Generating Verified LLVM from Isabelle/HOL

Peter Lammich

The University of Manchester

September 2019

Motivation: Fast and Verified Algorithms

- Stepwise refinement
  - modular and manageable proofs
  - Isabelle Refinement Framework

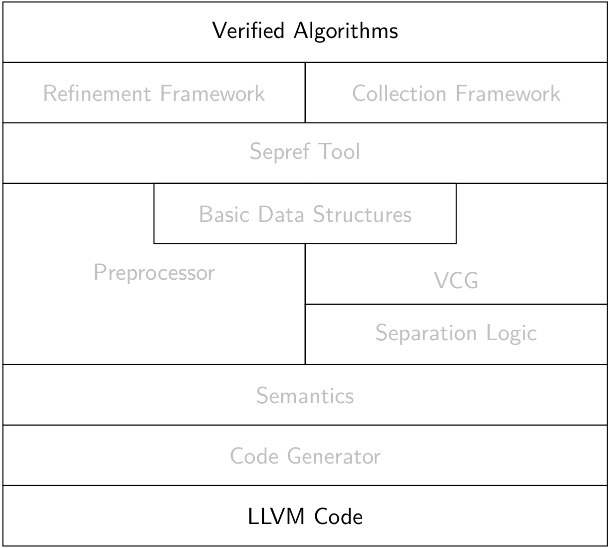
Motivation: Fast and Verified Algorithms

- Stepwise refinement
  - modular and manageable proofs
  - Isabelle Refinement Framework
- Isabelle Code Generator + Imperative HOL
  - generates Haskell, OCaML, SML, Scala
  - doesn't reach efficiency of C/C++

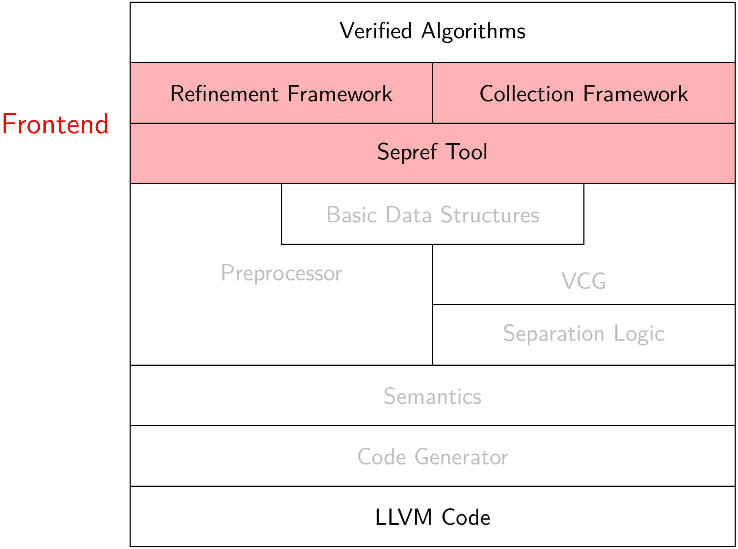
Motivation: Fast and Verified Algorithms

- Stepwise refinement
  - modular and manageable proofs
  - Isabelle Refinement Framework
- Isabelle Code Generator + Imperative HOL
  - generates Haskell, OCaML, SML, Scala
  - doesn't reach efficiency of C/C++
- This paper:
  - code generation to LLVM
  - verification infrastructure
  - link to Refinement Framework

