Generating Verified LLVM from Isabelle/HOL

Peter Lammich

The University of Manchester

December 2019

Motivation

- Desirable properties of software

Motivation

- Desirable properties of software
  - correct

Motivation

- Desirable properties of software
  - correct (formally verified)

Motivation

- Desirable properties of software
    - correct (formally verified)
    - fast

Motivation

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation effort

Motivation

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation and proof effort

Motivation

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation and proof effort
- Choose two!

Motivation

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation and proof effort
- Choose two!
- This talk: towards faster verified algorithms at manageable effort

Introduction

- What does it need to formally verify an algorithm?

Introduction

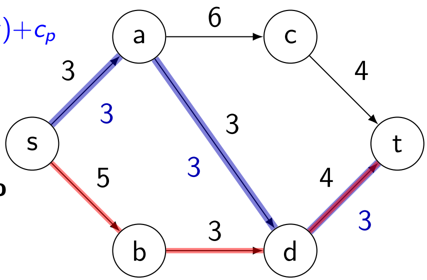
- What does it need to formally verify an algorithm?
  - E.g. maxflow algorithms

Introduction

- What does it need to formally verify an algorithm?
  - E.g. maxflow algorithms

procedure AUGMENT(g, f, p)
  $c_p \leftarrow \min\{g_f(u,v) \mid (u,v) \in p\}$
  for all $(u,v) \in p$ **do**
    if $(u,v) \in g$ **then** $f(u,v) \leftarrow f(u,v) + c_p$
    else $f(v,u) \leftarrow f(v,u) - c_p$
  return f

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u,v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)



g: flow network s, t: source, target g_f: residual network

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$ AUGMENT(g, f, p)

Theorem (Ford-Fulkerson)

For a flow network g and flow f, the following 3 statements are equivalent

1. *f is a maximum flow*
2. *the residual network g_f contains no augmenting path*
3. *$|f|$ is the capacity of a (minimal) cut of g*

Correctness

procedure $\textsc{Edmonds-Karp}(g, s, t)$
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow \textsc{augment}(g, f, p)$

Theorem (Ford-Fulkerson)

For a flow network g and flow f, the following 3 statements are equivalent

1. *f is a maximum flow*
2. *the residual network g_f contains no augmenting path*
3. *$|f|$ is the capacity of a (minimal) cut of g*

Proof.

a few pages of definitions and textbook proof (e.g. Cormen).

□

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$ AUGMENT(g, f, p)

Theorem (Ford-Fulkerson)

For a flow network g and flow f, the following 3 statements are equivalent

1. *f is a maximum flow*
2. *the residual network g_f contains no augmenting path*
3. *$|f|$ is the capacity of a (minimal) cut of g*

Proof.

a few pages of definitions and textbook proof (e.g. Cormen).
using basic concepts such as numbers, sets, and graphs. □

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)

Theorem

Let δ_f be the length of a shortest s, t - path in g_f.
When augmenting with a shortest path,

- *either δ_f decreases*
- *δ_f remains the same, and the number of edges in g_f that lay on a shortest path decreases.*

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$ AUGMENT(g, f, p)

Theorem

Let δ_f be the length of a shortest s, t - path in g_f.
When augmenting with a shortest path,

- *either δ_f decreases*
- *δ_f remains the same, and the number of edges in g_f that lay on a shortest path decreases.*

Proof.

two more textbook pages.

□

Correctness

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)

Theorem

Let δ_f be the length of a shortest s, t - path in g_f.
When augmenting with a shortest path,

- *either δ_f decreases*
- *δ_f remains the same, and the number of edges in g_f that lay on a shortest path decreases.*

Proof.

two more textbook pages.
using lemmas about graphs and shortest paths. □

Background Theory

- E.g. graph theory

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
    - with good library support (to not re-invent too many wheels)

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
  - with good library support (to not re-invent too many wheels)
- we use Isabelle

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
  - with good library support (to not re-invent too many wheels)
- we use Isabelle
  - Isabelle/HOL: based on Higher-Order Logic

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
  - with good library support (to not re-invent too many wheels)
- we use Isabelle
  - Isabelle/HOL: based on Higher-Order Logic
  - powerful automation (e.g. sledgehammer)

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
    - with good library support (to not re-invent too many wheels)
- we use Isabelle
    - Isabelle/HOL: based on Higher-Order Logic
    - powerful automation (e.g. sledgehammer)
    - large collection of libraries

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
  - with good library support (to not re-invent too many wheels)
- we use Isabelle
  - Isabelle/HOL: based on Higher-Order Logic
  - powerful automation (e.g. sledgehammer)
  - large collection of libraries
  - Archive of Formal Proofs

Background Theory

- E.g. graph theory
- Typically requires powerful (interactive) prover
  - with good library support (to not re-invent too many wheels)
- we use Isabelle
  - Isabelle/HOL: based on Higher-Order Logic
  - powerful automation (e.g. sledgehammer)
  - large collection of libraries
  - Archive of Formal Proofs
  - mature, production quality IDE, based on JEdit

Implementation

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$ AUGMENT(g, f, p)

```
int edmonds_karp(int s, int t) {
   int flow = 0;
   vector<int> parent(n);
   int new_flow;

   while (new_flow = bfs(s, t, parent)) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
         int prev = parent[cur];
         capacity[prev][cur] -= new_flow;
         capacity[cur][prev] += new_flow;
         cur = prev;
      }
   }

   return flow;
}
```

textbook proof typically covers abstract algorithm.

