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**

\[
p \leftarrow \text{shortest augmenting path}
\]

\[
f \leftarrow \text{AUGMENT}(g, f, p)
\]
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** Edmonds-Karp\((g, s, t)\)

\[
f \leftarrow \lambda(u, v). 0
\]

**while** exists augmenting path in \(g_f\) **do**

\[
p \leftarrow \text{shortest augmenting path}
\]

\[
f \leftarrow \text{AUGMENT}(g, f, p)
\]

**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
\[
p \leftarrow\text{shortest augmenting path}
\]
\[
f \leftarrow\text{AUGMENT}(g, f, p)
\]

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

\[
p \leftarrow \text{shortest augmenting path}
\]

\[
f \leftarrow \text{AUGMENT}(g, f, p)
\]

**Theorem**

Let \(\delta_f\) be the length of a shortest \(s, t\) - path in \(g_f\).

When augmenting with a shortest path,

- either \(\delta_f\) decreases
- \(\delta_f\) remains the same, and the number of edges in \(g_f\) that lay on a shortest path decreases.
Correctness

procedure \textsc{Edmonds-Karp}(g, s, t)
\[ f \leftarrow \lambda(u, v). 0 \]
\[ \textbf{while} \text{ exists augmenting path in } g_f \textbf{ do} \]
\[ p \leftarrow \text{shortest augmenting path} \]
\[ f \leftarrow \text{AUGMENT}(g, f, p) \]

Theorem
Let $\delta_f$ be the length of a shortest $s, t$ - path in $g_f$.
When augmenting with a shortest path,
\begin{itemize}
  \item either $\delta_f$ decreases
  \item $\delta_f$ remains the same, and the number of edges in $g_f$ that lay on a shortest path decreases.
\end{itemize}

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 $\delta_f$ be the length of a shortest $s$, $t$ - path in $g_f$. When augmenting with a shortest path,

- either $\delta_f$ decreases
- $\delta_f$ 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
\[
p \leftarrow \text{shortest augmenting path}
\]
\[
f \leftarrow \text{AUGMENT}(g, f, p)
\]

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 Edmonds-Karp\((g, s, t)\)
\[ f \leftarrow \lambda(u, v). 0 \]
\[ \text{while exists augmenting path in } g_f \text{ do} \]
\[ p \leftarrow \text{shortest augmenting path} \]
\[ f \leftarrow \text{AUGMENT}(g, f, p) \]

```
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

\textbf{procedure} 	extsc{Edmonds-Karp}(\(g, s, t\))
\[ f \leftarrow \lambda(u, v). 0 \]
\textbf{while} exists augmenting path in \(g_f\) do
\[ p \leftarrow \text{shortest augmenting path} \]
\[ f \leftarrow \text{AUGMENT}(g, f, p) \]
\textbf{return} \(f\).

```
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**

\[ p \leftarrow \text{shortest augmenting path} \]

\[ f \leftarrow \text{AUGMENT}(g, f, p) \]

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 Edmonds-Karp (g, s, t)
    f ← λ(u, v) · 0
    while exists augmenting path in gf do
        p ← shortest augmenting path
        f ← 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
\[ p \leftarrow \text{shortest augmenting path} \]
\[ f \leftarrow \text{AUGMENT}(g, f, p) \]

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:

  - 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).
      - $((s \uparrow, t \uparrow) \in \text{node 64}; (g \uparrow) \in \text{adjl} = \Rightarrow (\text{bfs } s \uparrow t \uparrow g \uparrow, \text{find shortest } s t g \uparrow) \in \text{array})$
    - Shortcut notation:
      - $(\text{bfs, find shortest}) \in \text{node 64} \rightarrow \text{node 64} \rightarrow \text{adjl} \rightarrow \text{array}$
  - Implementations used for different parts must fit together!
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

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).

