Extensible Extraction of Efficient Imperative Programs

with Foreign Functions, Manually Managed Memory, and Proofs

Clément Pit-Claudel[1] Peng Wang[2] Benjamin Delaware[3]
Jason Gross[1] Adam Chlipala[1]

[1]MIT CSAIL, [2]Google, [3]Purdue 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

```
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

```
SELECT COUNT(*) WHERE src = packet.src_ip
AND timestp > DATESUB(second, 1, NOW())
```

- Parsing

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

Our target domains (Where will this work?)

Programmers are already used to high-level DSLs

- Databases

```
SELECT COUNT(*) WHERE src = packet.src_ip
AND timestp > DATESUB(second, 1, NOW())
```

- Parsing

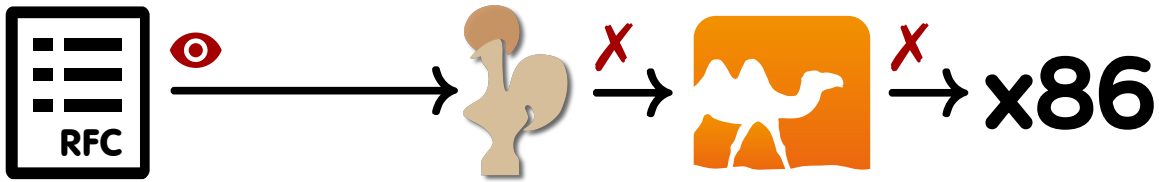
```
'for' '(' for_init_statement [ condition ] ';'
          [ expression ] ')' statement
```

- 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?)

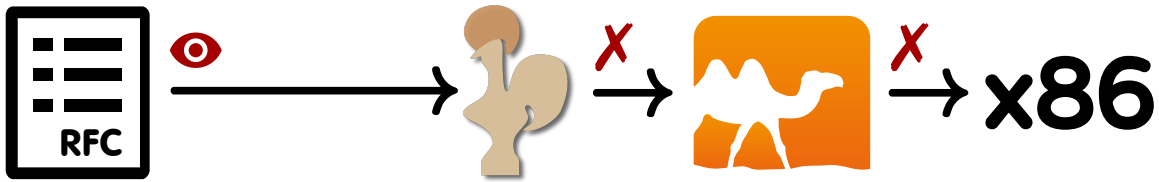
Usual approach: Extraction to OCaml



✗ Low assurance ✗ Moderate performance ✗ Low extensibility

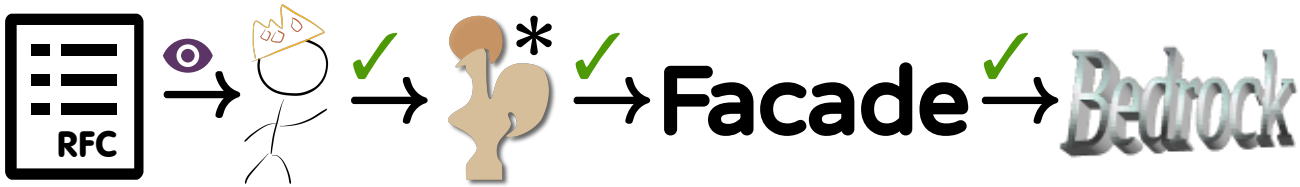
Our pipeline (How do we get assembly from specs?)

Usual approach: Extraction to OCaml



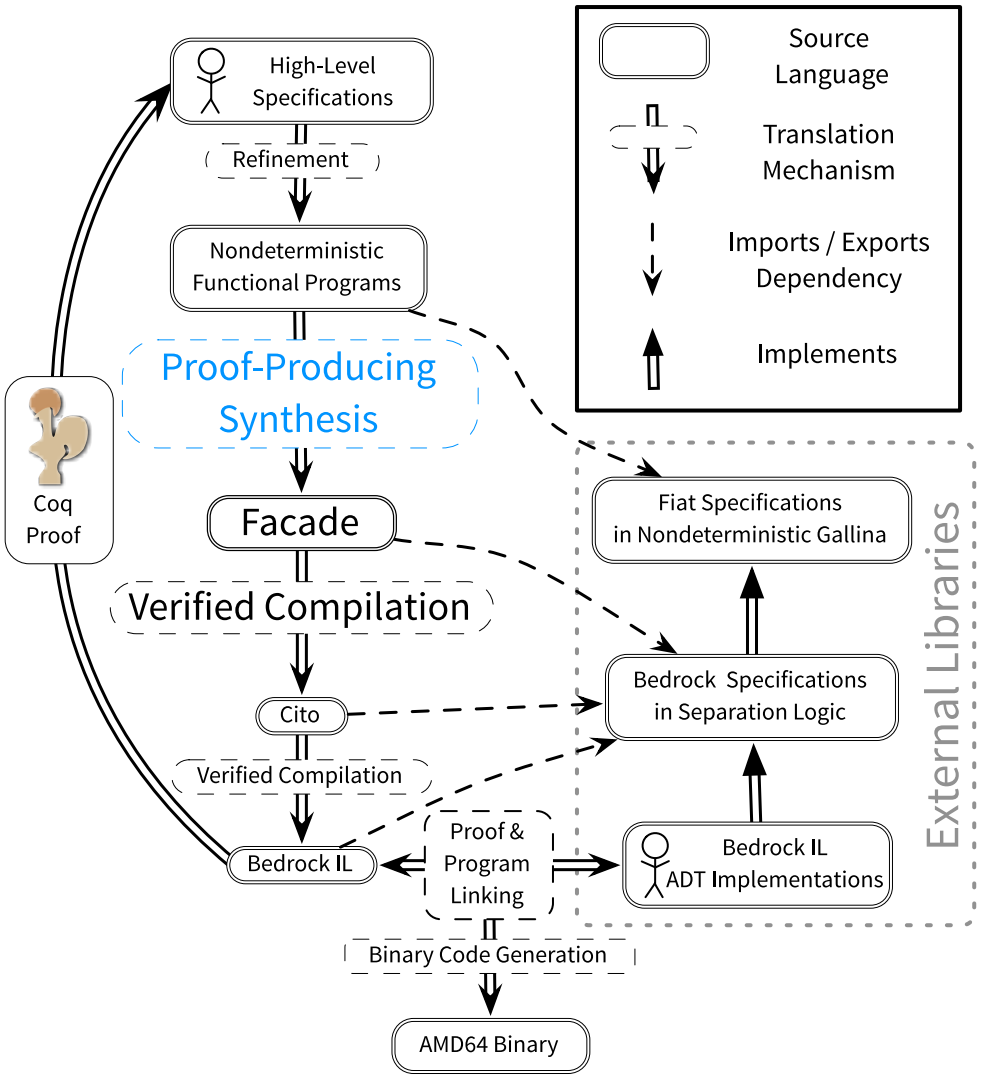
✗ Low assurance ✗ Moderate performance ✗ Low extensibility

Our approach



✓ 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

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