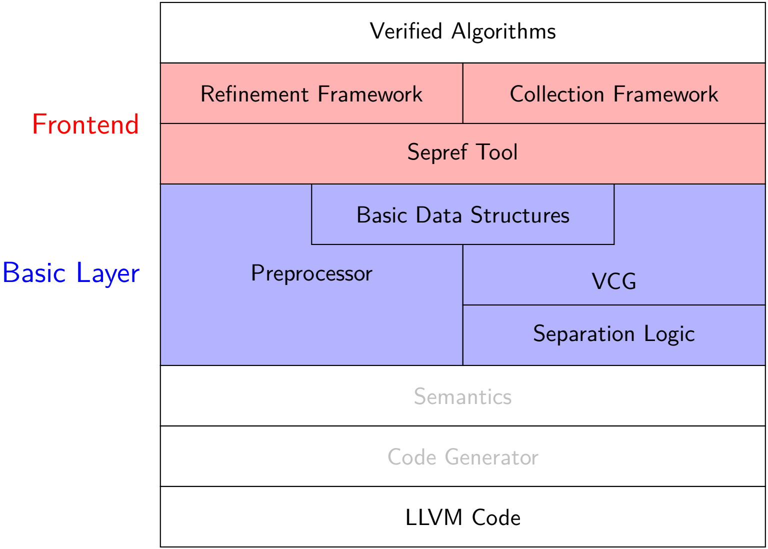
Overview


Verified Algorithms
Refinement Framework
Collection Framework
Sepref Tool
Basic Data Structures
Preprocessor
VCG
Separation Logic
Semantics
Code Generator
LLVM Code

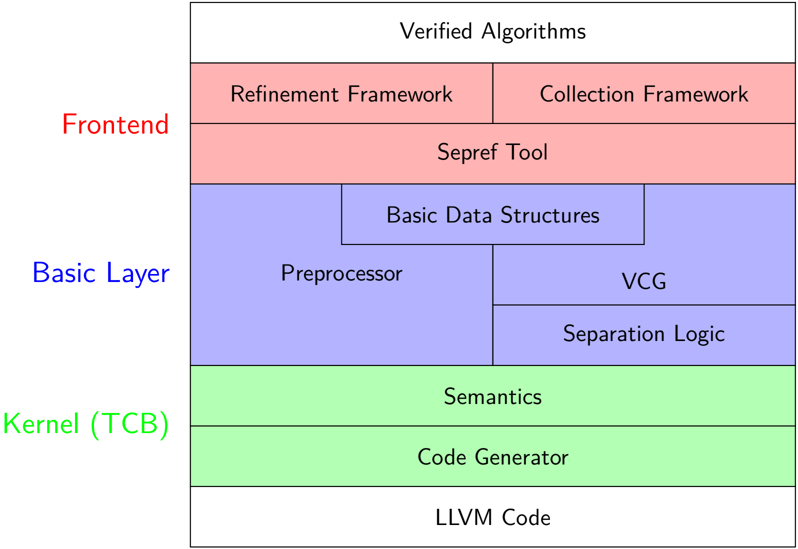
Overview


Frontend
Verified Algorithms
Refinement Framework
Collection Framework
Sepref Tool
Basic Data Structures
Preprocessor
VCG
Separation Logic
Semantics
Code Generator
LLVM Code

Overview


Frontend
Basic Layer
Verified Algorithms
Refinement Framework
Collection Framework
Sepref Tool
Basic Data Structures
Preprocessor
VCG
Separation Logic
Semantics
Code Generator
LLVM Code

Overview


Verified Algorithms
Refinement Framework
Collection Framework
Frontend
Sepref Tool
Basic Data Structures
Basic Layer
Preprocessor
VCG
Separation Logic
Semantics
Kernel (TCB)
Code Generator
LLVM Code

LLVM Semantics

- We don't need to formalize all of LLVM!
  - just enough to express meaningful programs
  - abstract away certain details (e.g. in memory model)

LLVM Semantics

- We don't need to formalize all of LLVM!
  - just enough to express meaningful programs
  - abstract away certain details (e.g. in memory model)
- Trade-off
  - complexity of semantics vs. trusted steps in code generator

LLVM Semantics

- We don't need to formalize all of LLVM!
  - just enough to express meaningful programs
  - abstract away certain details (e.g. in memory model)
- Trade-off
  - complexity of semantics vs. trusted steps in code generator
- Our choice:
  - rather simple semantics
  - code generator does some translations

Basics

- LLVM operations described in state/error monad

  α llM = llM (run: memory $\Rightarrow$ α mres)
  α mres = NTERM | FAIL | SUCC α memory