Implementation

procedure $\textsc{Edmonds-Karp}(g, s, t)$
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow \textsc{augment}(g, f, p)$

```
int edmonds_karp(int s, int t) {
   int flow = 0;
   vector<int> parent(n);
   int new_flow;

   while (new_flow = bfs(s, t, parent)) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
         int prev = parent[cur];
         capacity[prev][cur] -= new_flow;
         capacity[cur][prev] += new_flow;
         cur = prev;
      }
   }

   return flow;
}
```

textbook proof typically covers abstract algorithm.
but this is quite far from implementation. Still missing:

Implementation

procedure $\textsc{Edmonds-Karp}(g, s, t)$
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow \textsc{augment}(g, f, p)$

```
int edmonds_karp(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
}
```

textbook proof typically covers abstract algorithm.
but this is quite far from implementation. Still missing:

- optimizations: e.g., work on residual network instead of flow

Implementation

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)

```
int edmonds_karp(int s, int t) {
   int flow = 0;
   vector<int> parent(n);
   int new_flow;

   while (new_flow = bfs(s, t, parent)) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
         int prev = parent[cur];
         capacity[prev][cur] -= new_flow;
         capacity[cur][prev] += new_flow;
         cur = prev;
      }
   }

   return flow;
}
```

textbook proof typically covers abstract algorithm.
but this is quite far from implementation. Still missing:

- optimizations: e.g., work on residual network instead of flow
- algorithm to find shortest augmenting path (BFS)

Implementation

procedure $\textsc{Edmonds-Karp}(g, s, t)$
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow \textsc{Augment}(g, f, p)$

```
int edmonds_karp(int s, int t) {
   int flow = 0;
   vector<int> parent(n);
   int new_flow;

   while (new_flow = bfs(s, t, parent)) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
         int prev = parent[cur];
         capacity[prev][cur] -= new_flow;
         capacity[cur][prev] += new_flow;
         cur = prev;
      }
   }

   return flow;
}
```

textbook proof typically covers abstract algorithm.
but this is quite far from implementation. Still missing:

- optimizations: e.g., work on residual network instead of flow
- algorithm to find shortest augmenting path (BFS)
- efficient data structures: adjacency lists, weight matrix, FIFO-queue, ...

Implementation

procedure EDMONDS-KARP(g, s, t)
  $f \leftarrow \lambda(u, v).\ 0$
  while exists augmenting path in g_f **do**
    $p \leftarrow$ shortest augmenting path
    $f \leftarrow$AUGMENT(g, f, p)

```
int edmonds_karp(int s, int t) {
   int flow = 0;
   vector<int> parent(n);
   int new_flow;

   while (new_flow = bfs(s, t, parent)) {
      flow += new_flow;
      int cur = t;
      while (cur != s) {
         int prev = parent[cur];
         capacity[prev][cur] -= new_flow;
         capacity[cur][prev] += new_flow;
         cur = prev;
      }
   }

   return flow;
}
```

textbook proof typically covers abstract algorithm.
but this is quite far from implementation. Still missing:

- optimizations: e.g., work on residual network instead of flow
- algorithm to find shortest augmenting path (BFS)
- efficient data structures: adjacency lists, weight matrix, FIFO-queue, ...
- code extraction

Keeping it Manageable

- A manageable proof needs modularization:

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EDMONDSKARP

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EdmondsKarp
- Data refinement

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EdmondsKarp
- Data refinement
  - BFS implementation uses adjacency lists. EdmondsKarp used abstract graphs.

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EdmondsKarp
- Data refinement
  - BFS implementation uses adjacency lists. EdmondsKarp used abstract graphs.
  - refinement relations between
    - nodes and int64s ($node_{64}$);
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EdmondsKarp
- Data refinement
  - BFS implementation uses adjacency lists. EdmondsKarp used abstract graphs.
  - refinement relations between
    - nodes and int64s ($node_{64}$);
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

$$(s_\dagger, s) \in node_{64};\ (t_\dagger, t) \in node_{64};\ (g_\dagger, g) \in adjl$$
$$\Longrightarrow\ (bfs\ s_\dagger\ t_\dagger\ g_\dagger,\ find\_shortest\ s\ t\ g) \in array$$

