Raising the Level of Abstraction in Systems Programming with Fiat and Extensible, Correct-by-Construction Compilers

Adam Chlipala
MIT CSAIL
ENTROPY workshop
January 2018

Joint work with: Thomas Braibant, Santiago Cuellar, Benjamin Delaware, Samuel Duchovni, Jason Gross, Gregory Malecha, Clément Pit—Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye
How We're Doing

Software bug causes launch failure

Software bug leaks secret information

Hardware bug causes massive recall

Software bug causes loss of life

The time has come to settle for nothing less than high-assurance computer systems!
It is time for you to take your medicine.

Oh, sure. Sure, sure, sure.

Better get back to work....
Formal specifications and proofs deserve to be *the new glue* holding together complex systems and helping us understand them and their parts.

The design of systems should *change* to take advantage of formal methods to *raise the level of abstraction*.
The Big Idea

Language implementation should enforce that algorithms can't break data structure invariants.

Only need to read this part of the code, to understand non-performance aspects! Language implementation should enforce that optimizations can't break correctness.
That Looks Familiar

What do these pictures have in common?
Mere mortals fear to tread here:
State of the Art: Building an Internet Server

"The Cloud"

SQL Database

Persistent State

Cryptography

Packet Format Parsing

Core Protocol Logic

Library Reuse

Parser Generator

rockstar coders

source

rockstar coders

source
Complaints About: Talking to a Standard Server

- **SQL Database**: Persistent State
  - Database is a black box, maintained by an elite cadre, often not doing quite what you need.
  - And by the way, sometimes there are serious bugs.
  - Yet another language, only understandable after reading a pile of documentation.

- **Awkward API**: Often based on string manipulation, allowing code-injection vulnerabilities.

And by the way, sometimes there are serious bugs.
Complaints About: Using a Domain-Specific Language

And by the way, sometimes there are serious bugs.

Compiler is a black box, maintained by an elite cadre, often not doing quite what you need.

Yet another language, only understandable after reading a pile of documentation.

Awkward integration, with build processes instead of clean intra-language abstractions.
### What About *Embedded* DSLs?

<table>
<thead>
<tr>
<th>Complaint</th>
<th>Addressed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yet another language</td>
<td><em>Partly yes,</em> but still need to learn the semantics of the DSL, even if syntax may be standardized</td>
</tr>
<tr>
<td>Compiler is a black box</td>
<td>No!</td>
</tr>
<tr>
<td>Awkward integration</td>
<td>Yes!</td>
</tr>
<tr>
<td>Sometimes serious bugs</td>
<td><em>Partly yes,</em> as we usually avoid type-safety bugs but not deeper semantic bugs</td>
</tr>
</tbody>
</table>
Complaints About: Using Libraries Coded by Wizards

- Algorithms
- Prime #s
- HW Arches

Labor-intensive adaptation, with each combination taking at least several days for an expert.

Library Reuse

Cryptography

rockstar coders

And by the way, sometimes there are serious bugs.
Rethinking the Programming Framework

Correctness bugs? Ruled out by pervasive use of a proof assistant.

- Domain-Specific Notation
- Semantics

proof

proof

proof
Macros desugar into the common language of higher-order logic. Often the most concise code isn't obviously executable!

Functionality

Performance

Optimization scripts use Coq's tactic language and are correct by construction.
Fiat's Layers

1. **Coq**: logic and tactic language
2. **Computations**: nondeterministic functional programs
3. **Abstract data types**: encapsulated state
4. **Domains**: libraries for particular spec styles
5. **Applications**
Definition SchedulerSchema :=
Query Structure Schema [relation "Processes" has schema
   "pid" :: W, "state" :: State, "cpu" :: W
   where (UniqueAttribute "pid")
] enforcing [].

