user-guided search for the two components of this dependent pair.

The transitivity and reflexivity of refinement allow us to decompose a derivation into a sequence of applications of basic refinement facts. The search for an implementation of a specification $C_{spec}$ consists of transitively applying refinement facts drawn from a database until a deterministic implementation is found, at which point reflexivity finishes the proof of refinement:

$$\dfrac{\dfrac{}{C_{spec} \supseteq C_1}\,\textsc{Lem}_0 \qquad \dfrac{\dfrac{}{C_1 \supseteq C_2}\,\textsc{Lem}_1 \qquad \begin{matrix}\dfrac{}{C_{impl} \supseteq C_{impl}}\,\textsc{RRefl} \\ \vdots\end{matrix}}{C_1 \supseteq C_{impl}}\,\textsc{RTrans}}{C_{spec} \supseteq C_{impl}}\,\textsc{RTrans}$$

It is important to note that while building this proof trace interactively, $C_{impl}$ is unspecified (i.e. a unification variable): it is only when the user is satisfied with the implementation and applies RREFL that $C_{impl}$ is instantiated with an implementation. Refinement facts such as $\textsc{Lem}_0$ and $\textsc{Lem}_1$ are simply Coq theorems whose mechanically checked proofs are built in the standard way.

***Interfacing with high-assurance foreign code via ADTs*** Code synthesized by Fiat is parameterized by a set of abstract data types [Liskov and Zilles 1974] (ADTs) used to interface with external code. For instance, when we generate implementations of FiatL specs, we assume an implementation of an ADT of finite sets.

Fiat defines ADTs as records of state types and computations implementing operations over states. Figure 3 gives a specification of the finite-set ADT used by FiatL. In the signature of ADT methods, **rep** stands for an arbitrary abstract implementation type. The FiniteSetSpec functional specification is a *nondeterministic reference implementation* that uses mathematical sets as its representations type; its associated method implementations use standard set operations to express how any implementation of this ADT must behave. Here we use Coq's Ensemble type family, which is the equivalent of the informal idea of a mathematical set, with elements drawn only from a specified type.

While mathematical structures are convenient for specifying methods, they are unsuitable for an implementation. Fiat uses *abstraction relations* [He et al. 1986; Hoare 1972] to

```
Definition FiniteSetSpec : ADT FiniteSetSig :=
 ADTRep (Ensemble W) {
  Def Constructor Empty (_ : unit) : rep := ret ∅,
  Def Method Add (xs : rep , x : W) : unit :=
    ret (xs ∪ {x}, tt),
  Def Method Remove (xs : rep , x : W) : unit :=
    ret (xs − {x}, tt),
  Def Method In (xs : rep , x : W) : bool :=
    b ← { b : bool | b = true ↔x ∈ xs }; ret (xs, b),
  Def Method Size (xs : rep , _ : unit) : W :=
    n ← | xs | ; ret (xs, n) }.
```

**Figure 3.** Fiat specification of finite-set ADT

justify refinement of representation types. An abstraction relation A ≈ B between two ADTs sharing a common interface is a binary relation on the representation types of A and B that is preserved by each method. In other words, the operations of the two ADTs take similar input states to similar output states. Since operations in Fiat are implemented as computations, the methods of B may be computational refinements of A. Thus, an ADT method B.m is a refinement of A.m if (taken from [Delaware et al. 2015])

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

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

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

$$\mathsf{A} \simeq \mathsf{B} \quad \equiv \quad \forall \mathsf{m}.\ \mathsf{A.m} \simeq \mathsf{B.m}$$

The relation $\simeq$ is indexed by the abstraction relation $\approx$, so that we write A $\simeq_\approx$ B to indicate that relation $\approx$ demonstrates the compatibility of A and B. Thus, we define refinement formally as:

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

An implementation of a reference ADT implementation A is simply an ADT B that is a valid refinement of A whose methods always compute to precisely one value:

$$\mathsf{SharpenedADT\ A} \ \equiv\ \Sigma \mathsf{B}.\ \mathsf{A} \simeq \mathsf{B} \wedge \forall \mathsf{m}, \mathsf{x}.\ \exists \mathsf{v}.\ \mathsf{B.m}(x) \subseteq \mathbf{ret}\ \mathsf{v}$$

***Automated Derivation*** The core of Fiat includes a collection of theorems proving basic refinement facts for constructing these proof trees, which we have augmented with additional refinement rules about set operations and the ensemble and list conversion functions. Fiat automates mechanized derivations using Coq's setoid rewriting tactics, which extend Coq's rewriting machinery with support for partial-order relations other than Leibniz equality[1]. We have also written a library of automation tactics synthesizing implementations of specifications of set-manipulating programs. These tactics allow a multitude of fully automated examples, where many FiatL programs can be compiled in full to functionally correct Gallina code with a single tactic.

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

These tactics intelligently chain together rewrites with refinement lemmas to implement a specification incrementally by looking for subterms to rewrite. For example, the library includes a proof of the following fact: nondeterministic choice of the size of a set $S$ (expressed noncomputationally with a set-theory expression $|S|$) can be implemented by picking an equivalent list $l$, converting it into a finite set using some implementation of the ADT from Figure 3, and calling the Size method on the result:

$$|S| \ \supseteq\ l \leftarrow \{l \mid l \approx S\};\ \ f \leftarrow \{f \mid f \approx l\};\ \ \mathbf{ret}\ f.\mathsf{Size}()$$

We also include another refinement lemma showing how to build a finite set equivalent to a list:

$$\{f \mid f \approx l\} \ \supseteq\ \mathbf{ret}\ (\mathsf{fold}\ (\lambda x, f.\ f.\mathsf{Add}(x))\ \mathsf{Empty}\ l)$$

Our tactics combine these two lemmas automatically to build a function that counts the number of unique elements in a list by folding over the list to add the elements to a finite set, then returning the size of that finite set.