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EDMONDSKARP
- Data refinement
  - BFS implementation uses adjacency lists. EDMONDSKARP used abstract graphs.
  - refinement relations between
    - nodes and int64s (node_{64});
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

$$(s_\dagger, s) \in \text{node}_{64};\ (t_\dagger, t) \in \text{node}_{64};\ (g_\dagger, g) \in \text{adjl}$$
$$\Longrightarrow\ (\text{bfs}\ s_\dagger\ t_\dagger\ g_\dagger,\ \text{find\_shortest}\ s\ t\ g) \in \text{array}$$

Shortcut notation: $(\text{bfs}, \text{find\_shortest}) \in \text{node}_{64} \to \text{node}_{64} \to \text{adjl} \to \text{array}$

Keeping it Manageable

- A manageable proof needs modularization:
  - Prove separately, then assemble
- Formal framework: Refinement
  - e.g. implement BFS, and prove it finds shortest paths
  - insert implementation into EdmondsKarp
- Data refinement
  - BFS implementation uses adjacency lists. EdmondsKarp used abstract graphs.
  - refinement relations between
    - nodes and int64s ($node_{64}$);
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

$$(s_\dagger, s) \in node_{64};\ (t_\dagger, t) \in node_{64};\ (g_\dagger, g) \in adjl$$
$$\Longrightarrow\ (bfs\ s_\dagger\ t_\dagger\ g_\dagger,\ find\_shortest\ s\ t\ g) \in array$$

Shortcut notation: $(bfs, find\_shortest) \in node_{64} \rightarrow node_{64} \rightarrow adjl \rightarrow array$

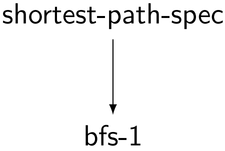
- Implementations used for different parts must fit together!

Refinement Architecture (simplified)

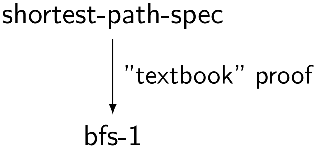
Refinement Architecture (simplified)

shortest-path-spec

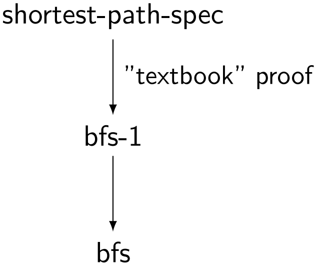
Refinement Architecture (simplified)



Refinement Architecture (simplified)



Refinement Architecture (simplified)



Refinement Architecture (simplified)

shortest-path-spec

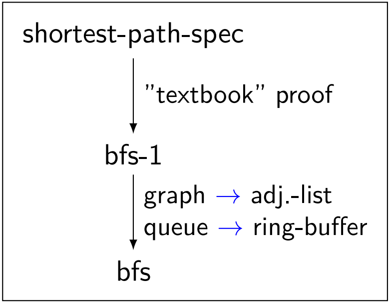
↓ "textbook" proof

bfs-1

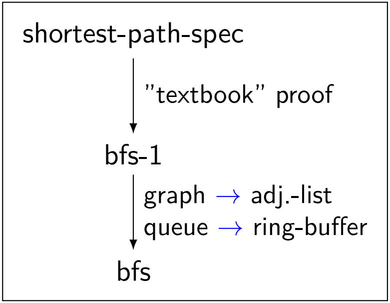
↓ graph $\rightarrow$ adj.-list
queue $\rightarrow$ ring-buffer

bfs

Refinement Architecture (simplified)

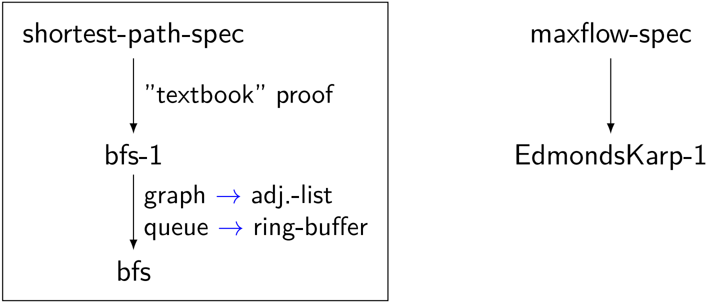


Refinement Architecture (simplified)