Shortcut notation:
(\textit{bfs}, \textit{find shortest}) \in \text{node 64} \rightarrow \text{node 64} \rightarrow \text{adjl} \rightarrow \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

Implementations used for different parts must fit together!
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 \texttt{EdmondsKarp}
- Data refinement
  - BFS implementation uses adjacency lists. \texttt{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 (node64);
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

\[(s, s) \in \text{node64}; (t, t) \in \text{node64}; (g, g) \in \text{adjl} \implies (\text{bfs} \ s, t, g, \text{find_shortest} \ s, t, g) \in \text{array}\]
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).

\[(s_{\uparrow}, s) \in \text{node}_{64}; (t_{\uparrow}, t) \in \text{node}_{64}; (g_{\uparrow}, g) \in \text{adjl}\]
\[\implies (\text{bfs } s_{\uparrow} t_{\uparrow} g_{\uparrow}, \text{find\_shortest } s t g) \in \text{array}\]

Shortcut notation: \((\text{bfs, find\_shortest}) \in \text{node}_{64} \rightarrow \text{node}_{64} \rightarrow \text{adjl} \rightarrow \text{array}\)
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 (node64);
    - adjacency lists and graphs (adjl);
    - arrays and paths (array).

\[(s^{\dagger}, s) \in \text{node}_{64}; (t^{\dagger}, t) \in \text{node}_{64}; (g^{\dagger}, g) \in \text{adjl} \Rightarrow (\text{bfs } s^{\dagger} \ t^{\dagger} \ g^{\dagger}, \text{find_shortest } s \ t \ g) \in \text{array}\]

Shortcut notation: \((\text{bfs}, \text{find_shortest}) \in \text{node}_{64} \rightarrow \text{node}_{64} \rightarrow \text{adjl} \rightarrow \text{array}\)

- Implementations used for different parts must fit together!
Refinement Architecture (simplified)
Refinement Architecture (simplified)

shortest-path-spec
Refinement Architecture (simplified)

shortest-path-spec

\[ \text{bfs-1} \]
Refinement Architecture (simplified)

shortest-path-spec
  "textbook" proof
  bfs-1
Refinement Architecture (simplified)

shortest-path-spec
  "textbook" proof
  bfs-1
  bfs
Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list

queue → ring-buffer

bfs
Refinement Architecture (simplified)

shortest-path-spec
  
  "textbook" proof
    
    bfs-1
      
      graph → adj.-list
        
        queue → ring-buffer
          
          bfs
Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list

queue → ring-buffer

bfs

maxflow-spec

modify residual graph

node → int

graph → adj.-list

capacity,flow → array

shortest-path → bfs
Refinement Architecture (simplified)

shortest-path-spec
   "textbook" proof
     bfs-1
       graph → adj.-list
       queue → ring-buffer
         bfs

maxflow-spec
     EdmondsKarp-1

Refinement Architecture (simplified)

shortest-path-spec
  "textbook" proof
  bfs-1
  graph → adj.-list
  queue → ring-buffer
  bfs

maxflow-spec
  "textbook" proof
  EdmondsKarp-1
  capacity, flow → array
  shortest-path → bfs

Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list
queue → ring-buffer

bfs

maxflow-spec

"textbook" proof

EdmondsKarp-1

EdmondsKarp-2
Refinement Architecture (simplified)

**shortest-path-spec**
- "textbook" proof
- bfs-1
  - graph → adj-list
  - queue → ring-buffer
- bfs

**maxflow-spec**
- "textbook" proof
- EdmondsKarp-1
  - modify residual graph
- EdmondsKarp-2
Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list
queue → ring-buffer

bfs

maxflow-spec

"textbook" proof

EdmondsKarp-1

modify residual graph

EdmondsKarp-2

EdmondsKarp
Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list
queue → ring-buffer

bfs

maxflow-spec

"textbook" proof

EdmondsKarp-1

modify residual graph

EdmondsKarp-2

node → int
graph → adj.-list
capacity, flow → array
shortest-path → bfs

EdmondsKarp
Refinement Architecture (simplified)

- **shortest-path-spec**
  - "textbook" proof
  - bfs-1
    - graph → adj.-list
    - queue → ring-buffer
  - bfs

- **maxflow-spec**
  - "textbook" proof
  - EdmondsKarp-1
    - modify residual graph
  - EdmondsKarp-2
    - node → int
    - graph → adj.-list
    - capacity, flow → array
  - shortest-path → bfs

- **substantial ideas**
  - requires interactive proof

8 / 29
Refinement Architecture (simplified)

shortest-path-spec
  "textbook" proof
  bfs-1
    graph → adj.-list
    queue → ring-buffer
  bfs

maxflow-spec
  "textbook" proof
  EdmondsKarp-1
    modify residual graph
    EdmondsKarp-2
      node → int
      graph → adj.-list
      capacity, flow → array
      shortest-path → bfs
      EdmondsKarp

straightforward
mainly automatic
The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL

- 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)
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

- 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)
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

- 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)
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
Translating HEAP to compilable code

1. **Imperative-HOL**:
   - Based on Isabelle’s code generator
   - OCaml, SML, Haskell, Scala (using imperative 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 $100 \cdot 10^6$ uint64s 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

<table>
<thead>
<tr>
<th>Verified Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refinement Framework</td>
</tr>
<tr>
<td>Collection Framework</td>
</tr>
<tr>
<td>Sepref Tool</td>
</tr>
<tr>
<td>Basic Data Structures</td>
</tr>
<tr>
<td>Preprocessor</td>
</tr>
<tr>
<td>VCG</td>
</tr>
<tr>
<td>Separation Logic</td>
</tr>
<tr>
<td>Semantics</td>
</tr>
<tr>
<td>Code Generator</td>
</tr>
<tr>
<td>LLVM Code</td>
</tr>
</tbody>
</table>
Isabelle-LLVM: Overview

Frontend

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

Frontend
- Refinement Framework
- Collection Framework
- Sepref Tool
- Basic Data Structures
  - Preprocessor
  - VCG
  - Separation Logic

Basic Layer
- Semantics
- Code Generator
- LLVM Code

Verified Algorithms
Isabelle-LLVM: Overview

Frontend

Basic Layer

Kernel (TCB)

Verified Algorithms
Refinement Framework
Collection Framework

Sepref Tool

Basic Data Structures
Preprocessor
VCG
Separation Logic

Semantics

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

\[ \alpha \text{ llM} = \text{llM} \text{ (run: memory } \Rightarrow \alpha \text{ mres)} \]
\[ \alpha \text{ mres} = \text{NTERM} \mid \text{FAIL} \mid \text{SUCC } \alpha \text{ memory} \]
Basics

- LLVM operations described in state/error monad

\[ \alpha \text{lIM} = \text{lIM} \ (\text{run: memory} \Rightarrow \alpha \ mres) \]
\[ \alpha \ mres = \text{NTERM} | \text{FAIL} | \text{SUCC} \alpha \ memory \]

\[ \text{ll}_\text{udiv} :: \text{n word} \Rightarrow \text{n word} \Rightarrow \text{n word} \ \alpha \text{lIM} \]
\[ \text{ll}_\text{udiv} \ a \ b = \text{do} \ {\text{assert} \ (b \neq 0); \text{return} \ (a \ \text{div} \ b)} \]
Basics

• LLVM operations described in state/error monad

$$\alpha \text{llM} = \text{llM} \ (\text{run}: \text{memory} \Rightarrow \alpha \text{mres})$$

$$\alpha \text{mres} = \text{NTERM} \mid \text{FAIL} \mid \text{SUCC} \ \alpha \ \text{memory}$$

$$\text{ll}_{\text{udiv}} :: \text{n word} \Rightarrow \text{n word} \Rightarrow \text{n word} \ \alpha \text{llM}$$

$$\text{ll}_{\text{udiv}} \ a \ b = \text{do} \ {\{ \ \text{assert} \ (b \neq 0); \ \text{return} \ (a \ \text{div} \ b) \ \}}$$

$$\text{llc}_{\text{if}} \ b \ t \ e = \text{if} \ b \neq 0 \ \text{then} \ t \ \text{else} \ e$$
Basics

• LLVM operations described in state/error monad

\[ \alpha \text{llM} = \text{llM} (\text{run}: \text{memory} \Rightarrow \alpha \text{mres}) \]
\[ \alpha \text{mres} = \text{NTERM} | \text{FAIL} | \text{SUCC} \alpha \text{memory} \]

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

\[ \text{llc_if} b \text{ t e} = \text{if} \ b \neq 0 \ \text{then} \ t \ \text{else} \ e \]

• Recursion via fixed-point

\[ \text{llc_while} b \ f \ s_0 = \text{fixp} (\lambda W \ s. \]
\[ \ \text{do} \{ \]
\[ \ \ \text{ctd} \leftarrow b \ s; \]
\[ \ \ \ \text{if} \ ctd \neq 0 \ \text{then} \ \text{do} \{s \leftarrow f \ s; W \ s\} \ \text{else} \ \text{return} \ s\]
\[ \} \ s_0 \]
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 lIM

