Extensible Extraction of Efficient Imperative Programs
with Foreign Functions, Manually Managed Memory, and Proofs

Clément Pit-Claudel¹  Peng Wang²  Benjamin Delaware³
Jason Gross¹  Adam Chlipala¹

¹MIT CSAIL, ²Google, ³Purdue University

Séminaire Cambium
2021-01-25
The big picture (What are we hoping to achieve?)

We’re working to generate fast, correct code from high-level specifications.
The big picture (What are we hoping to achieve?)

We’re working to generate fast, correct code from high-level specifications

Our ideal workflow:

- Describe your problem in mathematical terms
- Feed it to a domain-specific compiler tailored to your area
- Sprinkle a few problem-specific optimizations
- Enjoy fast code and minimal headaches
Motivation (What do we gain?)

- Improved conformance to domain logic
- Improved security and reliability
- Improved maintainability
- Improved performance
Our target domains (Where will this work?)

Programmers are already used to high-level DSLs
Our target domains (Where will this work?)

Programmers are already used to high-level DSLs

- Databases
  ```sql
  SELECT COUNT(*) WHERE src = packet.src_ip
  AND timestp > DATESUB(second, 1, NOW())
  ```
Our target domains (Where will this work?)

Programmers are already used to high-level DSLs

- **Databases**
  
  ```sql
  SELECT COUNT(*) WHERE src = packet.src_ip
  AND timestamp > DATESUB(second, 1, NOW())
  ```

- **Parsing**
  
  ```
  'for' 'for_init_statement [ condition ] ';' [ expression ] ')' statement
  ```
Programmers are already used to high-level DSLs

- **Databases**
  ```sql
  SELECT COUNT(*) WHERE src = packet.src_ip
  AND timestep > DATESUB(second, 1, NOW())
  ```

- **Parsing**
  ```
  'for' '(' for_init_statement [ condition ] ';'
  [ expression ] ')' statement
  ```

- **Image manipulation** (mogrify), constraint-solving (SMT), pattern-matching (grep, find), file-system manipulation (rsync), etc.
Our pipeline (How do we get assembly from specs?)
Our pipeline (How do we get assembly from specs?)

Usual approach: Extraction to OCaml

- Low assurance
- Moderate performance
- Low extensibility

RFC → x86
Our pipeline (How do we get assembly from specs?)

Usual approach: Extraction to OCaml

![Diagram showing usual approach process and its deficiencies]

- ❌ Low assurance
- ❌ Moderate performance
- ❌ Low extensibility

Our approach

![Diagram showing our approach process and its benefits]

- ✔ High assurance
- ✔ Good performance
- ✔ High extensibility
Zooming in: this paper

Top:
Correct-by-construction refinement into shallow-embedded functional programs

Bottom:
Verified compilation of imperative programs

Bridging the gap:
Certified extraction?
The problem

Gallina is convenient for proofs and specs, but not for performance

- Purity (no mutation)
- Garbage collection and extra allocation
- Closures and boxing
The problem

Gallina is convenient for proofs and specs, but not for performance

- Purity (no mutation)
- Garbage collection and extra allocation
- Closures and boxing

Key challenge: build trustworthy, efficient, extensible compilers for domain-specific functional languages
An example: increment all numbers in a list by n

**OCaml version**

```ocaml
let incrall xs n =
    List.map (fun x -> x + n) xs
```

**C version**

```c
for (int i = 0; i < len; i++)
    xs[i] += n;

.L4:
    movdqu (%rax), %xmm0
    addq $16, %rax
    paddq %xmm1, %xmm0
    movups %xmm0, -16(%rax)
    cmpq %rax, %rdx
    jne .L4
```

Problems: Stack usage, allocation, inlining, GC, …

Clément Pit-Claudel
Extensible Extraction of Efficient Imperative Programs
Challenge: proofs and extensibility

Programmers often know what they want, but there’s no way to teach the compiler how to get there