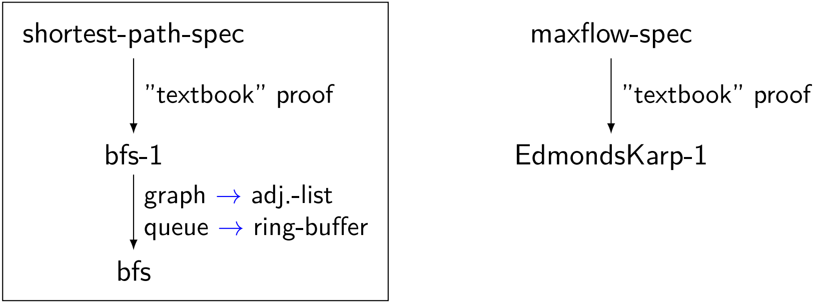


maxflow-spec

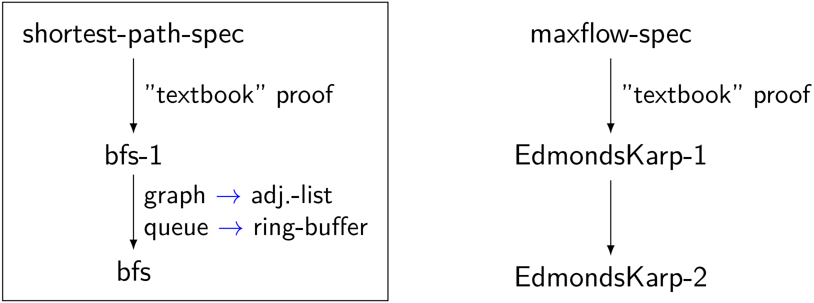
Refinement Architecture (simplified)



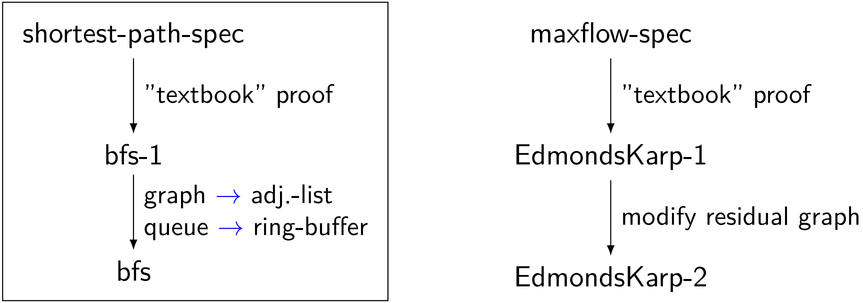
Refinement Architecture (simplified)



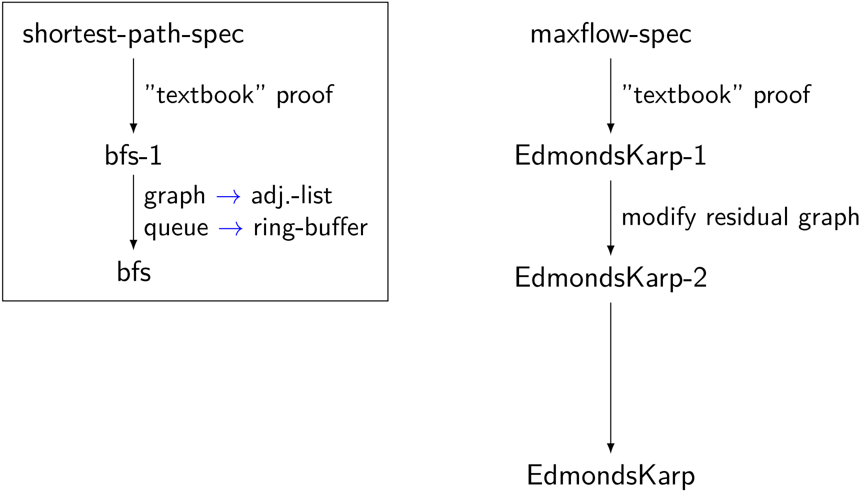
Refinement Architecture (simplified)



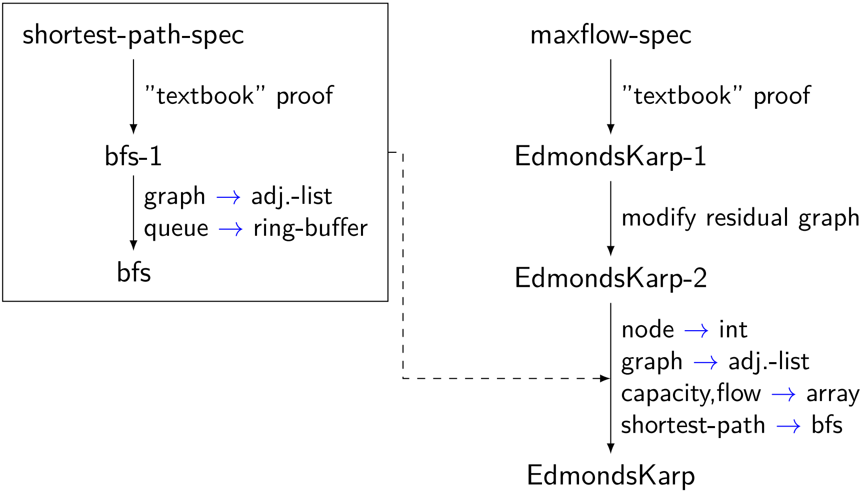
Refinement Architecture (simplified)