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

fib :: 64 word \rightarrow 64 word \texttt{llM}

\begin{verbatim}
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
  }))
\end{verbatim}
Shallow Embedding

fib:: 64 word ⇒ 64 word lIm

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

fib :: 64 word ⇒ 64 word III M
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 IIM

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

```
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 \Rightarrow 64 word \text{ IllM}

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_\langle opcode\rangle)

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**

compiling control flow + pretty printing

fib:: 64 word $\Rightarrow$ 64 word llM

```plaintext
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t (return n)
}
```

```plaintext
define i64 @fib(i64 %x) {
  start:
  %t = icmp ule i64 %x, 1
  br i1 %t, label %then, label %else
  then:
  br label %ctd_if
  %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.

\[
\begin{align*}
\text{memory} &= \text{block list} \quad \text{block} = \text{val list option} \\
\text{val} &= n \ \text{word} \mid \text{ptr} \mid \text{val} \times \text{val} \\
\text{rptr} &= \text{NULL} \mid \text{ADDR} \ \text{nat} \ \text{nat} \ (\text{dir list}) \quad \text{dir} = \text{FST} \mid \text{SND}
\end{align*}
\]

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

- Inspired by CompCert v1. But with structured values.

  \[
  \text{memory} = \text{block list} \quad \text{block} = \text{val list} \oplus \text{option}
  \]

  \[
  \text{val} = n \ \text{word} \mid \text{ptr} \mid \text{val} \times \text{val}
  \]

  \[
  \text{rptr} = \text{NULL} \mid \text{ADDR} \ \text{nat} \ \text{nat} \ \text{(dir list)} \quad \text{dir} = \text{FST} \mid \text{SND}
  \]

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

- Typeclass `llvm_rep`: shallow to deep embedding

  \[
  \text{to_val :: ‘a ⇒ val}
  \]

  \[
  \text{from_val :: val ⇒ ‘a}
  \]

  \[
  \text{init :: ‘a – Zero initializer}
  \]
Memory Model

- Inspired by CompCert v1. But with structured values.

  \[
  \text{memory} = \text{block list} \quad \text{block} = \text{val list option} \\
  \text{val} = n \ \text{word} \mid \text{ptr} \mid \text{val} \times \text{val} \\
  \text{rptr} = \text{NULL} \mid \text{ADDR nat nat (dir list)} \quad \text{dir} = \text{FST} \mid \text{SND}
  \]

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

- Typeclass \texttt{llvm_rep}: shallow to deep embedding

  \[
  \text{to_val} :: \ 'a \Rightarrow \text{val} \\
  \text{from_val} :: \text{val} \Rightarrow \ 'a \\
  \text{init} :: \ 'a \rightarrow \text{Zero initializer}
  \]