Definition SchedulerSpec : ADT _ :=
QueryADTRep SchedulerSchema |
(* ... *)
Def Method1 "Enumerate" (r : rep) (state : State) : rep * list W :=
For (p in r!"Processes")
  Where (p!"state" = state)
  Return (p!"pid"),
(* ... *)
].
Demo:
Query Structures
& the Bookstore example
Core Fiat: **Computations** as Sets of Results

- **Initial Spec** (nondeterministic, in logic)
- **Refined Spec** (some decisions made)
- **Refinement (subset) relation, with Coq proofs**
- **Awkward API?**
  Uniform notion of computations with common logic
- **Functional**
  - Verified compilation to assembly
- **Imperative**

One allowable behavior, within a space of choices
Computations naturally form a \textbf{monad}.

\[
\text{ret } v \overset{\text{def}}{=} \{ v \}
\]

\[
x c_1 ; c_2(x) \overset{\text{def}}{=} \{ v \in c_2(x) \mid x \in c_1 \}
\]

Example:
\[
x \{ n \in \mathbb{N} \mid \exists m. n = 2 \times m \};
\]
\[
y \mathbb{N};
\]
\[\text{ret } (x + 2 \times y)\]

In other words, choose an even number, by some very indirect means!
Refinement of computations is just subset.

In other words, resolving nondeterminism is the same as moving to a smaller set.

Example:

\[
\begin{align*}
    x & \in \{ n \in \mathbb{N} \mid \exists m. \ n = 2 \times m \}; \\
    y & \in \mathbb{N}; \\
    \text{ret} \ (x + 2 \times y)
\end{align*}
\]

\[\subseteq \{ n \in \mathbb{N} \mid \text{even}(n) \}\]

\[
\begin{array}{c}
\text{ret 42}
\end{array}
\]
Refinement is compatible with rewriting.

Proved lemma:
\[ \forall v. f(v) \supseteq g(v) \]

For \( v = e_1 \), specializes to
\[ c_y \supseteq d_y \]

Rewrite

```
a c_a;
b c_b;
...
y c_y;
z c_z;
ret e
```
Demo:
find an element not in a list
Fiat Principle #2: **Abstract Data Types** with computations for methods

Type of *private state*

ADT {
    rep = T
    method \(m_1(x : D_1) : R_1 = c_1\)
    method \(m_2(x : D_2) : R_2 = c_2\)
    ...
    method \(m_n(x : D_n) : R_n = c_n\)
}

Method has computation as body, allowing **nondeterminism**.
Macros as Documentation

Def Method1 "NumOrders" (r : rep) (author : string) : rep * nat :=
count <- Count (For (o in r!sORDERS) (b in r!sBOOKS)
  Where (author = b!sAUTHOR)
  Return ()); ret (r, count)

Yet another language?
Readable macros
desugaring into a common
language

Count b ≡
results ← b;
return length(results)

Where P b ≡
{l | P → l B b ¬P → l = []}
Ingredients for Optimization Scripts

\[
\text{filter } (\lambda(x, _){ f(x)} ) : (\text{join } l_1 l_2) = \text{join} \left(\text{filter } f \ l_1\right) l_2
\]

\[
\text{filter } (\lambda(k, v) \rightarrow k = k_0) \ (\text{BST.enumerate } t) = \text{BST.lookup } t \ k_0
\]

Compiler a black box?
Easy to add new optimization rules w/ proofs
Example Derivation: SQL-style database

table island : {Name : string, Size : int, Temp : int}

sizeOf(db, name) =
    for i ∈ db.island
    where i.Name = name
    return i.Size
STEP 1: representation change

dictionary island : string
Abstraction relation:
  \[ \text{db} \sim \text{fmap} \overset{\text{def}}{=} \forall r. \ r \in \text{db} \]
  \[ \text{fmap}(r.\text{Name}) = \{\text{Size} = r.\text{Size}, \text{Temp} = r.\text{Temp}\} \]

\[ \text{sizeOf}(\text{fmap}, \text{name}) = \]
  \[ \text{db} \quad \{\text{db} \mid \text{db} \sim \text{fmap}\} \]
  \[ \text{for } i \in \text{db.island} \]
  \[ \text{where } i.\text{Name} = \text{name} \]
  \[ \text{return } i.\text{Size} \]