- Many transformations are tricky to express as rewritings
- Compiler internals are complex and rely on subtle invariants
Challenge: proofs and extensibility

Programmers often know what they want, but there’s no way to teach the compiler how to get there

- Many transformations are tricky to express as rewritings
- Compiler internals are complex and rely on subtle invariants

Typical compilers are not very extensible:

- Best case: single-language rewriting (e.g. rewrite rules in GHC)
- Typical case: source-level annotations and ugly hacks
Our solution

Use custom, problem-dependent translations

- Use a single input language, but *one compiler per domain*
- Switch out pure functions for better ones
  E.g. change *map* to array loop
- Use domain-appropriate datastructures
  E.g. convert *list bool* to *char*

```plaintext
fold encW₈ ws []
  ↓
len := Vector.len(ws);
output := Buffer.new!(len);
idx := 0; done := len != 0;
While (!done)
  hd := List.pop!(ws);
  Buffer.write₈!(idx, ws);
  idx := idx + 1;
  done := List.empty?(ws);
EndWhile
List.delete!(ws);
return output;
```
Core idea: Leverage dialogue with Coq to soundly “synthesize” an imperative program

- Use `Ltac` to derive a proof of the following theorem:

  $$\exists \text{prog}, \forall a, \{ \text{"arg"} \mapsto a \} \text{prog} \{ \text{"output"} \mapsto f\ a \}$$

- Extract a witness (the compiled program) from the (constructive) proof
Assembling and extending compilers

Our DSL compilers are collections of tactics and lemmas

- Basic syntax lemmas (if, let, arithmetic exprs…)
- Custom rewriting lemmas (map \(\circ\) map, firstn \(\circ\) filter)
- Cross-language lemmas (map \(\rightarrow\) for, cons \(\rightarrow\) push)
Assembling and extending compilers

Our DSL compilers are collections of tactics and lemmas

- Basic syntax lemmas (\texttt{if}, \texttt{let}, arithmetic exprs...)
- Custom rewriting lemmas (\texttt{map} \circ \texttt{map}, \texttt{firstn} \circ \texttt{filter})
- Cross-language lemmas (\texttt{map} \rightarrow \texttt{for}, \texttt{cons} \rightarrow \texttt{push})
Assembling and extending compilers

Users can safely supply new building blocks
Assembling and extending compilers

Users can safely supply new building blocks

- New lemmas to support new datastructures and rewrites
- Tactics to apply these lemmas
- Decision procedures to prove side-conditions
Datastructures

Reasoning on Coq lists is convenient, but leads to inefficient code
Datastructures

Reasoning on Coq lists is convenient, but leads to inefficient code

- `list (key * value)`
  → `hashtbl<key, value>`

- `list bit`
  → `char*` or `vector<uint8_t>`

- `list git_commit_t`
  → `linked_list<git_commit_t>`

- `list (option record_t)`
  → `array of bit_t<sizeof(record_t) + 1>`

Purely functional datastructures are not enough (purity, locality, memory).

Our users map their Coq data to imperative representations and specify how to translate `fold`, `map`, etc.
Given a Gallina function $f$, produce a Facade term $\text{prog}$ such that

$$\forall a, \{ \text{"arg" } \mapsto a \} \text{ prog } \{ \text{"output" } \mapsto f \ a \}$$

**Source language**: Subsets of Gallina + nondeterminism monad

**Target language**: Facade, a simple imperative language enforcing linearity
Inputs and outputs

Inputs:
- A nondeterministic Gallina program $f$
- Typeclass instances to map Gallina types to low-level ADTs
- An initial memory layout ("calling convention")
- An expected final memory layout ("return convention")

Output:
- a Facade module
- Syntax of the generated program
- Signatures and specs of external functions
Inputs and outputs

**Inputs:**
- A nondeterministic Gallina program \( f \)
- Typeclass instances to map Gallina types to low-level ADTs
- An initial memory layout (“calling convention”)
- An expected final memory layout (“return convention”)