## 3. Bridging Gallina and Bedrock via Facade

We could stop here by extracting to OCaml and compiling the resulting code, but we want to push our Coq correctness proofs all the way down to the assembly level, in addition to freeing ourselves from trusting Coq extraction or the OCaml compiler or runtime system. The high-level approach is to synthesize an equivalent program in Facade, a new untyped imperative language without the complications of a heap, pointers, aliasing, or memory leaks. Facade additionally supports linking with external code by means of axiomatic ADT specifications. Those specifications may be implemented and verified with any tools connected to the Bedrock framework.

This crucial pair of features creates a convenient fit with the output of the refinements introduced in the prior section: we may mimic the structure of the ADT-parameterized, functional Gallina programs fairly easily, but we are also reasonably close to the level of assembly code, so we make genuine progress towards the machine level by translating between the two languages.

The name we chose for Facade suggests that it is a light wrapper on top of something else, and that something else is the Cito language, for which a to-Bedrock compiler was verified in prior work [Wang et al. 2014]. Cito is an idealized C++-like language with a simplified view of memory. In particular, in contrast to the C-style view of memory as a set of objects that contain primitive values like integers, Cito presents a heap of objects associated with ADTs. Each object is represented with a mathematical model, for instance with imperative finite sets represented with mathematical sets,

$$\begin{array}{rcll}
\text{Constant} & w & \in & \mathbb{W} \\
\text{Label} & l & \in & \mathbb{S}_{\text{module}} \times \mathbb{S}_{\text{fun}} \\
\text{Variable} & x & \in & \mathbb{S} \\
\text{Binary Op} & o & ::= & + \mid - \mid \times \mid = \mid \neq \mid < \mid \leq \\
\text{Expression} & e & ::= & x \mid w \mid e\ o\ e \\
\text{Statement} & s & ::= & \mathsf{skip} \mid s; s \mid \mathsf{if}\ e\ \{s\}\ \mathsf{else}\ \{s\} \mid x := e \\
 & & & \mid \mathsf{while}\ e\ \{s\} \mid x := \mathsf{call}\ l\ (\overline{x}) \\
\text{Function} & f & ::= & \mathsf{fun}\ (\overline{x})\ \mathsf{returns}\ x\ \{\ s\ \}
\end{array}$$

**Figure 4.** Facade syntax

and imperative linked lists represented with mathematical sequences. The semantics is parameterized over an arbitrary set of ADTs with mathematical models and associated methods.

Cito has a mixed operational-axiomatic semantics: a program operates in a context $\Psi$ that assigns specifications to function identifiers, where some functions are assigned Cito statements as usual, called *operational* specs, but others may be assigned Hoare-logic preconditions and postconditions, called *axiomatic* specs. In the operational semantics, a call to a function with an operational spec proceeds as usual, but a call to a function with an axiomatic spec checks that the precondition is satisfied and then nondeterministically transitions to a post-state satisfying the postcondition. Axiomatic specs are written in the small-footprint style of separation logic, only mentioning parts of the heap that functions will touch, and the Cito operational semantics takes care of extracting, modifying, and replacing subheaps during calls to axiomatic functions. The correctness proof for the Cito compiler justifies these manipulations using separation logic, but translation steps at the Cito level and above may take that implementation for granted, reasoning only at the level of Cito's operational semantics.

We designed Facade to sit at an even higher level of abstraction, where the only mutable state is in the local variables of dynamic function calls. Variables storing ADT values are passed to methods by reference, but otherwise there are no pointer-like features. To support straightforward translation to Cito, we impose additional syntactic rules. For instance, it is not legal to copy an ADT value directly from one variable to another with an assignment, since, in the corresponding Cito code, such a step would introduce aliasing, which we want to shield the programmer from. Another rule prohibits passing the same ADT variable multiple times in different arguments to a single method call. Overall, the restrictions are designed to support the idea that mutable objects are uniquely associated with particular local variables in particular dynamic function calls, following a style that is common in C++.

Figure 4 gives the syntax of Facade, which is almost identical to that of Cito and includes the usual *statement* constructs for assignment, sequencing, conditional tests, loops, and function calls. We write $\overline{X}$ for a sequence of zero or more $X$'s. To simplify the semantics, *expressions* $e$ can only include safe and total operations like arithmetic, while function calls can be made only via dedicated forms of statements. A function call is made via a label, which consists of a module name and a function name. A function consists of a list of formal parameter names, a body, and a choice of a variable to return.

The key difference between Facade and Cito is in their treatment of machine states. Cito machine states include local-variable stores, mapping variables to integers (which may be used as pointers); plus heaps mapping pointers to ADT values. Facade retains only the local-variable stores, notated $\sigma$, mapping variables to either scalars (written as $\mathsf{SCA}$) or ADT values (written as $\mathsf{ADT}$). As in Cito, the Facade syntax and semantics are parameterized on a domain $A$ of ADT models. For our case study in this paper, $A$ includes both sequences and finite sets. Importantly, the members of $A$ are *mathematical models* of ADT values, not pointer-based *implementations*. Just as with Fiat ADT specifications, the Facade finite-set ADT is represented by a mathematical set, rather than with a balanced tree or hash table.

Figure 5 shows the Facade specification of the finite-set ADT, which is identical to the Cito specification. We define the model domain $A$ as a variant type including a constructor $\mathsf{FSET}$, for finite sets represented with Coq ensembles. The ADT has an associated set of methods, each with a precondition and postcondition. A precondition is *a function over the values of the actual parameters*, where each may be a scalar $\mathsf{SCA}$ or an ADT value $\mathsf{ADT}$. For instance, the precondition of add requires that the input argument list has length 2, with the first argument encoding a finite set $s$ and the second argument holding a scalar $w$. A postcondition is *a function over the before and after versions of the parameters, plus the return value*. The postcondition of add explains the evolution of the parameters, where the first argument has transitioned from representing set $s$ to set $s \cup \{w\}$. *After* values for scalar arguments (such as the second argument of add) are ignored by the semantics. Furthermore, we force the return value to be an arbitrary scalar, promising that it is not some ADT object that the caller would be responsible for deallocating. A more interesting constraint on the return value appears in the postcondition of new, where we assert that the return value is a fresh finite set representing $\emptyset$.