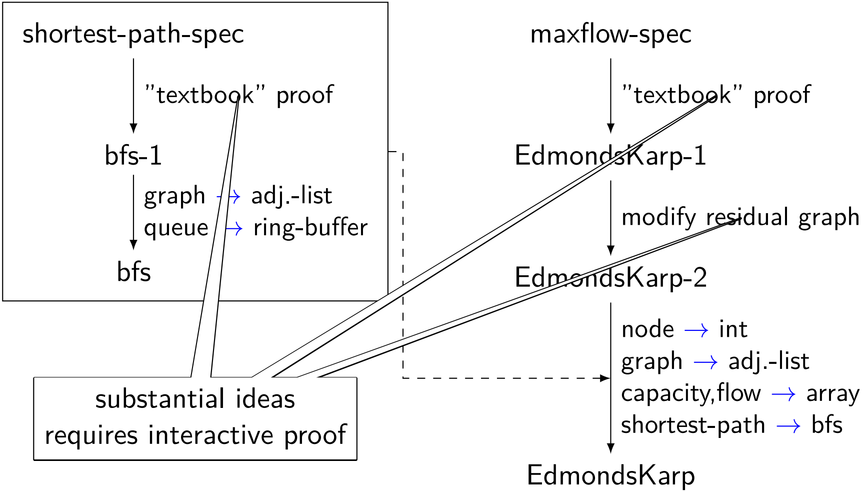
Refinement Architecture (simplified)



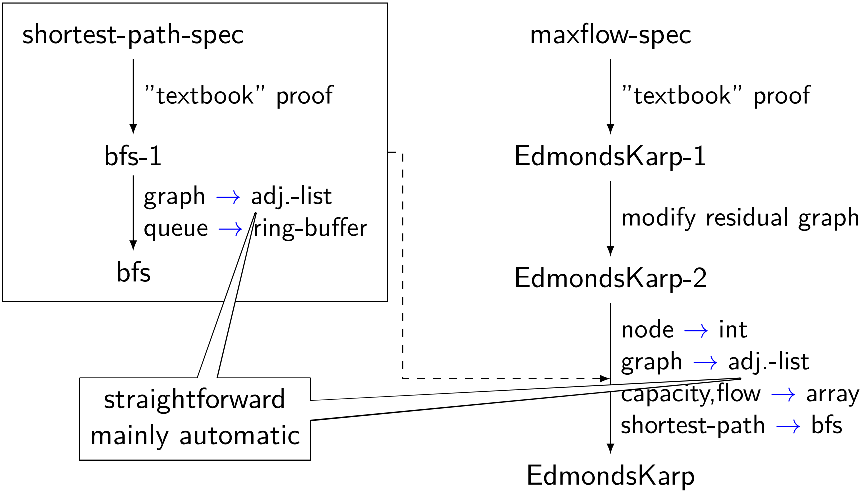
Refinement Architecture (simplified)



Refinement Architecture (simplified)



Refinement Architecture (simplified)



The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
    - Verification Condition Generator
    - Collection Framework
    - (Semi)automatic data refinement

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights
  - GRAT UNSAT certification toolchain
    - formally verified
    - faster than (verified and unverified) competitors

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights
  - GRAT UNSAT certification toolchain
    - formally verified
    - faster than (verified and unverified) competitors
  - Introsort (on par with libstd++ std::sort)

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights
  - GRAT UNSAT certification toolchain
    - formally verified
    - faster than (verified and unverified) competitors
  - Introsort (on par with libstd++ std::sort)
  - Timed Automata model checker

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights
  - GRAT UNSAT certification toolchain
    - formally verified
    - faster than (verified and unverified) competitors
  - Introsort (on par with libstd++ std::sort)
  - Timed Automata model checker
  - CAVA LTL model checker

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included
  - Verification Condition Generator
  - Collection Framework
  - (Semi)automatic data refinement
- Some highlights
  - GRAT UNSAT certification toolchain
    - formally verified
    - faster than (verified and unverified) competitors
  - Introsort (on par with libstd++ std::sort)
  - Timed Automata model checker
  - CAVA LTL model checker
  - Network flow (Push-Relabel and Edmonds Karp)

Formalizing Refinement

- Formal model for algorithms
  - Require: nondeterminism, pointers/heap, (data) refinement
  - VCG, also for refinements
  - can get very complex!