```
camlExample__incrall_5:
        subq    $24, %rsp
        cmpq    $1, %rax
        je      .L101
        movq    (%rax), %rdi
        leaq    -1(%rdi,%rbx), %rdi
        movq    %rdi, (%rsp)
        movq    8(%rax), %rax
        movq    camlExample__1@GOTPCREL(%rip), %rdi
        cmpq    $1, %rax
        je      .L103
        movq    (%rax), %rsi
        leaq    -1(%rsi,%rbx), %rsi
        movq    %rsi, 8(%rsp)
        movq    8(%rax), %rax
        call    camlExample__map_132@PLT
        … (50 lines)
camlExample__map_132:
        subq    $8, %rsp
        cmpq    $1, %rax
        je      .L112
        movq    (%rax), %rsi
        leaq    -1(%rsi,%rbx), %rsi
        movq    %rsi, (%rsp)
        movq    8(%rax), %rax
        call    camlExample__map_132@PLT
.L111:
        movq    %rax, %rbx
.L114:
        subq    $24, %r15
        cmpq    8(%r14), %r15
        jb      .L115
        leaq    8(%r15), %rax
        movq    $2048, -8(%rax)
```

C version

```
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
```

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*`**

`fold` encW_8 `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;
```

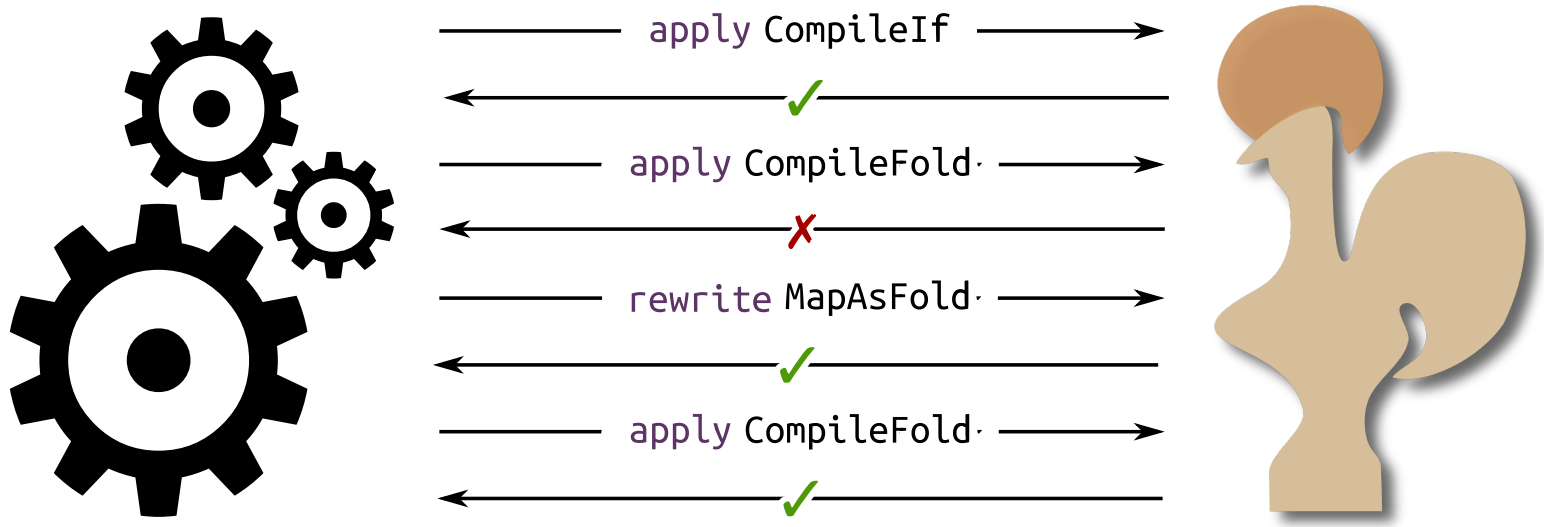
Tactic-based, syntax-driven code-generation

Core idea: Leverage dialogue with Coq to soundly "synthesize" an imperative program

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

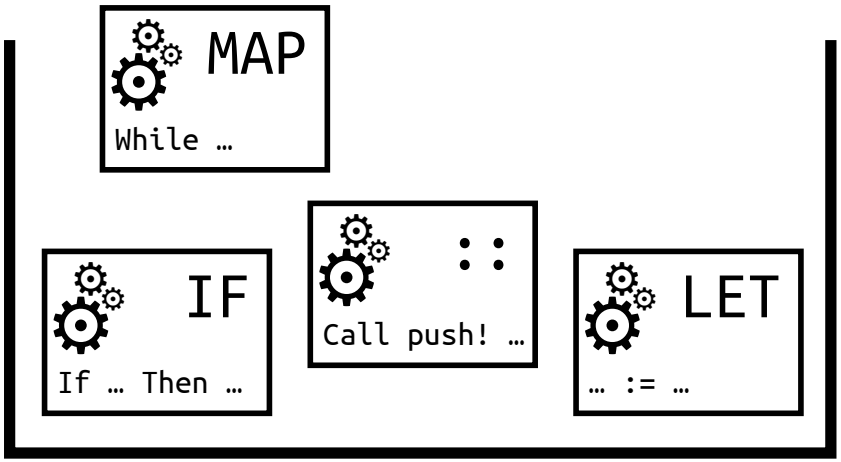
$$\exists\, \texttt{prog}, \forall\, a,\ \{\ \texttt{"arg"} \mapsto a\ \}\ \texttt{prog}\ \{\ \texttt{"output"} \mapsto f\ a\ \}$$