Figure 6 puts this notation to work in the Facade operational semantics, closely mirroring Cito's semantics. The general form is $\Psi \vdash (\sigma, s) \Downarrow \sigma'$, saying that statement $s$ may execute in initial state $\sigma$ and produce final state $\sigma'$, in context $\Psi$ that assigns specs to function labels. We omit $\Psi$ in rules where it is not used directly, implicitly threading it through all premises in the obvious way.

The middle rules of the figure are straightforward, giving the usual meanings of sequencing, conditionals, and loops. The first rule, for assignment, is unusual in requiring that (1) the expression being assigned must evaluate to some scalar, since blithely copying an ADT into a new variable would create aliasing in the pointer-based implementation underlying Facade; and (2) the target variable is not already

| | | |
|---|---|---|
| $A = \mathsf{FSET}(\mathbb{P}) + \cdots$ | | $\mathbb{P} = \mathcal{P}(\mathbb{W})$ (* sets of machine integers *) |
| $\{\lambda I.\ I = []\}$ | new | $\{\lambda(O,R).\ O = [] \wedge R = \mathsf{ADT}(\mathsf{FSET}(\emptyset))\}$ |
| $\{\lambda I.\ \exists s.\ I = [\mathsf{ADT}(\mathsf{FSET}(s))]\}$ | delete | $\{\lambda(O,R).\ \exists s.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \bot)] \wedge R = \mathsf{SCA}(\cdot)\}$ |
| $\{\lambda I.\ \exists s.\ I = [\mathsf{ADT}(\mathsf{FSET}(s))]\}$ | size | $\{\lambda(O,R).\ \exists s.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s))] \wedge R = \mathsf{SCA}(\lvert s\rvert)\}$ |
| $\{\lambda I.\ \exists s, w.\ I = [\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{SCA}(w)]\}$ | add | $\{\lambda(O,R).\ \exists s, w.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s \cup \{w\})),$ $(\mathsf{SCA}(w), \bot)] \wedge R = \mathsf{SCA}(\cdot)\}$ |

**Figure 5.** An example Facade ADT specification (finite sets) taken from [Wang et al. 2014]

$$\frac{[\![e]\!]_\sigma = \mathsf{SCA}(\_) \qquad \sigma(x) \neq \mathsf{ADT}(\_)}{(\sigma, x := e) \Downarrow \sigma[x \rightarrow [\![e]\!]_\sigma]}\ \textsc{Assign}$$

$$\frac{}{(\sigma, \mathsf{skip}) \Downarrow \sigma}\ \textsc{Skip} \qquad \frac{(\sigma, s_1) \Downarrow \sigma' \qquad (\sigma', s_2) \Downarrow \sigma''}{(\sigma, s_1; s_2) \Downarrow \sigma''}\ \textsc{Seq}$$

$$\frac{([\![e]\!]_\sigma \neq 0 \wedge (\sigma, s_T) \Downarrow \sigma')\ \vee\ ([\![e]\!]_\sigma = 0 \wedge (\sigma, s_F) \Downarrow \sigma')}{(\sigma, \mathsf{if}\ e\ \{s_T\}\ \mathsf{else}\ \{s_F\}) \Downarrow \sigma'}\ \textsc{If}$$

$$\frac{(\sigma, \mathsf{if}\ e\ \{s; \mathsf{while}\ e\ \{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}) \Downarrow \sigma'}{(\sigma, \mathsf{while}\ e\ \{s\}) \Downarrow \sigma'}\ \textsc{While}$$

$$\frac{\begin{array}{ccc}\Psi(l) = \mathsf{AX}(\mathsf{pre}, \mathsf{post}) & & \sigma(x) \neq \mathsf{ADT}(\_) \\ \mathsf{pre}(\sigma(\overline{y})) & \lvert\overline{v}\rvert = \lvert\overline{y}\rvert & \mathsf{post}(\sigma(\overline{y}) \rhd \overline{v}, r)\end{array}}{\Psi \vdash (\sigma, x := \mathsf{call}\ l\ (\overline{y})) \Downarrow \sigma[\overline{y} \rightarrow \overline{v}][x \rightarrow r]}\ \textsc{CallAx}$$

**Figure 6.** Operational semantics of Facade

$$\frac{(s_1, \sigma) \downarrow \quad \forall \sigma'.\ (\sigma, s_1) \Downarrow \sigma' \Rightarrow (s_2, \sigma') \downarrow}{(s_1; s_2, \sigma) \downarrow}$$

**Figure 7.** A selected rule of the Facade safety judgment

holding an ADT value, which we would need to deallocate first to avoid a memory leak under the hood.

The last rule is most interesting. It handles calls to functions with axiomatic specs. (The rule for operational specs, which we omit, is similar.) The premises say, in order: look up the spec for the callee, finding its precondition and postcondition; check the status of the variable $x$ where we will write the return value, making sure it does not already hold an ADT value (for the same reason as in basic assignment); verify that the actual arguments $\overline{y}$ satisfy the precondition; pick some *after* values $\overline{v}$ for the actual arguments, represented as a sequence of the same length; and check that the postcondition applies to the result of zipping together the before and after versions of the arguments with $\rhd$, plus some return value $r$. The rule conclusion tells us that $\overline{y}$ and $x$ are overwritten appropriately. Note that $\overline{v}$ might contain $\bot$ values, indicating that those arguments are no longer owned by the caller; or it might contain new ADT values for some arguments, indicating mutation. The return value can be either a scalar or an ADT value, with the latter indicating allocation or transfer of ownership.