**Output:**
- Syntax of the generated program
- Signatures and specs of external functions
Inputs and outputs

- **Inputs:**
  - A nondeterministic Gallina program \( f \)
  - Typeclass instances to map Gallina types to low-level ADTs
  - An initial memory layout (“calling convention”)
  - An expected final memory layout (“return convention”)

- **Output:** a Facade module
  - Syntax of the generated program
  - Signatures and specs of external functions

Final linking step checks that external modules implement specs.
Code generation through proof search

Source: \( r \leftarrow \text{anyOf} \ l; \text{if } r < 7 \text{ then } r \text{ else } 7 \)
Code generation through proof search

Source: \( r \leftarrow \text{anyOf } l; \text{ if } r < 7 \text{ then } r \text{ else } 7 \)

\{
"l" \mapsto l \}
?? \{
"l" \mapsto l;
\text{ "out" } \mapsto r \leftarrow \text{anyOf } l;
\text{ if } r < 7 \text{ then } r \text{ else } 7\}
Code generation through proof search

Source: \[ r \leftarrow \text{anyOf } l; \text{ if } r < 7 \text{ then } r \text{ else } 7 \]

\[
\begin{align*}
\{"l" & \mapsto l\} \ ?1 \ \{"l" & \mapsto l; \\
& \quad "r" \mapsto \text{anyOf } l\} \\
\{"l" & \mapsto l; \quad ?2 \ \{"l" & \mapsto l; \\
& \quad "r" \mapsto \text{anyOf } l\} \quad "r" \mapsto \text{anyOf } l \text{ as } r; \\
& \quad "out" \mapsto \text{if } r < 7 \text{ then } r \text{ else } 7\} 
\end{align*}
\]
Code generation through proof search

Source: $r \leftarrow \text{anyOf } l; \text{if } r < 7 \text{ then } r \text{ else } 7$

```plaintext
{"l" ↦ l}  r := Bag.peek(l)  {
  "l" ↦ l;
  "r" ↦ \text{anyOf } l
}

{"l" ↦ l;  ?2 {"l" ↦ l;  
  "r" ↦ \text{anyOf } l}  "r" ↦ \text{anyOf } l \text{ as } r;  
  "out" ↦ \text{if } r < 7 \text{ then } r \text{ else } 7}
```
Code generation through proof search

**Source:** \( r \leftarrow \text{anyOf } l; \text{if } r < 7 \text{ then } r \text{ else } 7 \)

\[
\{"l" \mapsto l\} r := \text{Bag.peek}(l) \{"l" \mapsto l; "r" \mapsto \text{anyOf } l\}
\]

\[
\{...\} ?2 \{"out" \mapsto \text{if } r < 7 \text{ then } r \text{ else } 7\} \]
Code generation through proof search

Source: \( r \leftarrow \text{anyOf} \ l; \quad \text{if} \quad r < 7 \quad \text{then} \quad r \quad \text{else} \quad 7 \)

\[
\begin{align*}
\{"l" \mapsto l\} \quad r & := \text{Bag.peek}(l) \quad \{"l" \mapsto l; \\
& \quad "r" \mapsto \text{anyOf} \ l\} \\
\{\ldots\} \ ?21 \quad \{\ldots"t" \mapsto r < 7\} \\
\{\ldots"t" \mapsto \text{true}\} \ ?22 \quad \{\ldots"t" \mapsto \text{true}; \ "out" \mapsto r\} \\
\{\ldots"t" \mapsto \text{false}\} \ ?23 \quad \{\ldots"t" \mapsto \text{false}; \ "out" \mapsto 7\}
\end{align*}
\]
Code generation through proof search