Formalizing Refinement

- Formal model for algorithms
  - Require: nondeterminism, pointers/heap, (data) refinement
  - VCG, also for refinements
  - can get very complex!
- Current approach:
  1. NRES: nondeterminism error monad with refinement ... but no heap
     - simpler model, usable tools (e.g. VCG)
  2. HEAP: deterministic heap-error monad
     - separation logic based VCG

Formalizing Refinement

- Formal model for algorithms
  - Require: nondeterminism, pointers/heap, (data) refinement
  - VCG, also for refinements
  - can get very complex!
- Current approach:
  1. NRES: nondeterminism error monad with refinement ... but no heap
     - simpler model, usable tools (e.g. VCG)
  2. HEAP: deterministic heap-error monad
     - separation logic based VCG
- Automated transition from NRES to HEAP
  - automatic data refinement (e.g. integer by int64)
  - automatic placement on heap (e.g. list by array)
  - some in-bound proof obligations left to user

Code Generation

Translate HEAP to compilable code

Code Generation

Translate HEAP to compilable code

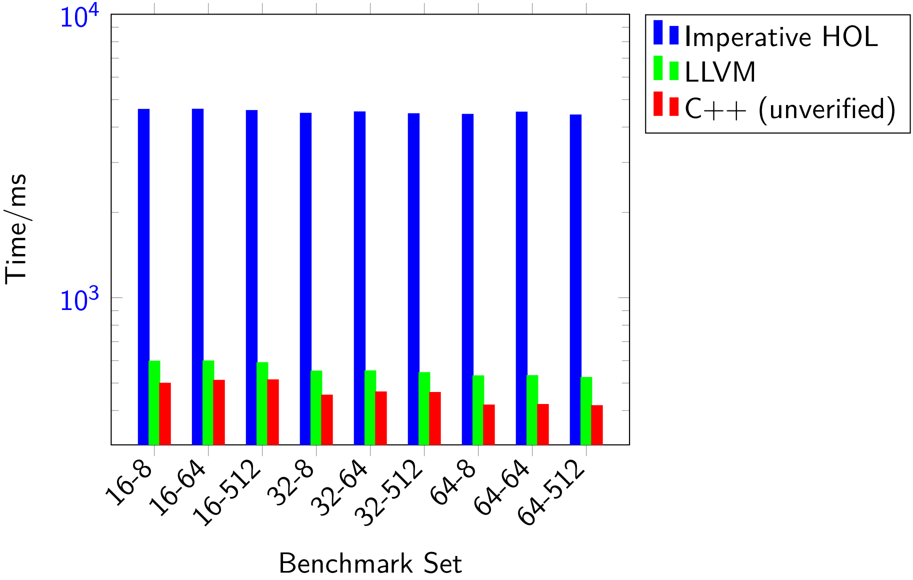
1. Imperative-HOL:
    - based on Isabelle's code generator
    - OCaml,SML,Haskell,Scala (using imp. features)
    - results cannot compete with optimized C/C++

Code Generation

Translate HEAP to compilable code

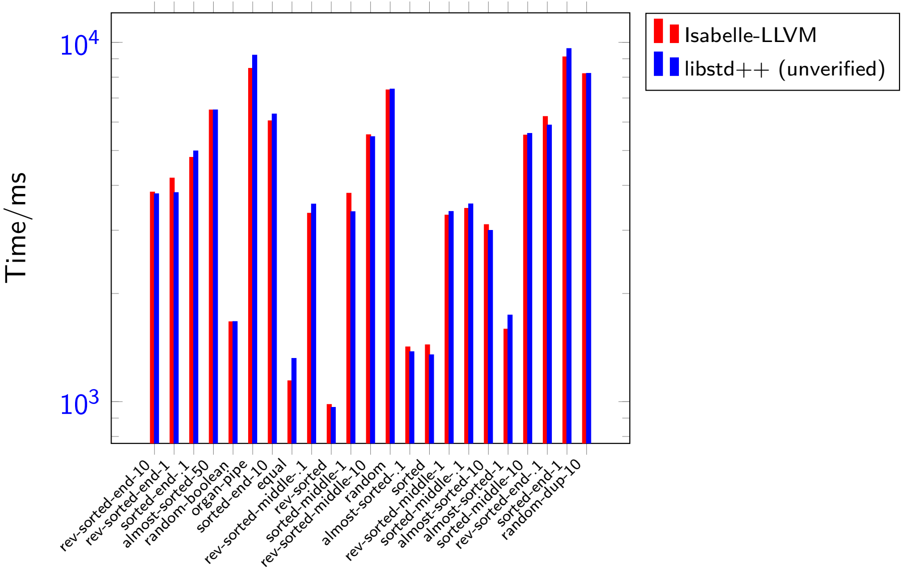
1. Imperative-HOL:
   - based on Isabelle's code generator
   - OCaml,SML,Haskell,Scala (using imp. features)
   - results cannot compete with optimized C/C++
2. NEW!: Isabelle-LLVM
   - shallow embedding of fragment of LLVM-IR
   - pretty-print to actual LLVM IR text
   - then use LLVM optimizer and compiler
   - faster programs
   - thinner (unverified) compilation layer