One last important ingredient is Facade's adaptation of the *safety* predicate from the Cito semantics. A judgment $(\sigma, s) \downarrow$ asserts that statement $s$ will execute safely from state $\sigma$, *along any possible nondeterministic execution path.* The only source of nondeterminism is in rule CallAx, where new variable values and return value are chosen subject to the postcondition. For instance, a finite-set implementation might include a method to choose an arbitrary element of a nonempty set.

The importance of safety is in proving that Facade programs avoid the equivalent of C's undefined behavior. The compiler correctness theorem will only apply to programs that are well-behaved in this way.

Figure 7 shows one example rule of the safety judgment, which, as for Cito, we define *coinductively* (hence the double line), so that certain kinds of infinite derivations are possible, covering programs that may not terminate along some nondeterministic paths.

## 4. Correct-by-Construction Facade Code

The next step on the road to assembly is to compile the Gallina implementations produced by our Fiat refinements into imperative Facade programs. We start by defining an equivalence relation, $\cong$, between the data types produced and manipulated by our high-level set operations and by Facade programs:

$$\mathsf{w} \in \mathsf{W} \cong\ \mathsf{SCA}\ \mathsf{w} \qquad\qquad \mathsf{l} \in \mathsf{List}\ \mathsf{W} \cong\ \mathsf{ADT}(\mathsf{List}\ \mathsf{l})$$
$$\mathsf{i} \in \mathsf{SharpenedADT}\ (\mathsf{FiniteSet}\ \mathsf{W}) \cong\ \mathsf{ADT}\ (\mathsf{Set}\ (\mathsf{s} \approx \mathsf{i}))$$

In summary, Gallina words are directly equivalent to Facade words, while a value $l$ of Gallina's algebraic data type of lists is equivalent to a Facade list ADT modeled by $l$. A value $i$ of a $\mathsf{SharpenedADT}\ (\mathsf{FiniteSet}\ \mathsf{W})$ is equivalent to a Facade finite-set ADT modeled by a set $s$ related to $i$ by the implementation's abstraction relation.

Armed with this notion of equivalence, we can phrase compilation of Gallina terms as another refinement process, living inside of Fiat and producing a Facade statement guaranteed to output a value equivalent to the original Gallina term. Thus, given a term $g$, a precondition $\mathsf{Pre}$ expressing our assumptions on the types of the function arguments, and a postcondition $\mathsf{Post}$ expressing requirements on output, we want to find a Facade statement $p$ satisfying two conditions, where the variables $x_i$ are the function formal parameters:

$$\forall\, \sigma\, x_1 \ldots x_n.\ \mathsf{Pre}\ \sigma\ x_1 \ldots x_n \implies$$
$$(p, \sigma) \downarrow \wedge (\forall\, \sigma'.\ (\sigma, p) \Downarrow \sigma' \implies \mathsf{Post}\ \sigma'\ x_1 \ldots x_n)$$

The exact forms that Pre and Post take depend on the number of parameters of the original specification $S\ x_1 \dots x_n$ in the FiatL DSL:

$$\mathsf{Pre}\ \sigma\ x_1 \dots x_n \equiv \sigma(\mathsf{y}_1) \approx x_1 \wedge \dots \wedge \sigma(\mathsf{y}_n) \approx x_n$$
$$\mathsf{Post}\ \sigma\ x_1 \dots x_n \equiv \sigma(\mathsf{r}) \approx Z\ x_1 \dots x_n$$

where $Z$ stands for an executable Gallina function. In practice, this value is an implementation of $S$ that has been synthesized by the first part of the pipeline.

Compilation proceeds by iteratively decomposing the Gallina term produced by the Fiat compiler, until parts become simple enough to map directly to Facade values. Phrasing compilation in terms of refinements allows compilation to reuse all of the existing Fiat machinery to implement each compilation step as an application of a refinement lemma, picked from a database of verified compilation rules. Each refinement lemma operates on a nondeterministic choice (or *pick*) in a particular form $\sigma \rightsquigarrow \sigma'$. This notation designates all programs that:

- Are safe when started in states containing exactly the same ADTs as $\sigma$ and all the scalars in $\sigma$.
- For any such starting state, produce a final state containing the same ADTs as $\sigma'$ and all the scalars in $\sigma'$.

Writing $\varnothing[\mathsf{x} \rightarrow \mathsf{a}]$ to denote an empty set of bindings augmented with a single binding $\mathsf{x} \rightarrow \mathsf{a}$, consider deriving a compilation for a function that computes the absolute value of a number $a$, specified using this notation as follows:

$$\varnothing[\mathsf{x} \rightarrow \mathsf{a}] \rightsquigarrow \varnothing[\mathsf{r} \leftarrow \mathbf{if}\ \mathsf{a} > 0\ \mathbf{then}\ \mathsf{a}\ \mathbf{else}\ -\mathsf{a}]$$