- Extract a witness (the compiled program) from the (constructive) proof



Assembling and extending compilers

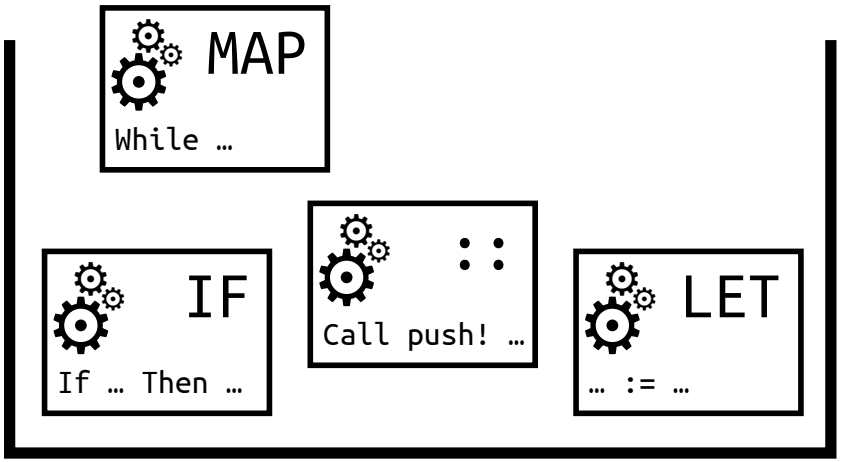
Our DSL compilers are collections of tactics and lemmas



Assembling and extending compilers

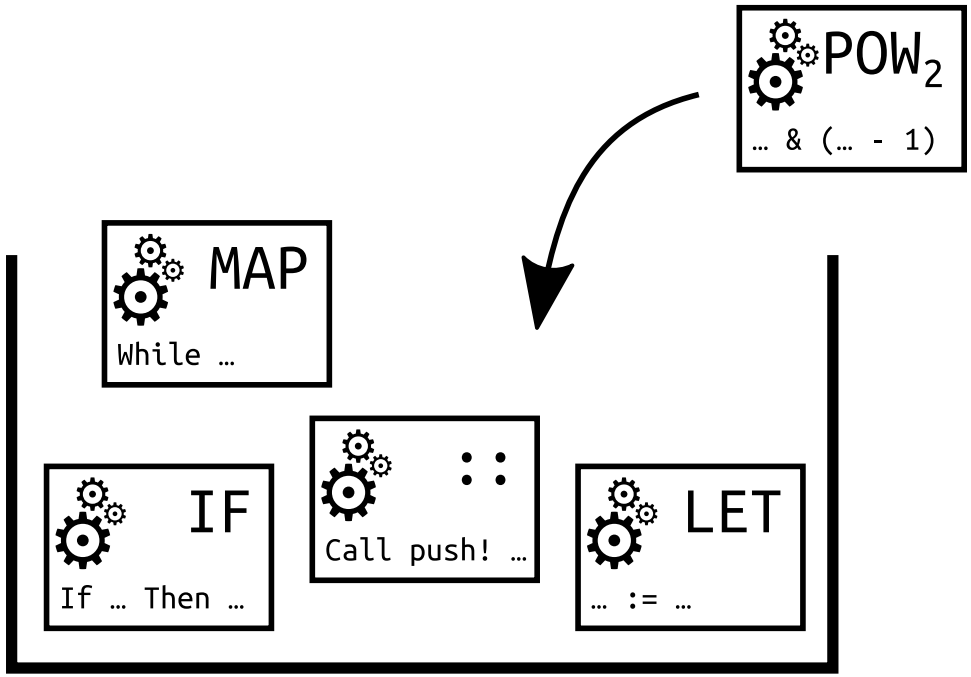
Our DSL compilers are collections of tactics and lemmas

- Basic syntax lemmas (`if`, `let`, arithmetic exprs...)
- Custom rewriting lemmas (`map` ∘ `map`, `firstn` ∘ `filter`)
- Cross-language lemmas (`map` → `for`, `cons` → `push`)



Assembling and extending compilers

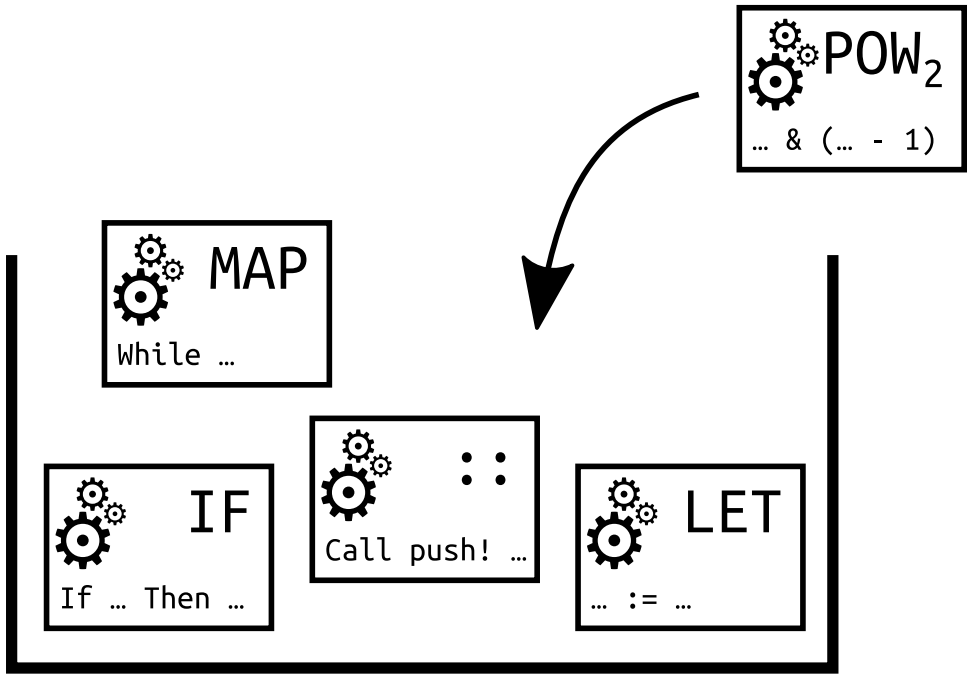
Users can safely supply new building blocks