Source: \( r \leftarrow \text{anyOf } l; \text{ if } r < 7 \text{ then } r \text{ else } 7 \)

\[
\{"l" \mapsto l\} \quad r := \text{Bag.peek}(l) \quad \{"l" \mapsto l; \quad "r" \mapsto \text{anyOf } l\}
\]

\[
\{\ldots\} \quad t := r<7 \quad \{\ldots"t" \mapsto r < 7\}
\]

\[
\{\ldots"t" \mapsto \text{true}\} \quad ?22 \quad \{\ldots"t" \mapsto \text{true}; "out" \mapsto r\}
\]

\[
\{\ldots"t" \mapsto \text{false}\} \quad ?23 \quad \{\ldots"t" \mapsto \text{false}; "out" \mapsto 7\}
\]
Code generation through proof search

Source: \( r \leftarrow \text{anyOf} \ l; \text{if} \ r < 7 \ \text{then} \ r \ \text{else} \ 7 \)

\[
\{"l" \mapsto l \} \ r := \text{Bag.peek}(l) \ \{"l" \mapsto l; "r" \mapsto \text{anyOf} \ l\}
\]

\[
\{\ldots\} \ t := r < 7 \ \{\ldots"t" \mapsto r < 7\}
\]

\[
\{\ldots"t" \mapsto \text{true}\} \ \text{out} := r \ \{\ldots"t" \mapsto \text{true}; "out" \mapsto r\}
\]

\[
\{\ldots"t" \mapsto \text{false}\} \ ?23 \ \{\ldots"t" \mapsto \text{false}; "out" \mapsto 7\} \]
Code generation through proof search

Source: \( r \leftarrow \text{anyOf} \ l; \ if \ r < 7 \ then \ r \ else \ 7 \)

\[
\{"l" \mapsto l\} \ r := \text{Bag.peek}(l) \ \{"l" \mapsto l; \\
\quad "r" \mapsto \text{anyOf} \ l\}
\]

\[
\{\ldots\} \ t := r < 7 \ \{\ldots"t" \mapsto r < 7\}
\]

\[
\{\ldots"t" \mapsto \text{true}\} \ \text{out} := r \ \{\ldots"t" \mapsto \text{true}; \ "\text{out}" \mapsto r\}
\]

\[
\{\ldots"t" \mapsto \text{false}\} \ \text{out} := 7 \ \{\ldots"t" \mapsto \text{false}; \ "\text{out}" \mapsto 7\} 
\]
Code generation through proof search

Source: \( r \leftarrow \text{anyOf} \ l; \text{if} \ r < 7 \ \text{then} \ r \ \text{else} \ 7 \)

\[
\{ \text{l} \mapsto l \} \quad r := \text{Bag.peek}(l) \quad \{ \text{l} \mapsto l; \quad \text{r} \mapsto \text{anyOf} \ l \}
\]

\[
r := \text{Bag.peek}(l);
\]
\[
t := r < 7;
\]
\[
\text{If } t \ \text{Then}
\]
\[
\text{out} := r
\]
\[
\text{Else}
\]
\[
\text{out} := 7
\]
\[
\text{EndIf}
\]
Unit tests

We implemented ~30 functionality tests.

```plaintext
ParametricExtraction
  #vars seq

  #program
  ret (revmap (λ w ⇒ w * 2) seq)

  #arguments
  [[ "seq" ↦ seq ]] :: Nil

  #env UnitTest_Env.

out := List[W].nil()
test := List[W].empty?(seq)
While (test == 0)
  hd := List[W].pop(seq)
r := 2
  hd' := hd * r
call List[W].push(out, hd')
test := List[W].empty?(seq)
EndWhile
call List[W].delete(seq)
```
Adding extensions

Definition nibble_power_of_two_p (w: W) :=
   Eval simpl in bool2w (Inb w (map Word.NToWord [[[1; 2; 4; 8]]%N])).

Ltac _compile_early_hook ::= progress unfold nibble_power_of_two_p.

Example micro_nibble_power_of_two :
   ParametricExtraction
      #vars x
      #program ret (nibble_power_of_two_p (x ⊕ 1))
      #arguments [[["x" ⇝ x as _]] :: Nil
      #env Microbenchmarks_Env.