Compilation can be reduced (with a generic if rule) to producing three smaller programs: one for the condition $\mathsf{a} > 0$, one for the true part $\mathsf{a}$, and one for the false part $-\mathsf{a}$:

$$\begin{aligned}
&\mathsf{ptest} \leftarrow (\varnothing[\mathsf{x} \rightarrow \mathsf{a}] \rightsquigarrow \varnothing[\mathsf{x} \rightarrow \mathsf{a},\ \mathsf{y} \rightarrow (\mathsf{a} > 0)]);\\
&\mathsf{ptrue} \leftarrow (\mathsf{a} > 0 \rightarrow \varnothing[\mathsf{x} \rightarrow \mathsf{a},\ \mathsf{y} \rightarrow 1] \rightsquigarrow \varnothing[\mathsf{r} \rightarrow \mathsf{a}]);\\
&\mathsf{pfalse} \leftarrow (\mathsf{a} \leq 0 \rightarrow \varnothing[\mathsf{x} \rightarrow \mathsf{a},\ \mathsf{y} \rightarrow 0] \rightsquigarrow \varnothing[\mathsf{r} \rightarrow -\mathsf{a}]);\\
&\mathbf{ret}\ (\mathsf{ptest};\ \mathbf{if}\ \mathsf{cond} = 0\ \mathbf{then}\ \mathsf{ptrue}\ \mathbf{else}\ \mathsf{pfalse})
\end{aligned}$$

Compiling each fragment further yields a final program

$$\mathbf{ret}\ (\mathsf{y} := \mathsf{x} > 0;\ \mathbf{if}\ \mathsf{y} = 0\ \mathbf{then}\ \mathsf{r} := \mathsf{x}\ \mathbf{else}\ \mathsf{r} := 0 - \mathsf{x})$$

This particular form of pre- and postconditions on the program fragments is designed to make it possible to guarantee that the resulting programs satisfy Facade's sanity requirements, and in particular the requirements preventing one from overwriting or aliasing variables pointing to ADTs. Thus, unlike scalars, uses of ADTs must be tracked carefully throughout program execution, marking the transition from the purely functional world of Gallina programs to the mutable one of Facade programs. Our choice of pre- and postconditions leverages this distinction to put less stringent requirements on how information about scalars flows through the program, so that program fragments can easily discard information about the scalar variables that appear in their preconditions.

When a program is compiled into smaller fragments, it is not always apparent which ADTs the eventual implementation of each fragment will use, as compilation might produce an implementation that allocates new ADTs, or modifies or removes existing ones. It is thus necessary to keep "holes" in the pre- and postconditions of consecutive program fragments – placeholders that connect the set of live ADTs and variables at the end of the fragment to the ones at the beginning of the next fragment. These placeholders are filled by unifying them with the particular pre- and postcondition shapes imposed by refinement lemmas, and as such can also serve to ease compilation and lend more generality to existing compilation refinements. As an example, consider the following specification of a program that reads the head of a list and discards the tail: $\varnothing[\mathsf{x} \rightarrow \mathsf{List}\ (\mathsf{hd} :: \mathsf{tl})] \rightsquigarrow \varnothing[\mathsf{y} \rightarrow \mathsf{hd}]$

This specification requires a Facade implementation to both copy the head of the list to the $\mathsf{y}$ variable and deallocate $\mathsf{x}$ before exiting. This operation is not one of the those supported directly by the list ADT, but it can be broken down into two smaller operations, by inserting a placeholder via a very general refinement rule for breaking a Facade computation into two steps:

$$\begin{aligned}
&\mathsf{p} \leftarrow \varnothing[\mathsf{x} \rightarrow \mathsf{List}\ (\mathsf{hd} :: \mathsf{tl})] \rightsquigarrow \boxed{1};\\
&\mathsf{q} \leftarrow \boxed{1} \rightsquigarrow \varnothing[\mathsf{y} \rightarrow \mathsf{hd}];\\
&\mathbf{ret}\ (\mathsf{p};\ \mathsf{q})
\end{aligned}$$

At this point, searching a database of refinement lemmas matching the form of the first fragment will yield a lemma describing the semantics of the Pop() method of the linked-list ADT, which might look like

$$\begin{aligned}
&\forall\ \mathsf{hd}\ \mathsf{tl}\ \mathsf{x}\ \mathsf{y}.\\
&\quad (\sigma[\mathsf{x} \rightarrow \mathsf{List}\ (\mathsf{hd} :: \mathsf{tl})] \rightsquigarrow \sigma[\mathsf{y} \rightarrow \mathsf{hd},\ \mathsf{x} \rightarrow \mathsf{List}\ \mathsf{tl}])\\
&\quad \supseteq \mathbf{ret}\ (\mathsf{y} := \mathsf{Pop}(\mathsf{x}))
\end{aligned}$$

Applying this refinement rule implicitly fills the placeholder $\boxed{1}$ with the term $\varnothing[\mathsf{x} \rightarrow \mathsf{hd},\ \mathsf{y} \rightarrow \mathsf{List}\ \mathsf{tl}]$; this change propagates to $\mathsf{q}$, making it easy then to refine the second fragment into a call to the Delete() method of the list ADT. Furthermore, if this type of program fragment shows up often in a particular domain, it is possible to add a specialized lemma to recognize it and speed up compilation.

Refinement lemmas can be classified broadly into four categories:

- low-level refinements that deal with refining language constructs such as conditionals, copying values between scalar variables, and introducing holes;
- refinements that handle calls to ADT methods;
- high-level refinements that compile complex language constructs such as folds; and
- user-specified lemmas, which combine and extend other refinements to achieve particular optimizations.

The third category, high-level compilation lemmas, yields particularly interesting refinement examples. Our library supports compiling folds, producing either words or ADTs, into Facade's while loops. As an example of user extension, we chose to compile our folds producing pairs into loops modifying two variables, to avoid the runtime expense of manipulating boxed pair objects. The compilation lemma for folds producing ADTs looks like:

```
$\forall$ x r l acc Adt thead tis_empty.
($\sigma$[x → List l] $\rightsquigarrow$ $\sigma$[r → Adt (fold_left loop l acc)])
$\supseteq$init ← ($\sigma$[x → List l] $\rightsquigarrow$ $\sigma$[x → List l, r → Adt acc]);
  body ← ($\forall$ acc hd tl. $\sigma$[r → Adt acc, thead → hd, x → tl]
                           $\rightsquigarrow$ $\sigma$[r → Adt (loop acc hd), x → tl]);
  ret (init;
       tis_empty := x.IsEmpty();
       while (!tis_empty) {
         thead := x.Pop();
         body;
         tis_empty := x.IsEmpty()};
       x.Delete())
```

Adt designates an arbitrary ADT constructor; the first pick produces a program that stores the starting accumulator of the fold in r; the second pick produces a program that updates the accumulator of the loop; the ∀ acc hd tl part indicates that this pick is in fact a pick of a program that is safe and computes the right values for any choices of acc, hd, and tl. After applying this rule, the subprograms init and body can be refined further, until arriving at a ret of a concrete Facade program.