- Shallow pointers carry phantom type

  \[
  'a \ \text{ptr} = \text{PTR rptr}
  \]
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

```haskell
c 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

\[
\text{return } ((a+b)+c) \mapsto \text{do } \{t \leftarrow \text{ll}_\text{add} a \ b; \text{ll}_\text{add} t \ c\}
\]
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

\[
\text{return } (\text{return } (a+b)+c) \mapsto \text{do } \{ t \leftarrow \text{ll_add } a \text{ b}; \text{ll_add } t \text{ c} \}
\]

- tuples

\[
\text{return } (a,b) \mapsto \text{do } \{ t \leftarrow \text{ll_insert}_1 \text{ init } a; \text{ll_insert}_2 t \text{ b } \}
\]
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

  \[
  \text{return } ((a+b)+c) \mapsto \text{do } \{ t \leftarrow \text{ll_add } a \; b; \; \text{ll_add } t \; c \} 
  \]

- tuples

  \[
  \text{return } (a,b) \mapsto \text{do } \{ t \leftarrow \text{ll_insert}_1 \; \text{init } a; \; \text{ll_insert}_2 \; t \; b \} 
  \]

- Define recursive functions for fixed points
Example: Preprocessing Euclid’s Algorithm

euclid :: 64 word ⇒ 64 word ⇒ 64 word

\[
\text{euclid } a \ b = \text{do } \{
(a, b) \leftarrow \text{l1c \_while}
\quad (\lambda(a, b) \Rightarrow \text{l1 \_cmp } (a \neq b))
\quad (\lambda(a, b) \Rightarrow \text{if } (a \leq b) \text{ then return } (a, b - a) \text{ else return } (a - b, b))
\quad (a, b);
\text{return a } \}
\]
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 euclid₀ 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

\[ \alpha :: \text{memory} \rightarrow \text{amemory} :: \text{sep_algebra} \]
\[ wp \ c \ Q \ s = \exists r \ s'. \ \text{run} \ c \ s = \text{SUCC} \ r \ s' \land Q \ r (\alpha \ s') \]
\[ |\{P\} c \{Q\} = \forall F \ s. \ (P \ast F) (\alpha \ s) \rightarrow wp \ c (\lambda r \ s'. \ (Q \ r \ast F) s') s \]
Reasoning about LLVM Programs

- Separation Logic
- Hoare-triples

\[ \alpha :: \text{memory} \rightarrow \text{amemory} :: \text{sep_algebra} \]
\[ \text{wp} \ c \ Q \ s = \exists r \ s'. \ \text{run} \ c \ s = \text{SUCC} \ r \ s' \land Q \ r \ (\alpha \ s') \]
\[ \models \{P\} \ c \ \{Q\} = \forall F \ s. \ (P \ast F) \ (\alpha \ s) \xrightarrow{} \text{wp} \ c \ (\lambda r \ s'. \ (Q \ r \ast F) \ s') s \]

- memory primitives

  \[ p \mapsto x - p \text{ points to value } x \]
  \[ m_{\text{tag}} \ n \ p - \text{ownership of block (not its contents)} \]

  range \ \{i_1, \ldots, i_n\} \ f \ p = (p+i_1)\mapsto(f \ i_1) \ast \ldots \ast (p+i_n)\mapsto(f \ i_n) \]
Reasoning about LLVM Programs

- Separation Logic
- Hoare-triples

\[ \alpha :: \text{memory} \to \text{amemory} :: \text{sep_algebra} \]

\[ \wp c \ Q \ s = \exists r \ s'. \ \text{run} \ c \ s = \text{SUCC} \ r \ s' \land Q \ r \ (\alpha \ s') \]

\[ \models \{P\} \ c \ \{Q\} = \forall F \ s. \ (P * F) \ (\alpha \ s) \to \wp c \ (\lambda r \ s'. \ (Q \ r * F) \ s') \ s \]

- memory primitives

\( p \mapsto x \) – p points to value x

\( m\_\text{tag} \ n \ p \) – ownership of block (not its contents)

\[ \text{range} \ \{i_1, \ldots, i_n\} \ f \ p = (p+i_1) \mapsto (f \ i_1) \ast \ldots \ast (p+i_n) \mapsto (f \ i_n) \]

- rules for commands

\( b \neq 0 \quad \implies \quad \models \{\square\} \ \text{ll\_udiv} \ a \ b \ \{\lambda r. \ r = a \ \text{div} \ b\} \]

\[ \models \{p \mapsto x\} \ \text{ll\_load} \ p \ \{\lambda r. \ r = x \ast p \mapsto x\} \]