RULE: representation change

state \( x : \tau^x \)
  \[ m(x, \text{args}) = e \]

state \( y : \tau^y \)
  \[ m(y, \text{args}) = \]
  \[ x \quad \{x \mid x \sim y\} \]
  \[ e \]
STEP 2: for+fmap

dictionary island : string {Size : int, Temp : int}

sizeOf(fmap, name) =
    for k, v ∈ fmap
    let i = {Name = k, Size = v.Size, Temp = v.Temp}
    where i.Name = name
    return i.Size

RULE: use dictionary

x {x | ∀r. r ∈ x \ y(k(r)) = v(r)}
for r ∈ x
q(r)

for k, v ∈ y
let r = kv⁻¹(k, v)
q(r)
STEP 3: simplify

dictionary island : string   {Size : int, Temp : int}

sizeOf(fmap, name) =
  for k, v ∈ fmap
  where k = name
  return v.Size

sizeOf(fmap, name) =
  for k, v ∈ fmap
  let i = {Name = k, Size = v.Size, Temp = v.Temp}
  where i.Name = name
  return i.Size
dictionary island : string   {Size : int, Temp : int}

sizeOf(fmap, name) =
    for v ∈ fmap.lookup(name)
    return v.Size

RULE: equality test to lookup

for k, v ∈ fmap
    where k = x
    q(k, v)

for v ∈ fmap.lookup(x)
    q(x, v)
Example Derivation: binary decoder

$T = \{A : \text{int}, B : \text{string}, C : \text{list int}\}$

$\text{encode}(t : T) = \text{encodeInt}(t.A)
++ \text{encodeInt}(\text{len}(t.C))
++ \text{encodeString}(t.B)
++ \text{encodeList}(\text{encodeInt}, t.C)$

$\text{decode}(s : \text{bitstring}) =
\{t \mid s = \text{encode}(t)\}$
Example Derivation: binary decoder

\[
T = \{A : \text{int}, B : \text{string}, C : \text{list int}\}
\]

\[
\text{encode}(t : T) = \text{encodeInt}(t.A) \quad ++ \quad \text{encodeInt}(\text{len}(t.C)) \\
\quad ++ \quad \text{encodeString}(t.B) \\
\quad ++ \quad \text{encodeList}((\text{encodeInt}, t.C))
\]

\[
\text{decode}(s : \text{bitstring}) = \\
\{ t \mid s = \text{el}(t.A) \quad ++ \quad \text{el}(\text{len}(t.C)) \quad ++ \quad \text{eS}(t.B) \quad ++ \quad \text{eL}((\text{el}, t.C))\}
\]
Example Derivation: binary decoder

decode(s : bitstring) =
{t | s = el(t.A) ++ el(len(t.C)) ++ eS(t.B) ++ eL(el, t.C)}
STEP 1: decode A

\[
\text{decode}(s : \text{bitstring}) = \\
\quad \text{let } a, s = \text{dl}(s) \\
\quad \{ t \mid t.A = a \land s = eI(\text{len}(t.C)) ++ eS(t.B) ++ eL(eI, t.C) \}
\]

**RULE:** *decode integer*

\[
\{ x \mid P(x) \land s = eI(f(x)) ++ s' \} \\
\text{let } v, s = \text{dl}(s) \\
\{ x \mid f(x) = v \land P(x) \land s = s' \}
\]
STEP 2: decode length

decode(s : bitstring) =
  let a, s = dl(s)
  let n, s = dl(s)
  \{ t | \text{len}(t.C) = n \land t.A = a \land s = eS(t.B) ++ eL(eI, t.C) \}

let a, s = dl(s)
\{ t | t.A = a \land s = eI(\text{len}(t.C)) ++ eS(t.B) ++ eL(eI, t.C) \}
STEP 3: decode B

decode(s : bitstring) =
   let a, s = dl(s)
   let n, s = dl(s)
   let b, s = dS(s)
   \{ t | t.B = b \land \text{len}(t.C) = n \land t.A = a \land s = eL(eI, t.C) \}