Knuth Morris Pratt



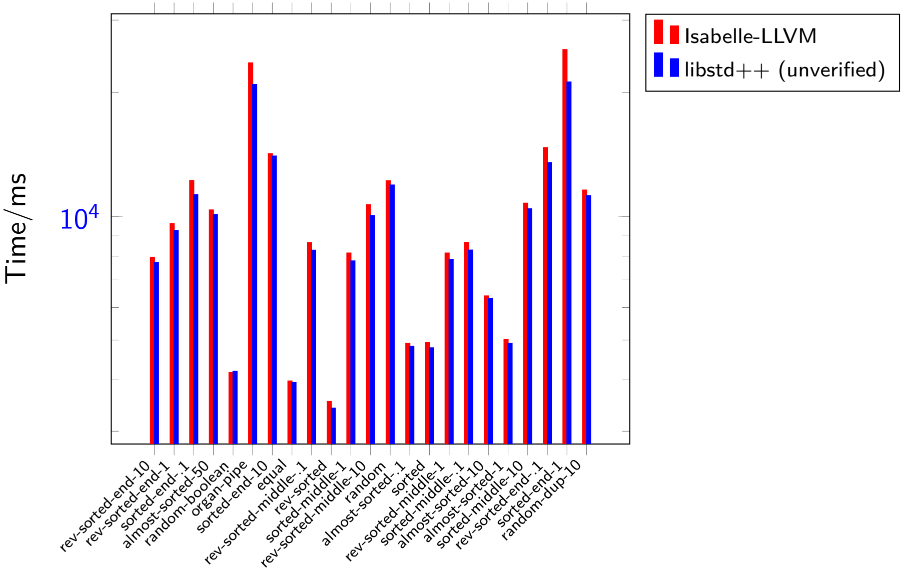
Execute *a-l* benchmark set from StringBench. Stop at first match.

Verified Introsort Algorithm



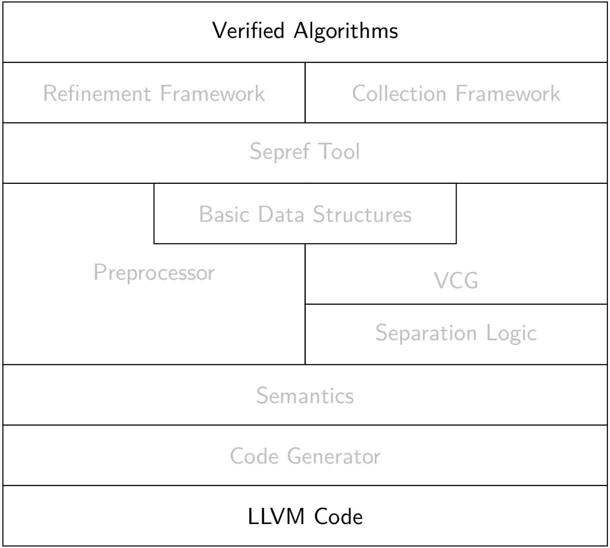
Sorting $100 \cdot 10^6$ `uint64`s on Intel Core i7-8665U CPU, 32GiB RAM.

Verified Introsort Algorithm

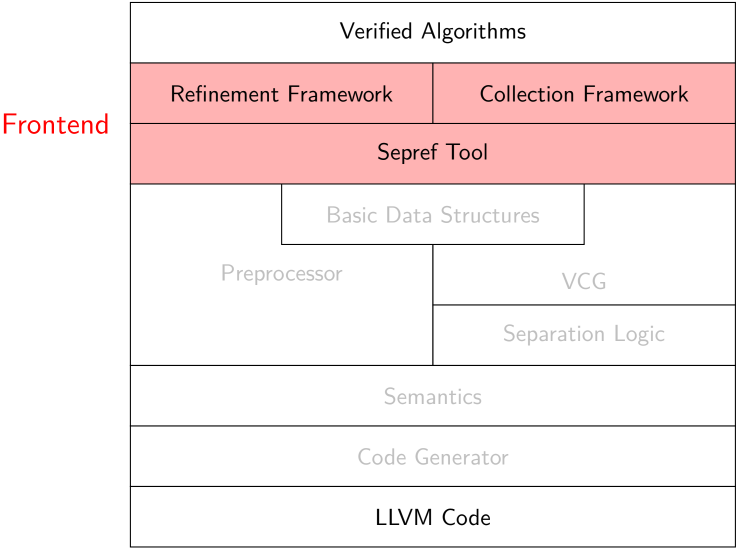


Sorting $100 \cdot 10^6$ `uint64s` on AMD Opteron 6176 24 core, 128GiB RAM.