\[ \models \{n \neq 0\} \ \text{ll\_malloc} \ n \ \{\lambda p. \ \text{range} \ \{0..<n\} \ (\lambda. \ \text{init}) \ p * m\_\text{tag} \ n \ p\} \]

\[ \models \{\text{range} \ \{0..<n\} \ \times s \ p * m\_\text{tag} \ n \ p\} \ \text{ll\_free} \ p \ \{\lambda. \ \square\} \]
Reasoning about LLVM Programs

- Separation Logic
  - Hoare-triples
    \[ \alpha :: \text{memory} \rightarrow \text{amemory} :: \text{sep\_algebra} \]
    \[
    \text{wp} \ c \ Q \ s = \exists r \ s'. \ \text{run} \ c \ s = \text{SUCC} \ r \ s' \land Q \ r (\alpha \ s')
    \]
    \[
    \models \ \{P\} \ c \ \{Q\} = \forall F \ s. \ (P\ast F) (\alpha \ s) \rightarrow \text{wp} \ c (\lambda r \ s'. \ (Q \ r \ast F) s') \ s
    \]
- Memory primitives
  - \( p \mapsto x \) – \( p \) points to value \( x \)
  - \( \text{m\_tag} \ n \ p \) – ownership of block (not its contents)
  - \( \text{range} \ \{i_1, \ldots, i_n\} \ f \ p = (p+i_1) \mapsto (f \ i_1) \ast \ldots \ast (p+i_n) \mapsto (f \ i_n) \)
- Rules for commands
  - \( b \neq 0 \implies \models \{\square\} \text{ll\_udiv} \ a \ b \ \{\lambda r. \ r = a \ div \ b\} \)
  - \( \models \{p \mapsto x\} \text{ll\_load} \ p \ \{\lambda r. \ r=x \ast p \mapsto x\} \)
  - \( \models \{n \neq 0\} \text{ll\_malloc} \ n \ \{\lambda p. \ \text{range} \ \{0..<n\} \ (\lambda_. \ \text{init}) \ p \ast \text{m\_tag} \ n \ p\} \)
  - \( \models \{\text{range} \ \{0..<n\} \times s \ p \ast \text{m\_tag} \ n \ p\} \text{ll\_free} \ p \ \{\lambda_. \ \square\} \)

- Automation: VCG, frame inference, heuristics to discharge VC}s
Reasoning about LLVM Programs

• Separation Logic
  • Hoare-triples

\[ \alpha :: \text{memory} \rightarrow \text{amemory} :: \text{sep_algebra} \]
\[ \text{wp} \ c \ Q \ s = \exists r \ s'. \ \text{run} \ c \ s = \text{SUCC} \ r \ s' \land Q \ r \ (\alpha \ s') \]
\[ \models \{P\} \ c \ \{Q\} = \forall F \ s. \ (P \ast F) (\alpha \ s) \longrightarrow \text{wp} \ c \ (\lambda r \ s'. \ (Q \ r \ast F) \ s') \ s \]

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

\[ \text{range} \ \{i_1, \ldots, i_n\} \ f \ p = (p+i_1) \mapsto (f \ i_1) \ast \ldots \ast (p+i_n) \mapsto (f \ i_n) \]

• rules for commands

b \neq 0 \implies \models \{\square\} \ \text{ll_udiv} \ a \ b \ \{\lambda r. \ r = a \ \text{div} \ b\}
\models \{p \mapsto x\} \ \text{ll_load} \ p \ \{\lambda r. \ r=x \ast p \mapsto x\}
\models \{n \neq 0\} \ \text{llMalloc} \ n \ \{\lambda p. \ \text{range} \ \{0..<n\} \ (\lambda_. \ \text{init}) \ p \ast m\_tag \ n \ p\}
\models \{\text{range} \ \{0..<n\} \ xs \ p \ast m\_tag \ n \ p\} \ \text{ll_free} \ p \ \{\lambda_. \ \square\} \]

• Automation: VCG, frame inference, heuristics to discharge VC

• Basic Data Structures: signed/unsigned integers, Booleans, arrays

Basic Layer
Example: Proving Euclid’s Algorithm

lemma
\[ \vdash \{ \text{uint}_64 \ a \ \hat{\times} \ \text{uint}_64 \ b \ b \ \hat{\times} \ 0 < a \times 0 < b \} \ \text{euclid} \ a \ b \ \{ \lambda r \ . \ \text{uint}_64 \ (\gcd \ a \ b) \ r \} \]
Example: Proving Euclid’s Algorithm

lemma
\( \models \{ \text{uint}_64 \ a \ a^\dagger \ast \ \text{uint}_64 \ b \ b^\dagger \ast 0 < a \ast 0 < b \} \ \text{euclid} \ a^\dagger \ b^\dagger \ \{ \lambda r^\dagger. \ \text{uint}_64 \ (\gcd \ a \ b) \ r^\dagger \} \)

unfolding euclid_def
apply (rewrite annotate_llc_while[where l = \ldots \ and R = \text{measure nat}])