Basics

- LLVM operations described in state/error monad

  α llM = llM (run: memory $\Rightarrow$ α mres)
  α mres = NTERM | FAIL | SUCC α memory

  ll_udiv :: n word $\Rightarrow$ n word $\Rightarrow$ n word llM
  ll_udiv a b = `do` { `assert` (b $\neq$ 0); `return` (a div b) }

Basics

- LLVM operations described in state/error monad

  α llM = llM (run: memory ⇒ α mres)
  α mres = NTERM | FAIL | SUCC α memory

  ll_udiv :: n word ⇒ n word ⇒ n word llM
  ll_udiv a b = do { assert (b ≠ 0); return (a div b) }

  llc_if b t e = if b≠0 then t else e

Basics

- LLVM operations described in state/error monad

  α llM = llM (run: memory $\Rightarrow$ α mres)
  α mres = NTERM | FAIL | SUCC α memory

  ll_udiv :: n word $\Rightarrow$ n word $\Rightarrow$ n word llM
  ll_udiv a b = do { assert (b $\neq$ 0); return (a div b) }

  llc_if b t e = if b$\neq$0 then t else e

- Recursion via fixed-point

```
llc_while b f s₀ = fixp (λW s.
    do {
      ctd ← b s;
      if ctd≠0 then do {s ← f s; W s} else return s
    }
  ) s₀
```

Shallow Embedding

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding

state/error monad

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding

state/error monad

types: words, pointers, pairs

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Shallow Embedding

state/error monad

types: words, pointers, pairs

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

monad: bind, return

Shallow Embedding

state/error monad

types: words, pointers, pairs

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

standard instructions (ll_<opcode>)

monad: bind, return

Shallow Embedding

state/error monad

types: words, pointers, pairs

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

standard instructions (ll_<opcode>)

arguments: variables and constants

monad: bind, return

Shallow Embedding

state/error monad

types: words, pointers, pairs

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n1 ← ll_sub n 1;
      a  ← fib n1;
      n2 ← ll_sub n 2;
      b  ← fib n2;
      c  ← ll_add a b;
      return c
    }) }
```

control flow (if, [optional: while])

standard instructions (ll_<opcode>)

arguments: variables and constants

monad: bind, return

Shallow Embedding

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

- state/error monad
- types: words, pointers, pairs
- control flow (if, [optional: while])
- standard instructions (ll_<opcode>)
- function calls
- arguments: variables and constants
- monad: bind, return

Code Generation

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t

    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

Code Generation

compiling control flow + pretty printing

```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t

    (return n)
    (do {
      n₁ ← ll_sub n 1;
      a  ← fib n₁;
      n₂ ← ll_sub n 2;
      b  ← fib n₂;
      c  ← ll_add a b;
      return c
    }) }
```

```
define i64 @fib(i64 %x) {
  start:
    %t = icmp ule i64 %x, 1
    br i1 %t, label %then, label %else
  then:
    br label %ctd_if
  else:
    %n_1 = sub i64 %x, 1
    %a   = call i64 @fib (i64 %n_1)
    %n_2 = sub i64 %x, 2
    %b   = call i64 @fib (i64 %n_2)
    %c   = add i64 %a, %b
    br label %ctd_if
  ctd_if:
    %x1a = phi i64 [%x,%then], [%c,%else]
    ret i64 %x1a }
```

Memory Model

- Inspired by CompCert v1. But with structured values.

  memory = block list   block = val list option
  val = n word | ptr | val×val
  rptr = NULL | ADDR nat nat (dir list)   dir = FST | SND

  - ADDR i j p block index, value index, path to value

Memory Model

- Inspired by CompCert v1. But with structured values.

  memory = block list   block = val list option
  val = n word | ptr | val×val
  rptr = NULL | ADDR nat nat (dir list)   dir = FST | SND

  - ADDR i j p block index, value index, path to value

- Typeclass llvm_rep: shallow to deep embedding

  to_val :: 'a ⇒ val
  from_val :: val ⇒ 'a
  init :: 'a – Zero initializer

Memory Model

- Inspired by CompCert v1. But with structured values.

  memory = block list   block = val list option
  val = n word | ptr | val$\times$val
  rptr = NULL | ADDR nat nat (dir list)   dir = FST | SND

  - ADDR i j p block index, value index, path to value

- Typeclass llvm_rep: shallow to deep embedding

  to_val :: 'a $\Rightarrow$ val
  from_val :: val $\Rightarrow$ 'a
  init :: 'a – Zero initializer

- Shallow pointers carry phantom type

  'a ptr = PTR rptr

Example: malloc