***Compiling Fiat ADT Calls*** The Fiat and Facade specifications of the finite-set ADTs (shown in Figures 3 and 5) closely mirror each other. Thus, the refinement proof in SharpenedADT FiniteSetSpec allows a direct compilation of Fiat method calls into corresponding Facade calls. This refinement proof only holds for method calls on a concrete implementation state related via the abstraction relation to a particular set. Since Coq lacks an abstraction theorem that would allow us to use parametricity to derive this result for free, we have to add it as an assumption of the compilation lemma, as demonstrated in:

```
$\forall\ \Psi\ x_s\ x_w\ \sigma\ x\ \ l\ \ s\ \ r\ \ w,$
 $s \approx r \wedge \Psi(l) = \text{AddSpec} \wedge \text{NoDup}([x_s, x_w, x]) \rightarrow$
 $(\sigma[x_s \rightarrow s,\ x_w \rightarrow w]\ \rightsquigarrow \sigma[(x, x_s) \rightarrow r.\text{Add}(w)])$
$\supseteq$ ret $(x := \text{call}\ l(x_s, x_w))$
```

We use the notation $(x, y) \rightarrow e$ to reflect updating variables x and y with the two components of pair e. The respective hypotheses of the lemma assert that set s in the prestate can be implemented by some internal representation state r of a finite-set ADT implementation, that some function label l is present in the context Ψ and is mapped to the predefined method spec AddSpec, and that the three schematic Facade variables are all distinct from each other. Our domain-specific refinement tactics discharge those side conditions automatically.

***Automation*** Much like the transition from mathematical specifications to computational programs in Fiat, the compilation from Fiat to Facade is meant to be fully automated.

We begin by constructing the nondeterministic choice of a safe Facade program computing a value corresponding to the mathematical specification provided by the user. We then repeatedly decompose and refine that choice into smaller program fragments, by applying a multitude of small decomposition lemmas, akin to the rule for compiling conditionals presented above. The decomposition process goes on until we reach fragments whose specifications can be directly related to concrete Facade statements. If we ever encounter a situation in which no compilation lemma applies, and which does not seem further decomposable, we split the problematic fragment into multiple consecutive fragments, replacing ADTs and scalars with placeholders (for practical reasons, our compiler stores known ADTs and scalars in two separate maps), and try to refine these newly create fragments. The definition of refinement guarantees that our compilation does not compromise the safety or correctness of the program, as the refinement process itself produces a proof trace relating the final program to the original specification.

## 5. From Facade to Assembly

Having built a Facade program, we can now realize our goal of generating verified assembly code. The basic process is to compile Facade to Cito, which we then compile to Bedrock using a compiler verified in prior work [Wang et al. 2014].

The biggest difference between Facade and Cito is their notion of machine states. As mentioned above, Facade's machine state is a partial mapping from variables to values (either SCA or ADT), while Cito's machine state consists of two mappings, one (a total mapping) from variables to integers (possibly pointers), the other (a partial mapping) from pointers to ADT values. In order to define the correctness of the Facade-to-Cito compiler, firstly we need to define a relation between these two machine states, to formalize the notion that "a Cito state faithfully implements a Facade state." A relation $\sigma \approx (\delta, \mu)$ between Facade state $\sigma$ and Cito state $(\delta, \mu)$ is defined as

$$(\forall x, a.\ \sigma(x) = \mathsf{ADT}(a) \Rightarrow \exists w.\ \delta(x) = w \wedge \mu(w) = a) \wedge$$
$$(\forall x, w.\ \sigma(x) = \mathsf{SCA}(w) \Rightarrow \delta(x) = w) \wedge$$
$$(\forall w, a.\ \mu(w) = a \Rightarrow \exists! x.\ \sigma(x) = \mathsf{ADT}(a) \wedge \delta(x) = w)$$

This relation conveys three facts: (1) mappings contained in $\sigma$ are all covered by $(\delta, \mu)$; (2) mappings contained in $\mu$ are all covered by $\sigma$; (3) each ADT value in $\mu$ is referenced by a unique variable in $\sigma$.

Fact (3) may appear deceptively similar to a requirement that all allocated ADTs are reachable from the local variables of the current dynamic function call, but recall that Facade employs a small-footprint style of semantics, where unused state elements need not be mentioned explicitly in the semantics.

When a Cito state implements a Facade state according to the relation we have sketched, fact (2) guarantees that there is no memory leak (memory that is unreachable from the Facade state), and fact (3) guarantees that no two ADT values in the Facade state will accidentally be implemented by the same object in Cito's heap (aliasing).

The correctness theorem of the Facade-to-Cito compiler is a standard semantics-preservation theorem:

$$\forall \Sigma, \Sigma', \sigma.\ (\Sigma, t) \Downarrow_C \Sigma' \wedge \sigma \approx \Sigma \wedge (\sigma, s) \downarrow_F \\ \Rightarrow \exists \sigma'.\ (\sigma, s) \Downarrow_F \sigma' \wedge \sigma' \approx \Sigma'.$$

Let $s$ be the Facade source program and $t$ be the Cito target program generated by the compiler. We use subscripts "F" and "C" to distinguish between the Facade and Cito versions of the "runs-to" and "safe" judgments. This theorem says that starting from equivalent states, whenever the target program may terminate in a state, there is a matching source-program execution, finishing in a related state. In other words, this theorem states a refinement of the source program by the target program.