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.

Precise problem statement

Given a Gallina function f, produce a Facade term `prog` such that

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

Source language: Subsets of Gallina + nondeterminism monad

Target language: Facade, a simple imperative language enforcing linearity

Inputs and outputs

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”)

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 ← anyOf l; if r < 7 then r else 7`

Code generation through proof search

Source:
```
r ← anyOf l; if r < 7 then r else 7
```

```
{"l" ↦ l} ?? {"l" ↦ l;
              "out" ↦ r ← anyOf l;
                      if r < 7 then r else 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

```
        {"l" ↦ l} ?1 {"l" ↦ l;
                      "r" ↦ anyOf l}

{"l" ↦ l;         ?2 {"l" ↦ l;
 "r" ↦ anyOf l}       "r" ↦ anyOf l as r;
                      "out" ↦ if r < 7 then r else 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

 {"l" ↦ l;          ?2 {"l" ↦ l;
  "r" ↦ anyOf l}        "r" ↦ anyOf l as r;
                        "out" ↦ if r < 7 then r else 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

        {…} ?2 {…"out" ↦ if r < 7 then r else 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

                  {…} ?21 {…"t" ↦ r < 7}
       {…"t" ↦ true} ?22 {…"t" ↦ true; "out" ↦ r}
      {…"t" ↦ false} ?23 {…"t" ↦ false; "out" ↦ 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

              {…} t := r<7 {…"t" ↦ r < 7}
  {…"t" ↦ true}    ?22     {…"t" ↦ true; "out" ↦ r}
 {…"t" ↦ false}    ?23     {…"t" ↦ false; "out" ↦ 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

              {…} t := r<7 {…"t" ↦ r < 7}
 {…"t" ↦ true} out := r {…"t" ↦ true; "out" ↦ r}
{…"t" ↦ false}    ?23    {…"t" ↦ false; "out" ↦ 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

              {…} t := r<7 {…"t" ↦ r < 7}
 {…"t" ↦ true} out := r {…"t" ↦ true; "out" ↦ r}
{…"t" ↦ false} out := 7 {…"t" ↦ false; "out" ↦ 7}
```

Code generation through proof search

Source: `r ← anyOf l; if r < 7 then r else 7`

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

```
r := Bag.peek(l);
t := r<7;
If t Then
  out := r
Else
  out := 7
EndIf
```

Unit tests

```
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)
```

We implemented ~30 functionality tests.

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;
test := l = r;
If $1 = test Then
  out := $1
Else
  r := $1;
  l := x + r;
  r := $2;
  test0 := l = r;
  If $1 = test0 Then
    out := $1
  Else
    r := $1;
    l := x + r;
    r := $4;
    …
```

```
    …
    test1 := l = r;
    If $1 = test1 Then
      out := $1
    Else
      r := $1;
      l := x + r;
      r := $8;
      test2 := l = r;
      If $1 = test2 Then
        out := $1
      Else
        out := $0
      EndIf
    EndIf
  EndIf
EndIf
```

Adding extensions

```
Ltac _compile_early_hook ::= fail.

Example micro_nibble_power_of_two_intrinsic    :
  ParametricExtraction
    #vars      x
    #program   ret (nibble_power_of_two_p (x ⊕ 1))
    #arguments [[`"x" ↦ x as _]] :: Nil
    #env       Microbenchmarks_Env ### ("Intrinsics", "nibble_pow2")
               ↦ (Axiomatic (LiftWW _ nibble_power_of_two_p)).
Proof.
  _compile.
Defined.
```