```
allocn (v::val) (s::nat) = do {
  bs ← get;
  set (bs@[Some (replicate s v)]);
  return (ADDR |bs| 0 []) }
```

Example: malloc

```
allocn (v::val) (s::nat) = do {
  bs ← get;
  set (bs@[Some (replicate s v)]);
  return (ADDR |bs| 0 []) }

ll_malloc (s::n word) :: 'a ptr = do {
  assert (unat n > 0); – Disallow empty malloc
  r ← allocn (to_val (init::'a)) (unat n);
  return (PTR r) }
```

Example: malloc

```
allocn (v::val) (s::nat) = do {
  bs ← get;
  set (bs@[Some (replicate s v)]);
  return (ADDR |bs| 0 []) }

ll_malloc (s::n word) :: 'a ptr = do {
  assert (unat n > 0); – Disallow empty malloc
  r ← allocn (to_val (init::'a)) (unat n);
  return (PTR r) }
```

- Code generator maps ll_malloc to libc's calloc.
  - out-of-memory: terminate in defined way exit(1)

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)
  - monomorphization (instantiate polymorphic definitions)

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)
  - monomorphization (instantiate polymorphic definitions)
  - flattening of expressions

    `return` $((a+b)+c) \mapsto$ `do` $\{t \leftarrow ll\_add\ a\ b;\ ll\_add\ t\ c\}$

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually

- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)
  - monomorphization (instantiate polymorphic definitions)
  - flattening of expressions

    return ((a+b)+c) ↦ do {t←ll_add a b; ll_add t c}

  - tuples

    return (a,b) ↦ do { t←ll_insert$_1$ init a; ll_insert$_2$ t b }

Preprocessor

- Restricted terms accepted by code generator
  - good to keep code generation simple
  - tedious to write manually
- Preprocessor transforms terms into restricted format
  - proves equality (via Isabelle kernel)
  - monomorphization (instantiate polymorphic definitions)
  - flattening of expressions

    `return` $((a+b)+c) \mapsto$ `do` $\{t \leftarrow \text{ll\_add a b; ll\_add t c}\}$

  - tuples

    `return` $(a,b) \mapsto$ `do` $\{ t \leftarrow \text{ll\_insert}_1 \text{ init a; ll\_insert}_2 \text{ t b } \}$

  - Define recursive functions for fixed points

Example: Preprocessing Euclid's Algorithm

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
 (a,b) ← llc_while
   (λ(a,b) ⇒ ll_cmp (a ≠ b))
   (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
   (a,b);
 return a }
```

Example: Preprocessing Euclid's Algorithm

```
euclid :: 64 word ⇒ 64 word ⇒ 64 word
euclid a b = do {
 (a,b) ← llc_while
   (λ(a,b) ⇒ ll_cmp (a ≠ b))
   (λ(a,b) ⇒ if (a≤b) then return (a,b−a) else return (a−b,b))
   (a,b);
 return a }
```

preprocessor defines function euclid_0 and proves

```
euclid a b = do {
   ab ← ll_insert₁ init a; ab ← ll_insert₂ ab b;
   ab ← euclid₀ ab;
   ll_extract₁ ab  }
euclid₀ s = do {
 a ← ll_extract₁ s;
 b ← ll_extract₂ s;
 ctd ← ll_icmp_ne a b;
 llc_if ctd do {...; euclid₀ ...} }
```

Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples

α :: memory $\rightarrow$ amemory :: sep_algebra
wp c Q s $=$ $\exists$r s$'$. run c s $=$ SUCC r s$'$ $\wedge$ Q r (α s$'$)
$\models$ $\{$P$\}$ c $\{$Q$\}$ $=$ $\forall$F s. (P$*$F) (α s) $\longrightarrow$ wp c (λr s$'$. (Q r $*$ F) s$'$) s

Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples

    α :: memory $\rightarrow$ amemory :: sep_algebra
    wp c Q s $=$ $\exists$r s$'$. run c s $=$ SUCC r s$'$ $\wedge$ Q r (α s$'$)
    $\models$ $\{$P$\}$ c $\{$Q$\}$ $=$ $\forall$F s. (P$*$F) (α s) $\longrightarrow$ wp c (λr s$'$. (Q r $*$ F) s$'$) s

  - memory primitives
    p$\mapsto$x – p points to value x
    m_tag n p – ownership of block (not its contents)

    range $\{$i$_1$,...,i_n$\}$ f p $=$ (p$+$i$_1$)$\mapsto$(f i$_1$) $*$ ... $*$ (p$+$i_n)$\mapsto$(f i_n)

Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples

    α :: memory $\rightarrow$ amemory :: sep_algebra
    wp c Q s = $\exists$r s'. run c s = SUCC r s' $\wedge$ Q r (α s')
    $\models$ {P} c {Q} = $\forall$F s. (P$*$F) (α s) $\longrightarrow$ wp c (λr s'. (Q r $*$ F) s') s

  - memory primitives
    p$\mapsto$x – p points to value x
    m_tag n p – ownership of block (not its contents)

    range {i_1,...,i_n} f p = (p+i_1)$\mapsto$(f i_1) $*$ ... $*$ (p+i_n)$\mapsto$(f i_n)

  - rules for commands

    b $\neq$ 0 $\Longrightarrow$ $\models$ {$\square$} ll_udiv a b {λr. r = a div b}
    $\models$ {n$\neq$0} ll_malloc n {λp. range {0..<n} (λ_. init) p $*$ m_tag n p}
    $\models$ {p$\mapsto$x} ll_load p {λr. r=x $*$ p$\mapsto$x}

Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples

    α :: memory $\rightarrow$ amemory :: sep_algebra
    wp c Q s = $\exists$r s′. run c s = SUCC r s′ $\wedge$ Q r (α s′)
    $\models$ {P} c {Q} = $\forall$F s. (P$*$F) (α s) $\longrightarrow$ wp c (λr s′. (Q r $*$ F) s′) s

  - memory primitives
    p$\mapsto$x – p points to value x
    m_tag n p – ownership of block (not its contents)

    range {i_1,...,i_n} f p = (p+i_1)$\mapsto$(f i_1) $*$ ... $*$ (p+i_n)$\mapsto$(f i_n)

  - rules for commands

    b $\neq$ 0 $\Longrightarrow$ $\models$ {$\square$} ll_udiv a b {λr. r = a div b}
    $\models$ {n$\neq$0} ll_malloc n {λp. range {0..<n} (λ_. init) p $*$ m_tag n p}
    $\models$ {p$\mapsto$x} ll_load p {λr. r=x $*$ p$\mapsto$x}

- Automation: VCG, frame inference, heuristics to discharge VCs

Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples

    α :: memory $\rightarrow$ amemory :: sep_algebra
    wp c Q s = $\exists$r s′. run c s = SUCC r s′ $\wedge$ Q r (α s′)
    $\models$ {P} c {Q} = $\forall$F s. (P$*$F) (α s) $\longrightarrow$ wp c (λr s′. (Q r $*$ F) s′) s

  - memory primitives
    p$\mapsto$x – p points to value x
    m_tag n p – ownership of block (not its contents)

    range {i_1,. . . ,i_n} f p = (p+i_1)$\mapsto$(f i_1) $*$ . . . $*$ (p+i_n)$\mapsto$(f i_n)