Example: Proving Euclid’s Algorithm

lemma
\[ \models \{ \text{uint}_64 \ a \ a_{\dagger} \ast \text{uint}_64 \ b \ b_{\dagger} \ast \ 0 < a \ast \ 0 < b \} \ \text{euclid} \ a_{\dagger} \ b_{\dagger} \ \{ \lambda r_{\dagger}. \ \text{uint}_64 \ (\text{gcd} \ a \ b) \ r_{\dagger} \} \]

unfolding euclid_def
apply (rewrite annotate_llc_while[where l = ... and R = measure nat])
apply (vcg; clarsimp?)
Example: Proving Euclid’s Algorithm

lemma
\[ \{ \text{uint}_{64} \ a \ a^\top \,* \, \text{uint}_{64} \ b \ b^\top \,* \, \,0<a \,* \,0<b \} \, \text{euclid} \ a^\top \,b^\top \{ \lambda r^\top. \, \text{uint}_{64} \, (\gcd \ a \ b) \, r^\top \} \]

unfolding euclid_def
apply (rewrite annotate_llc_while[where l = \ldots \text{ and } R = \text{measure nat}])
apply (vcg; clarsimp?)

Subgoals:
1. \( \forall x \, y. \, [\, \gcd \ x \, y \, = \, \gcd \ a \ b ; \, x \neq y ; \, x \leq y; \, \ldots \, ] \implies \gcd \ x \, (y - x) \, = \, \gcd \ a \ b \)
2. \( \forall x \, y. \, [\, \gcd \ x \, y \, = \, \gcd \ a \ b ; \, \neg \, x \leq y; \, \ldots \, ] \implies \gcd \ (x \, - \, y) \, y \, = \, \gcd \ a \ b \)
Example: Proving Euclid’s Algorithm

\textbf{Basic Layer}

\textbf{lemma}
| \{ \text{uint}64 \ a \ a^\dagger \ \ast \ \text{uint}64 \ b \ b^\dagger \ \ast \ 0 < a \ \ast \ 0 < b \} \ \text{euclid} \ a^\dagger \ b^\dagger \ \{ \lambda r^\dagger. \ \text{uint}64 \ (\gcd \ a \ b) \ r^\dagger \} \\

\text{unfolding euclid_def}
apply (\text{rewrite annotate\_llc\_while[where I = \ldots \ \text{and} \ R = \text{measure nat}])}

apply (\text{vcg; clarsimp?})

Subgoals:
1. \( \forall x, y. [\gcd x y = \gcd a b; x \neq y; x \leq y; \ldots ] \implies \gcd x (y - x) = \gcd a b \)
2. \( \forall x, y. [\gcd x y = \gcd a b; \neg x \leq y; \ldots ] \implies \gcd (x - y) y = \gcd a b \)

by (\text{simp\_all add: gcd\_diff1 gcd\_diff1'})
Automatic Refinement

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

Frontend

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)
• ...
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)
    - ...

Frontend
Example: Binary Search

definition bin_search xs x = do {
  (l,h) ← WHILE (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 href_with_rdomI, annot_snat_const TYPE(size_t))
by sepref
```
Example: Binary Search — Refinement

\texttt{sepref\_def} \texttt{bin\_search\_impl} \texttt{is} uncurry \texttt{bin\_search}\
\>:: (larray\_assn\' \texttt{TYPE} (\texttt{size\_t}) (sint\_assn\' \texttt{TYPE} (elem\_t)))^k
\hspace{1cm} \ast (sint\_assn\' \texttt{TYPE} (elem\_t))^k
\hspace{1cm} \rightarrow \texttt{snat\_assn\' \texttt{TYPE} (\texttt{size\_t})}
\texttt{unfolding} \texttt{bin\_search\_def}
\texttt{apply} (\texttt{rule hfref\_with\_rdomI, annot\_snat\_const \texttt{TYPE} (\texttt{size\_t})})
\texttt{by} \texttt{sepref}

\texttt{sint\_assn\' \texttt{sz} — (mathematical) integers by \texttt{sz} bit integers}
\texttt{snat\_assn\' \texttt{sz} — natural numbers by \texttt{sz} bit integers}
\texttt{larray\_assn\' \texttt{sz e} — lists by arrays + \texttt{sz}-bit length, elements refined by \texttt{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 href_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;

defines
Example: Binary Search — Generated Code

Frontend

Produces LLVM code and header file:

```c
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