After generating Cito programs, we rely on the preexisting Cito-to-Bedrock compiler [Wang et al. 2014] to do all the remaining translation. The attractive *vertical* and *horizontal* compositionality properties of that compiler proof mean that we can combine the Facade-to-Cito correctness theorem and the Cito-to-Bedrock correctness theorem to get a Facade-to-Bedrock correctness theorem, and the generated Bedrock module can be linked with other Bedrock modules, fulfilling each other's imports.

The combined Facade-to-Bedrock compiler has type

$$\forall P, Q.\ \mathsf{FModule}(P, Q) \to \mathsf{BModule}(P, Q),$$

The compiler is parameterized on a precondition $P$ and postcondition $Q$. $\mathsf{FModule}(P, Q)$ is the compilation unit fed to the compiler, which is a dependent record containing a Facade program $s$ and two proofs:

- When $P$ holds, this program is safe to run: $\forall \sigma.\ P(\sigma) \Rightarrow (\sigma, s) \downarrow$
- When this program finishes, the final state satisfies $Q$: $\forall \sigma, \sigma'.\ (\sigma, s) \Downarrow \sigma' \wedge P(\sigma) \Rightarrow Q(\sigma, \sigma')$

The compiler's output, $\mathsf{BModule}(P, Q)$, is a Bedrock verified module with $P$ and $Q$ as its specification (written in Bedrock's specification language, as in Figure 2d). From this type signature, we may deduce that for any property $(P, Q)$, this compiler will faithfully translate a Facade program satisfying $(P, Q)$ to a Bedrock program that satisfies $(P, Q)$.

## 6. Putting It All Together

The last steps of our development process follow past work on Bedrock: we implemented the finite-set and list ADTs in Bedrock's C-like language with invariant annotations [Chlipala 2013] and proved the implementations correct using the separation-logic proof automation [Malecha et al. 2014]. Now we may finish the derivation process for any FiatL spec, taking the Bedrock module produced at the end of the last section's step and *linking* it with our ADT implementations

| **Description** | **FiatL specification** |
|---|---|
| Remove duplicates | $\lambda\, \ell.\ \upharpoonleft \downharpoonright \ell \downharpoonleft \upharpoonright$ |
| Sum of all | $\lambda\, \ell.\ \mathsf{fold\_right}\ (+)\ \ell\ 0$ |
| Sum of all (unique) | $\lambda\, \ell.\ \mathsf{Fold}\ (+)\ \downharpoonright \ell \downharpoonleft\ 0$ |
| Filter < | $\lambda\, \ell, x.\ \mathsf{filter}\ (\lambda\, y.\ y < x)\ \ell$ |
| Filter < (unique) | $\lambda\, \ell, x.\ \upharpoonleft \downharpoonright \ell \downharpoonleft -\ \{y \mid y < x\} \upharpoonright$ |
| Union (unique) | $\lambda\, \ell_1, \ell_2.\ \upharpoonleft \downharpoonright \ell_1 \downharpoonleft\ \cup\ \downharpoonright \ell_2 \downharpoonleft \upharpoonright$ |

**Figure 8.** Some of the examples we handle automatically

(compiled to assembly by Bedrock). This linking process is not just syntactic but also semantic, checking for agreement between the import and export specifications associated to code labels by different modules. For instance, the final Bedrock module for a FiatL program will import the finite-set methods with particular specifications, while our manually verified finite-set module exports the same methods, and linking checks that the specs agree syntactically across those two positions.

The result of linking is a *closed assembly program*, with no more imports, to which applies the main functional-correctness theorem of the Bedrock framework, based on final specs like the one in Figure 2d. We may also run closed programs directly.

Figure 8 summarizes the FiatL specifications that we have experimented with. For each one, we derived a correct-by-construction assembly program, automatically. Our implementation produces efficient, correct implementations of both the list-based and set-based operations, returning either machine words or linked lists of machine words. All examples are compiled into while loops and never resort to recursion. The "Sum of all (unique)" test case inspired the running example of Figure 2. In total, the formal compilation process per example takes about 1 second to refine the high-level specification, 1 minute to refine to Facade, and an extra 5 minutes to check the linking conditions.

Running our examples on real hardware is also straightforward, if a bit inconvenient because of inefficiencies in Coq's normalization of terms as large as ours. We wind up with one Coq term standing for a closed Bedrock module, and we normalize it to find the literal assembly code within, in the process running all of our suite of compilers, linkers, etc., inside Coq. The result is pretty-printed as a `.s` file and compiled with the standard GNU toolchain.

As a concrete example, we compiled our running "Sum of all (unique)" function example (to about 2500 lines of x86 assembly) and timed its execution on random input lists of different lengths. The final code is somewhat inefficient, since we currently use an unsorted-list representation of finite sets; in the future, a more efficient verified implementation can be dropped into our framework without changing any other modules. Still, the extracted assembly code is reasonably efficient on moderately sized inputs, taking (on an Intel Xeon E5620 CPU at 2.4 GHz) about 10 ms to handle a length-1000 list, 200 ms for length 5000, and 600 ms for length 10,000.

## 7. Related Work

***Deductive Synthesis*** Specware [Specware] and its predecessors KIDS [Smith 1990] and DTRE [Blaine and Goldberg 1991] are deductive synthesis tools for deriving correct-by-construction implementations of high-level specifications, akin to Fiat. At each step, Specware checks the validity of the refinement by generating Isabelle/HOL proof obligations justifying each transformation. The result of these proved-correct refinements is a program in MetaSlang, a functional, ML-like language. Through a series of automated and quite sophisticated transformations, these MetaSlang programs are transformed into C code. In contrast to the automated compilation from Gallina to Facade presented here, however, these final steps are currently unverified, and these transformations represent a key piece of Specware's trusted code base, in addition to whatever compiler is applied to the final C code.