  - rules for commands

    b $\neq$ 0 $\Longrightarrow$ $\models$ {$\square$} ll_udiv a b {λr. r = a div b}
    $\models$ {n$\neq$0} ll_malloc n {λp. range {0..<n} (λ_. init) p $*$ m_tag n p}
    $\models$ {p$\mapsto$x} ll_load p {λr. r=x $*$ p$\mapsto$x}

- Automation: VCG, frame inference, heuristics to discharge VCs
- Basic Data Structures: signed/unsigned integers, Booleans, arrays

Example: Proving Euclid's Algorithm

lemma

$\models \{\mathsf{uint}_{64}\ a\ a_\dagger * \mathsf{uint}_{64}\ b\ b_\dagger * 0{<}a * 0{<}b\}\ \mathsf{euclid}\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ \mathsf{uint}_{64}\ (\mathsf{gcd}\ a\ b)\ r_\dagger\}$

Example: Proving Euclid's Algorithm

lemma
$\models \{\mathsf{uint}_{64}\ \mathsf{a}\ \mathsf{a}_\dagger * \mathsf{uint}_{64}\ \mathsf{b}\ \mathsf{b}_\dagger * 0{<}\mathsf{a} * 0{<}\mathsf{b}\}\ \mathsf{euclid}\ \mathsf{a}_\dagger\ \mathsf{b}_\dagger\ \{\lambda \mathsf{r}_\dagger.\ \mathsf{uint}_{64}\ (\mathsf{gcd}\ \mathsf{a}\ \mathsf{b})\ \mathsf{r}_\dagger\}$

unfolding euclid_def
apply (rewrite annotate_llc_while[**where** I = ... **and** R = measure nat])

Example: Proving Euclid's Algorithm

lemma

$\models \{\text{uint}_{64}\ a\ a_\dagger * \text{uint}_{64}\ b\ b_\dagger * 0{<}a * 0{<}b\}\ \text{euclid}\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ \text{uint}_{64}\ (\text{gcd}\ a\ b)\ r_\dagger\}$

unfolding euclid_def

apply (rewrite annotate_llc_while[**where** I = ... **and** R = measure nat])

apply (vcg; clarsimp?)

Example: Proving Euclid's Algorithm

lemma
$\models \{\text{uint}_{64}\ a\ a_\dagger * \text{uint}_{64}\ b\ b_\dagger * 0{<}a * 0{<}b\}\ \text{euclid}\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ \text{uint}_{64}\ (\text{gcd}\ a\ b)\ r_\dagger\}$

unfolding euclid_def
apply (rewrite annotate_llc_while[**where** I = ... **and** R = measure nat])

apply (vcg; clarsimp?)

Subgoals:

1. $\bigwedge x\ y.\ [\![\ \text{gcd}\ x\ y = \text{gcd}\ a\ b;\ x \neq y;\ x \leq y;\ \ldots\ ]\!] \implies \text{gcd}\ x\ (y - x) = \text{gcd}\ a\ b$
2. $\bigwedge x\ y.\ [\![\ \text{gcd}\ x\ y = \text{gcd}\ a\ b;\ \neg\ x \leq y;\ \ldots\ ]\!] \implies \text{gcd}\ (x - y)\ y = \text{gcd}\ a\ b$

Example: Proving Euclid's Algorithm

lemma
$\models \{\text{uint}_{64}\ a\ a_\dagger * \text{uint}_{64}\ b\ b_\dagger * 0{<}a * 0{<}b\}\ \text{euclid}\ a_\dagger\ b_\dagger\ \{\lambda r_\dagger.\ \text{uint}_{64}\ (\text{gcd}\ a\ b)\ r_\dagger\}$

unfolding euclid_def
apply (rewrite annotate_llc_while[**where** I = ... **and** R = measure nat])

apply (vcg; clarsimp?)