RULE: decode string
\{ x | P(x) \land s = eS(f(x)) ++ s' \}

let v, s = dS(s)
\{ x | f(x) = v \land P(x) \land s = s' \}
STEP 4: decode C

decode(s : bitstring) =
  let a, s = dI(s)
  let n, s = dI(s)
  let b, s = dS(s)
  let c, s = dL(dI, s, n)
  \{t | t.C = c \land t.B = b \land \text{len}(t.C) = n \land t.A = a \land s = []\}

**RULE: decode list**

\{x | P(x) \land s = eL(eI, f(x)) ++ s'\}
let v, s = dL(eI, s, n)
\{x | f(x) = v \land P(x) \land s = s'\}
when \forall x. P(x) \land \text{len}(f(x)) = n
decode(s : bitstring) =
   let a, s = dl(s)
   let n, s = dl(s)
   let b, s = dS(s)
   let c, s = dL(dl, s, n)
   if s = []:
      {A = a, B = b, C = c}
   else:
      fail

RULE: use
   witness
   \{ x \mid P(x) \land s = []\}
   if s = []:
      v
   else:
      fail

when P(v)
A Relational Abstract Data Type

ADT FiniteSet(α) {
    private set : ℘(α);

    constructor init() {
        set := {};
    }

    method add(x : α) {
        set := {x} ∪ set;
    }

    method member(x : α) {
        return {b : bool | b = true ↔ x ∈ set};
    }

    method toList() {
        return {l : list(α) | NoDup(l) ∧ ∀x. x ∈ set ↔ In(x, l)};
    }
}
ADT Delegation

ADT MyBookCollection(FS : FiniteSet(nat × string)) {
    private books : FS;

    constructor init() {
        books := new FS();
    }

    method newBook(isbn : nat, title : string) {
        books.add((isbn, title));
    }

    method allTitles() {
        return map (fn (_, title) => title) (books.toList());
    }
}
Translating to Lower-Level Imperative Code

ADT MyBookCollection(FS : FiniteSet(nat × string)) {
    private books : FS;

    method newBook(isbn : nat, title : string) {
        tup := new Tuple();
        tup.set(0, isbn);
        tup.set(1, title);
        books.add(tup);  }

    method allTitles() {
        ls := books.toList();
        out := new List();
        while (!ls.isEmpty()) {
            x := ls.pop();
            title := tup.get(x, 1);
            out.push(title);
        }
        delete ls;
        out.reverse();
        return out; } }
Verif i ed Compilation with ADTs

method allTitles() {
    ls := books.toList();
    out := new List();
    while (!ls.isEmpty()) {
        x := ls.pop();
        title := tup.get(x, 1);
        out.push(title);
    }
    delete ls;
    out.reverse();
    return out;
}

Two key parameters to operational semantics of imperative language:
1. Domain of abstract models for foreign data types
2. Hoare-style precondition and postcondition for every foreign function, using abstract predicates for foreign data types

Verif i ed compiler justi fies linking with arbitrary implementations of those types and operations in other Bedrock languages.
Ongoing Related Work

- Relational Spec
- Functional Code
- Imperative Code
- Assembly Code
- Processor Impl.

Fiat Cryptography

Generates low-level ECC code automatically, with proof. Adopted by Google’s BoringSSL library, thus transitively for TLS in Chrome and Android.

RISC-V

Implementations of RISC-V open instruction set, proving against official formal semantics.
Separate **functionality** and **performance**

Functionality as macros desugaring to a common logic

Performance via proved optimization rules

http://plv.csail.mit.edu/fiat/

Work supported by:

- DARPA
- NSF

I2O: HACMS and BRASS programs  
National Science Foundation