Proof.
   _compile.
Defined.
Adding extensions

\[ r := 1; \]
\[ l := x + r; \]
\[ r := 1; \]
\[ \text{test} := l = r; \]
\[ \text{If } 1 = \text{test} \text{ Then} \]
\[ \quad \text{out} := 1 \]
\[ \text{Else} \]
\[ \quad r := 1; \]
\[ \quad l := x + r; \]
\[ \quad r := 2; \]
\[ \quad \text{test}_0 := l = r; \]
\[ \quad \text{If } 1 = \text{test}_0 \text{ Then} \]
\[ \quad \quad \text{out} := 1 \]
\[ \quad \text{Else} \]
\[ \quad \quad r := 1; \]
\[ \quad \quad l := x + r; \]
\[ \quad \quad r := 8; \]
\[ \quad \quad \text{test}_2 := l = r; \]
\[ \quad \quad \text{If } 1 = \text{test}_2 \text{ Then} \]
\[ \quad \quad \quad \text{out} := 1 \]
\[ \quad \quad \text{Else} \]
\[ \quad \quad \quad \text{out} := 0 \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
\[ \text{...} \]
\[ \text{test}_1 := l = r; \]
\[ \text{If } 1 = \text{test}_1 \text{ Then} \]
\[ \quad \text{out} := 1 \]
\[ \text{Else} \]
\[ \quad r := 1; \]
\[ \quad l := x + r; \]
\[ \quad r := 8; \]
\[ \quad \text{test}_2 := l = r; \]
\[ \quad \text{If } 1 = \text{test}_2 \text{ Then} \]
\[ \quad \quad \text{out} := 1 \]
\[ \quad \text{Else} \]
\[ \quad \quad \text{out} := 0 \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
\[ \text{EndIf} \]
Adding extensions

Ltac \texttt{\_compile\_early\_hook} ::= \texttt{fail}.

Example \texttt{micro\_nibble\_power\_of\_two}_{\text{intrinsic}} :

\begin{verbatim}
ParametricExtraction
    #vars x
    #program ret (nibble\_power\_of\_two\_p (x \oplus 1))
    #arguments [["x" \mapsto x as _]] :: Nil
    #env Microbenchmarks\_Env ### ("Intrinsics", "nibble\_pow2")
\end{verbatim}

\begin{verbatim}
Proof.
    _\_compile.
    Defined.
\end{verbatim}
Adding extensions

\[ r := 1; \]
\[ \text{arg} := x + r; \]
\[ \text{out} := \text{Intrinsics.nibble}_{\text{pow}_2}(\text{arg}) \]
Macrobenchmark: a process scheduler

Build a process scheduler

- Express operations naturally
- Compile to efficient datastructures
  ‘Bag’ ADT implemented with hand-written binary trees written in Bedrock
A concrete benchmark: database queries

Def ADT {
  rep := QueryStructure SchedulerSchema,

  Def Constructor "Init" : rep := empty,,

  Def Method "Spawn" (r : rep) (new_pid cpu state : W) : rep * ⋄ :=
    Insert (<"pid" :: new_pid, "state" :: state, "cpu" :: cpu> : RawTuple)
    into r!"Processes",

  Def Method "Enumerate" (r : rep) (state : State) : rep * list W :=
    procs ← For (p in r!"Processes")
      Where (p!"state" = state)
      Return (p!"pid");
    ret (r, procs),

  Def Method "GetCPUTime" (r : rep) (id : W) : rep * list W :=
    proc ← For (p in r!"Processes")
      Where (p!"pid" = id)
      Return (p!"cpu");
    ret (r, proc)
}. 
Refined implementation

(** Spawn *)

\[
\begin{align*}
    &a \leftarrow \text{bfind} \ \text{} \_ , \ \text{} \_ , \ \_ , \ \text{} \_ ; \\
    &\text{if length} \ (\text{snd} \ a) = 0 \ \text{then} \\
        &\quad \text{u} \leftarrow \text{binsert} \ (\text{fst} \ a) \ [\text{} \_ , \ \text{} \_ , \ \text{} \_ ] \\
        &\quad \text{ret} \ (\text{fst} \ u, \ true) \\
    &\text{else} \\
        &\quad \text{ret} \ (\text{fst} \ a, \ false)
\end{align*}
\]