Subgoals:
1. $\bigwedge x\ y.\ [\![\ \text{gcd}\ x\ y = \text{gcd}\ a\ b;\ x \neq y;\ x \leq y;\ \ldots\ ]\!] \Longrightarrow \text{gcd}\ x\ (y - x) = \text{gcd}\ a\ b$
2. $\bigwedge x\ y.\ [\![\ \text{gcd}\ x\ y = \text{gcd}\ a\ b;\ \neg\ x \leq y;\ \ldots\ ]\!] \Longrightarrow \text{gcd}\ (x - y)\ y = \text{gcd}\ a\ b$

by (simp_all add: gcd_diff1 gcd_diff1′)

Automatic Refinement

- Isabelle Refinement Framework
  - supports verification by stepwise refinement
  - many verified algorithms already exists

Automatic Refinement

- Isabelle Refinement Framework
  - supports verification by stepwise refinement
  - many verified algorithms already exists
- Sepref tool
  - refinement from Refinement Framework to imperative program
    - already existed for Imperative/HOL
    - we adapted it for LLVM
  - existing proofs can be re-used
    - need to be amended if they use arbitrary-precision integers

Automatic Refinement

- Isabelle Refinement Framework
  - supports verification by stepwise refinement
  - many verified algorithms already exists
- Sepref tool
  - refinement from Refinement Framework to imperative program
    - already existed for Imperative/HOL
    - we adapted it for LLVM
  - existing proofs can be re-used
    - need to be amended if they use arbitrary-precision integers
- Collections Framework
  - provides data structures
  - we ported some to LLVM (work in progress)
    - dense sets/maps of integers (by array)
    - heaps, indexed heaps
    - two-watched-literals for BCP
    - graphs (by adjacency lists)
    - ...

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← WHILEIT (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      ASSERT (l<length xs ∧ h≤length xs ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then RETURN (m+1,h) else RETURN (l,m)
    })
    (0,length xs);
  RETURN l
}
```

Example: Binary Search

```
definition bin_search xs x = do {
  (l,h) ← WHILEIT (bin_search_invar xs x)
    (λ(l,h). l<h)
    (λ(l,h). do {
      ASSERT (l<length xs ∧ h≤length xs ∧ l≤h);
      let m = l + (h−l) div 2;
      if xs!m < x then RETURN (m+1,h) else RETURN (l,m)
    })
    (0,length xs);
  RETURN l
}

lemma bin_search_correct:
  sorted xs ⟹ bin_search xs x ≤ SPEC (λi. i=find_index (λy. x≤y) xs)
```

Example: Binary Search — Refinement

```
sepref_def bin_search_impl is uncurry bin_search
  :: (larray_assn' TYPE(size_t) (sint_assn' TYPE(elem_t)))^k
       * (sint_assn' TYPE(elem_t))^k
     → snat_assn' TYPE(size_t)
  unfolding bin_search_def
  apply (rule hfref_with_rdomI, annot_snat_const TYPE(size_t))
  by sepref
```

Example: Binary Search — Refinement

```
sepref_def bin_search_impl is uncurry bin_search
  :: (larray_assn' TYPE(size_t) (sint_assn' TYPE(elem_t)))^k
      * (sint_assn' TYPE(elem_t))^k
     → snat_assn' TYPE(size_t)
  unfolding bin_search_def
  apply (rule hfref_with_rdomI, annot_snat_const TYPE(size_t))
  by sepref

export_llvm bin_search_impl is int64_t bin_search(larray_t, elem_t)
defines
  typedef uint64_t elem_t;
  typedef struct { int64_t len; elem_t *data; } larray_t;
file code/bin_search.ll
```

Example: Binary Search — Generated Code

Produces LLVM code and header file:

```
typedef uint64_t elem_t;
typedef struct {
  int64_t len;
  elem_t*data;
} larray_t;

int64_t bin_search(larray_t,elem_t);
```

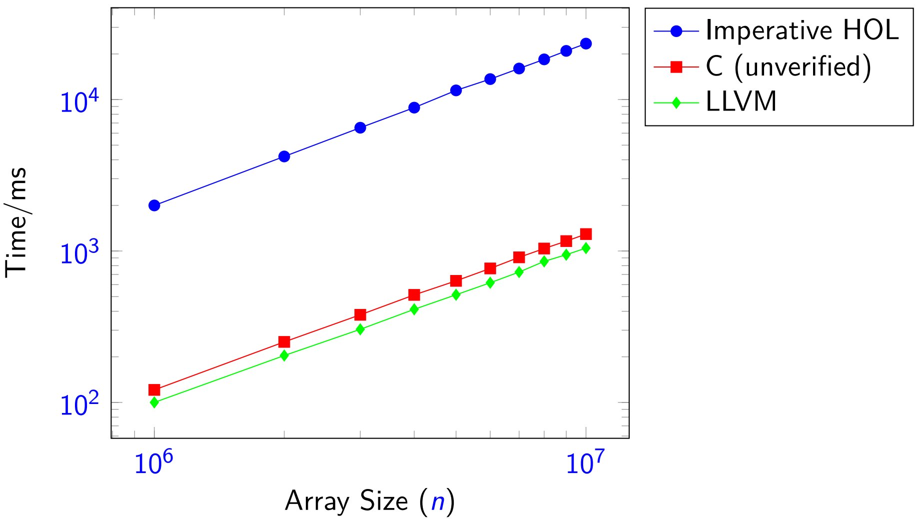
Case Studies

- Binary Search and Knuth-Morris-Pratt
  - manageable amount of changes to original formalization

Case Studies

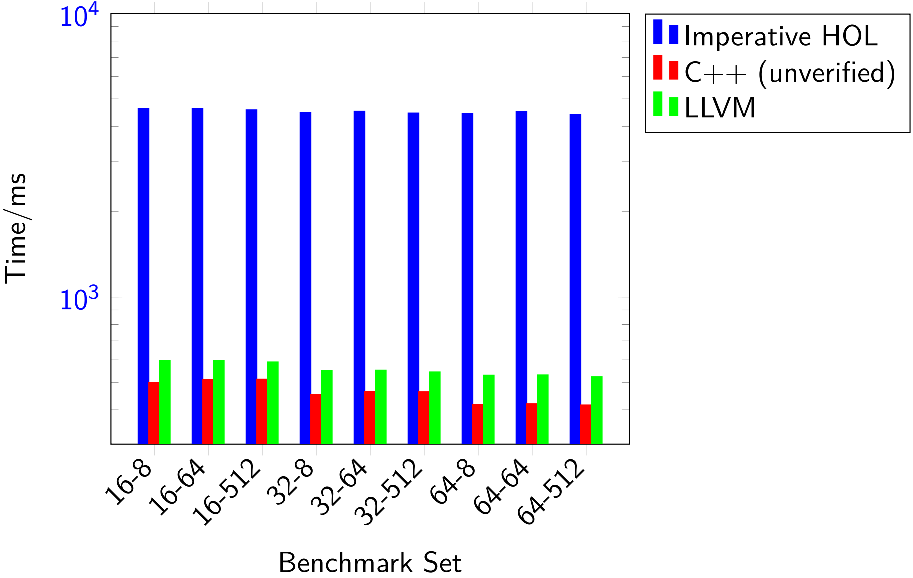
- Binary Search and Knuth-Morris-Pratt
  - manageable amount of changes to original formalization
- Efficiency
  - on par with unverified C/C++
  - one order of magnitude faster than original

Binary Search



Search for the values $0, 2, \ldots < 5n$ in an array $[0, 5, \ldots < 5n]$

Knuth Morris Pratt



Execute *a*-*l* benchmark set from StringBench. Stop at first match.

Conclusions

- Fast and verified algorithms
  - LLVM code generator
  - using Refinement Framework
  - manageable proof overhead
- Case studies
  - generate really fast, verified code
  - re-use existing proofs
- Current/future work
  - more complex algorithms
    - promising (preliminary) results for SAT-solver, Prim's algorithm
  - deeply embedded semantics
  - generic Sepref (Imp-HOL, LLVM) $\times$ (nres, nres+time)

https://github.com/lammich/isabelle_llvm