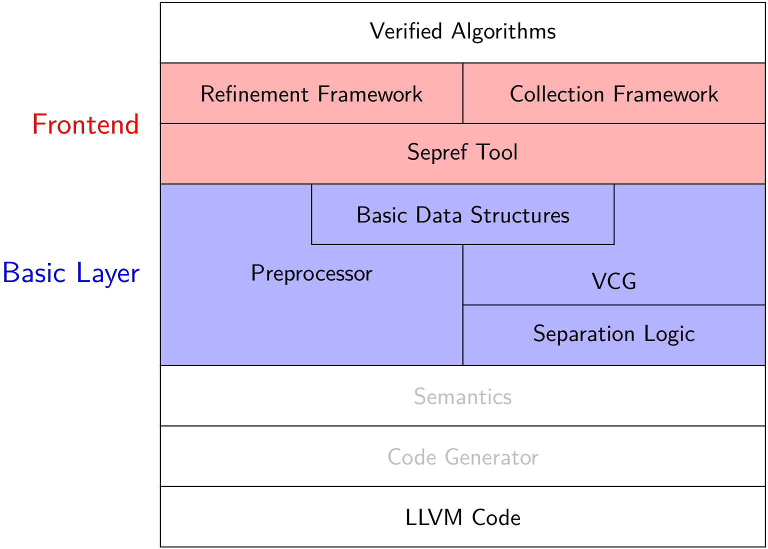
Isabelle-LLVM: Overview



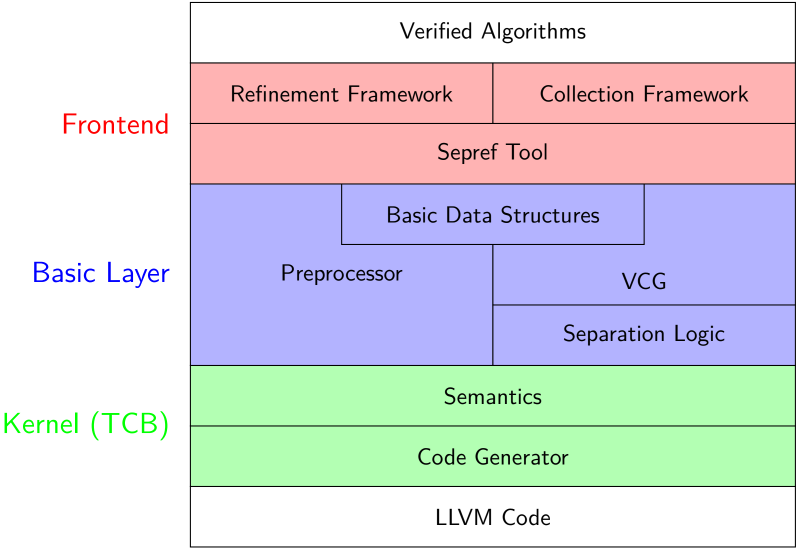
Isabelle-LLVM: Overview



Isabelle-LLVM: Overview



Isabelle-LLVM: Overview



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 $\Rightarrow$ 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

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

- 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

control flow (if, [optional: while])

standard instructions (ll_<opcode>)

arguments: variables and constants

monad: bind, return

```
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

control flow (if, [optional: while])

standard instructions (ll_<opcode>)

function calls

arguments: variables and constants

monad: bind, return

```
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

```
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×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

- 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←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) $\mapsto$ do {t←ll_add a b; ll_add t c}

  - tuples

    return (a,b) $\mapsto$ 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 $\{t \leftarrow$ ll_add a b; ll_add t c$\}$

  - tuples

    return (a,b) $\mapsto$ do $\{$ t$\leftarrow$ll_insert$_1$ init a; ll_insert$_2$ 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 $\{i_1, \ldots, i\_n\}$ 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}
    $\models$ {p$\mapsto$x} ll_load p {λr. r=x $*$ p$\mapsto$x}
    $\models$ {n$\neq$0} ll_malloc n {λp. range {0..<n} (λ_. init) p $*$ m_tag n p}
    $\models$ {range {0..<n} xs p $*$ m_tag n p} ll_free p {λ_. $\square$}

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}
    $\models$ {p$\mapsto$x} ll_load p {λr. r=x $*$ p$\mapsto$x}
    $\models$ {n$\neq$0} ll_malloc n {λp. range {0..<n} (λ_. init) p $*$ m_tag n p}
    $\models$ {range {0..<n} xs p $*$ m_tag n p} ll_free p {λ_. $\square$}

- 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 $\{i_1, \ldots, i\_n\}$ f p $=$ $(p+i_1)\mapsto(f\ i_1) * \ldots * (p+i\_n)\mapsto(f\ i\_n)$

  - rules for commands

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

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

Example: Proving Euclid's Algorithm

lemma

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

Example: Proving Euclid's Algorithm

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

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

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
```

sint_assn' sz — (mathematical) integers by *sz* bit integers
snat_assn' sz — natural numbers by *sz* bit integers
larray_assn' sz e — lists by arrays + *sz*-bit length, elements refined by *e*

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);
```

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
  - unify NRES and HEAP monads
  - generic Sepref (Imp-HOL, LLVM) $\times$ (nres, nres+time)

https://github.com/lammich/isabelle_llvm