(** Enumerate *)

\[
\begin{align*}
    &a \leftarrow \text{bfind} \ \text{} \_ , \ \text{} \_ ; \\
    &\text{ret} \ (\text{fst} \ a, \ \text{revmap} \ (\lambda \ x \Rightarrow \text{GetAttribute} \ x \ 0) \ (\text{snd} \ a))
\end{align*}
\]

(** GetCPUTime *)

\[
\begin{align*}
    &a \leftarrow \text{bfind} \ \text{} \_ , \ \text{} \_ ; \\
    &\text{ret} \ (\text{fst} \ a, \ \text{revmap} \ (\lambda \ x \Rightarrow \text{GetAttribute} \ x \ 2) \ (\text{snd} \ a))
\end{align*}
\]
Extracted implementation (GetCPUPTime)

\[
\begin{align*}
\text{snd} & := \text{Bag}_2[\text{Tuple}[W]].\text{findSecond}(\text{rep}, \text{arg}) \\
\text{ret} & := \text{List}[W].\text{new}() \\
\text{test} & := \text{List}[\text{Tuple}[W]].\text{empty?}(\text{snd}) \\
\text{While} & \ (\text{test} == 0) \\
& \quad \text{head} := \text{List}[\text{Tuple}[W]].\text{pop}(\text{snd}) \\
& \quad \text{pos} := 2 \\
& \quad \text{head'} := \text{Tuple}[W].\text{get}(\text{head}, \text{pos}) \\
& \quad \text{size} := 3 \\
& \quad \text{call Tuple}[W].\text{delete}(\text{head}, \text{size}) \\
& \quad \text{call List}[W].\text{push}(\text{ret}, \text{head'}) \\
& \quad \text{test} := \text{List}[\text{Tuple}[W]].\text{empty?}(\text{snd}) \\
\text{EndWhile} \\
& \text{call List}[\text{Tuple}[W]].\text{delete}(\text{snd})
\end{align*}
\]

**Effort level:** ~500 lines of custom lemmas, ~200 lines of tactics, ~400 lines of assembly, ~300 lines of calling convention specs; 15 + 5 minutes derivations.
Running 20,000 Enumerate and 10,000 GetCPUTime queries after inserting increasingly large numbers of processes. We maintain the invariant that the number of processes in the RUNNING state is about 10. “PG”: PostgreSQL; “CPU”: Unmodified GetCPUTime query; “CPU'”: GetCPUTime query modified to tip PostgreSQL and SQLite about proper index use; “Enum”: Enumerate query.
Related work and alternative approaches

- Imperative/HOL (Lammich 2015): Single monadic input language, no linking
- CakeML extraction (Myreen et al. 2012): Proof-producing extraction, GC
- Cogent (O’Connor et al. 2016): Traditional compiler for a linear language
- Reification + denotation (Mullen et al. 2016): Malloc and forget
- Program extraction: Unverified implementation, limited customization
- Translation validation (e.g. Klein 2009): Our proofs are their witnesses
- Verified compilers (e.g. Leroy 2006): Deep-embeddings only; limited extensibility
- Extensible compilers (e.g. Tobin-Hochstadt 2011, Lerner 2005): Optimization DSLs; Tatlock 2010: DSL for CompCert extensions
- Program synthesis (e.g. Manna 1980)
A lightweight and extensible approach for extracting shallow-embedded functional programs to imperative code with foreign-functions, manually-managed memory, and proofs of correctness

We’re already working on the next iterations of this language, with a focus on security-critical applications:

- Packet parsing
- Bits of crypto code
- Network applications
- ...

Clément Pit-Claudel
Extensible Extraction of Efficient Imperative Programs