Adding extensions

```
r := $1;
arg := x + r;
out := Intrinsics.nibble_pow2(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 * 𝔹 :=
    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

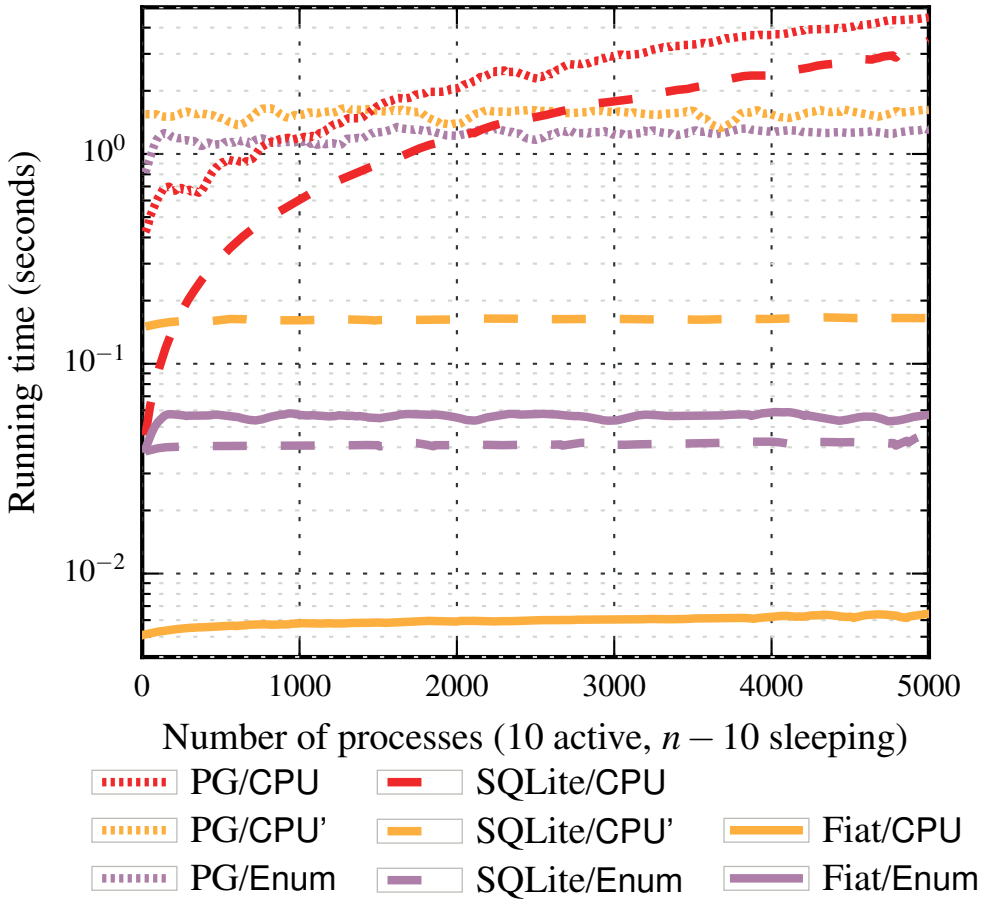
```
(** Spawn *)
a ← bfind $table (_, $new_pid, _);
if length (snd a) = 0 then
  u ← binsert (fst a) [$new_pid, $state, $cpu]
  ret (fst u, true)
else
  ret (fst a, false)

(** Enumerate *)
a ← bfind $table ($state, _, _);
ret (fst a, revmap (λ x ⇒ GetAttribute x 0) (snd a))

(** GetCPUTime *)
a ← bfind $table (_, $id, _);
ret (fst a, revmap (λ x ⇒ GetAttribute x 2) (snd a))
```

Extracted implementation (GetCPUTime)

```
snd := Bag_2[Tuple[W]].findSecond(rep, arg)
ret := List[W].new()
test := List[Tuple[W]].empty?(snd)
While (test == 0)
  head := List[Tuple[W]].pop(snd)
  pos := 2
  head' := Tuple[W].get(head, pos)
  size := 3
  call Tuple[W].delete(head, size)
  call List[W].push(ret, head')
  test := List[Tuple[W]].empty?(snd)
EndWhile
call List[Tuple[W]].delete(snd)
```

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.

Benchmarks



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)

Conclusion

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
- ...