***More Automated Synthesis*** In contrast to the algorithmically complex software that Kestrel has synthesized including families of garbage collectors [Pavlovic et al. 2010], SAT solvers [Smith and Westfold 2008], and network protocols, we envision the toolchain presented here being used to implement specifications for algorithmically "simple" domains that are amenable to automation. A number of domains have been shown to have this property: Paige et al. [Paige and Henglein 1987] demonstrate how efficient implementations specified in terms of set operations can be synthesized [Paige and Henglein 1987], for example. Computations specified using query-like operations are another such domain. P2 [Batory et al. 1993] was a DSL extension to C that allowed users to specify the container data structures implementing an iterator method for querying contents of the container. More recently, Hawkins et al. [Hawkins et al. 2011] have shown how to synthesize the implementations of abstract data types specified by abstract relational descriptions supporting query and update operations.

***Program Extraction from Proof Assistants*** The program-extraction mechanisms of Coq [Letouzey 2003] and other proof assistants are used widely, such as in CompCert and in the Ynot framework [Nanevski et al. 2008] for verifying Haskell-style monadic Gallina programs and the Reflex system [Ricketts et al. 2014] built on top of it for automated verification of reactive programs. Our new transformation pipeline with the first step chopped off is a starting point for providing a similarly general kind of extraction, with stronger formal guarantees and allowing the programmer more flexibility to control which optimizations are applied.

***Compiler Verification*** The best-known mechanically verified compiler is CompCert [Leroy 2006]. Its main theorem requires that code only be linked with modules compiled by the same version of CompCert, disallowing the sort of cross-language linking that allows us to link Facade code with Bedrock ADT implementations. Recent work [Stewart et al. 2015] generalized the theorem to support separate and cross-language compilation, though it has not yet been used to do functional verification of an assembly program with pieces compiled by different compilers. Variants of this twist on compiler correctness have also motivated other recent projects without mechanized proofs [Benton and Hur 2009; Ahmed and Blume 2008, 2011].

A sometimes lighter-weight alternative to compiler verification is translation validation, where only particular compiler outputs are proved to preserve source-program behavior, similarly to how our program derivations only prove theorems about particular translations. For instance, one recent project [Sewell et al. 2013] applied the approach in the context of OS kernel verification. Translation validation may often work with out-of-the-box compilers, instead of new ones built with formal verification in mind, though sophisticated optimizations frequently require special compiler support.

***Extensible Compilers*** Of the approaches that have been suggested for making compilers extensible, the most relevant to our work is XCert [Tatlock and Lerner 2010], which extends CompCert with a verified execution engine for a domain-specific language of program analyses and transformations, enforcing soundness by construction. XCert more comprehensively addresses this sort of traditional C-compiler optimization (e.g., based on dataflow analysis), compared to the optimizations built into the Cito compiler [Wang et al. 2014], and those transformations are complementary to the ones we perform from high-level specs and functional programs to imperative programs. Other extensible compilers like xtc [Grimm 2006] and xoc [Cox et al. 2008] provide no (formal or informal) soundness guarantees. The Lisp world has a long and related tradition of extension via macros, generally without any connection to formal program-correctness proofs, with the state of the art probably exemplified by Racket's extension API [Tobin-Hochstadt et al. 2011].

## 8. Conclusion

We have presented the first certified pipeline, implemented within Coq, from specifications to assembly code that constructs machine-checked proofs that assembly functions implement relational specifications. The stages of our pipeline are extensible; programmers may add new compilation strategies as refinement lemmas and tactics that apply them intelligently. Nonetheless, such programmer extensions cannot compromise the soundness of the system, thanks to pervasive proof generation. Our case studies so far start from a modest spec domain of expressions over lists and sets, and we plan to explore scaling the approach up to more complex specs, like the SQL-style queries from past work on Fiat [Delaware et al. 2015].

## References

A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proc. ICFP*, pages 157–168. ACM, 2008.

A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proc. ICFP*, pages 431–444. ACM, 2011.

D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 1993.

N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, pages 97–108. ACM, 2009.

L. Blaine and A. Goldberg. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, pages 165–204. Elsevier, 1991.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. PLDI*, pages 234–245. ACM, 2011.

A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*, pages 391–402. ACM, 2013.

R. Cox, T. Bergan, A. T. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. ASPLOS*, pages 244–254. ACM, 2008.

D. Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, pages 85–95. Springer-Verlag, 2000.

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

R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51. ACM, 2006.

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

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

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

X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. ACM, 2006.

P. Letouzey. A new extraction for Coq. In *Proc. TYPES*. Springer-Verlag, 2003.

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

G. Malecha, A. Chlipala, and T. Braibant. Compositional computational reflection. In *Proc. ITP*, pages 374–389. Springer-Verlag, 2014.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *Proc. ICFP*, pages 229–240. ACM, 2008.

R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code – a case study. *J. Symb. Comput.*, 4(2):207–232, Oct. 1987. ISSN 0747-7171. doi: `10.1016/S0747-7171(87)80066-4`. URL `http://dx.doi.org/10.1016/S0747-7171(87)80066-4`.

M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. POPL*, pages 247–258. ACM, 2005.

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

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.

D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. In *Proc. PLDI*, pages 452–462. ACM, 2014.

T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proc. PLDI*, pages 471–482. ACM, 2013.

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

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

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

G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proc. POPL*. ACM, 2015.

Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. PLDI*, pages 111–121. ACM, 2010.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proc. PLDI*, pages 132–141. ACM, 2011.

P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proc. OOPSLA*, pages 675–690. ACM, 2014.