The Isabelle Refinement Framework

Peter Lammich

The University of Manchester

March 2020
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 \hspace{1cm} \(s, t\): source, target \hspace{1cm} \(g_f\): 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

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

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

**Theorem (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 lie 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**

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

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

**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 lie 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**

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

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

**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 lie on a shortest path decreases.

**Proof.**

two more textbook pages.

using lemmas about graphs and shortest paths.

\[ \square \]
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 $g_f$
    $p \leftarrow$ 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.
textbook proof typically covers abstract algorithm. But this is quite far from implementation. Still missing:

```cpp
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;
}
```
procedure Edmonds-Karp \((g, s, t)\)

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

while exists augmenting path in \(g_f\) do

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

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

\[ \text{int edmonds_karp(int s, int t)} \{ \]

\[ \text{int flow = 0;} \]

\[ \text{vector<int> parent(n);} \]

\[ \text{int new_flow;} \]

\[ \text{while (new_flow = bfs(s, t, parent)) \{} \]

\[ \text{flow += new_flow;} \]

\[ \text{int cur = t;} \]

\[ \text{while (cur != s) \{} \]

\[ \text{int prev = parent[cur];} \]

\[ \text{capacity[prev][cur] -= new_flow;} \]

\[ \text{capacity[cur][prev] += new_flow;} \]

\[ \text{cur = prev;} \]

\[ \}\]}

\[ \text{return flow;} \]

} 


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 $g_f$

$p \leftarrow$ shortest augmenting path

$f \leftarrow$ AUGMENT($g$, $f$, $p$)

\[
\text{int edmonds_karp(int s, int t) \{}
\]

\[
\text{int flow = 0;}
\]

\[
\text{vector<int> parent(n);}
\]

\[
\text{int new_flow;}
\]

\[
\text{while (new_flow = bfs(s, t, parent)) \{}
\]

\[
\text{flow += new_flow;}
\]

\[
\text{int cur = t;}
\]

\[
\text{while (cur != s) \{}
\]

\[
\text{int prev = parent[cur];}
\]

\[
\text{capacity[prev][cur] -= new_flow;}
\]

\[
\text{capacity[cur][prev] += new_flow;}
\]

\[
\text{cur = prev;}
\]

\[
\}
\]

\[
\}
\]

\[
\text{return flow;}
\]

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

```cpp
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 \(g_f\)

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

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

\[
\text{int edmonds_karp(int s, int t)} \{
\]

\[
\text{int flow = 0;}
\]

\[
\text{vector<int> parent(n);}
\]

\[
\text{int new_flow;}
\]

\[
\text{while (new_flow = bfs(s, t, parent))} \{
\]

\[
\text{flow += new_flow;}
\]

\[
\text{int cur = t;}
\]

\[
\text{while (cur != s)} \{
\]

\[
\text{int prev = parent[cur];}
\]

\[
\text{capacity[prev][cur] -= new_flow;}
\]

\[
\text{capacity[cur][prev] += new_flow;}
\]

\[
\text{cur = prev;}
\]

\[
\}
\]

\[
\}
\]

\[
\text{return flow;}
\]

**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).

(\(s, s\) \(\in\) node 64; \(t, t\) \(\in\) node 64; \(g, g\) \(\in\) adjl)
\(\Rightarrow\) \((bfs s t g, find shortest s t g) \in array\)

Shortcut notation:
\((bfs, find shortest) \in node 64 \rightarrow node 64 \rightarrow adjl \rightarrow array\)

• Implementations used for different parts must fit together!
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

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

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

Shortcut notation:
- \((\text{bfs}, \text{find shortest}) \in \text{node} \rightarrow \text{node} \rightarrow \text{adjl} \rightarrow \text{array}\)
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 \texttt{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 \texttt{EdmondsKarp}
- Data refinement
  - BFS implementation uses adjacency lists. \texttt{EdmondsKarp} used abstract graphs.
  - refinement relations between
    - nodes and int64s (\texttt{node}_{64});
    - adjacency lists and graphs (\texttt{adjl});
    - arrays and paths (\texttt{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).

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

$$(s_{\uparrow},s) \in \text{node}_{64}; (t_{\uparrow},t) \in \text{node}_{64}; (g_{\uparrow},g) \in \text{adjl} \implies (\text{bfs } s_{\uparrow} t_{\uparrow} g_{\uparrow}, \text{find_shortest } s \ t \ g) \in \text{array}$$

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).

\[(s_t, s) \in \text{node}_64; (t_t, t) \in \text{node}_64; (g_t, g) \in \text{adjl} \implies (\text{bfs } s_t, t_t, g_t, \text{find_shortest } s, t, g) \in \text{array}\]

Shortcut notation: \((\text{bfs, find_shortest}) \in \text{node}_64 \rightarrow \text{node}_64 \rightarrow \text{adjl} \rightarrow \text{array}\)

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

shortest-path-spec
Refinement Architecture (simplified)

shortest-path-spec

→ 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

EdmondsKarp
   "textbook" proof
      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

EdmondsKarp
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

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
    EdmondsKarp

straightforward
mainly automatic
straightforward
mainly automatic
straightforward
mainly automatic
straightforward
mainly automatic
straightforward
mainly automatic

8/17
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
- GRAT UNSAT certification toolchain
  - formally verified
  - faster than (verified and unverified) competitors
- Introsort (on par with libstd++ \texttt{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
The Isabelle Refinement Framework

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

- 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
  - 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!

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
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

Benchmark Set

Execute a-l benchmark set from StringBench. Stop at first match.
Verified Sorting Algorithms: Introsort

![Bar chart showing comparison between Isabelle-LLVM and libstd++ (unverified) in sorting performance. The chart includes various test cases such as rev-sorted, end-10, sorted-end-1, random, boolean, organ-pipe, sorted-end-10, equal, rev-sorted-middle-1, rev-sorted-end-middle-10, random, sorted-middle-1, almost-sorted-end-1, rev-sorted-middle-10, almost-sorted-middle-1, sorted-end-middle-1, rev-sorted-end-middle-10, random-dup-10. The x-axis represents the test cases, and the y-axis represents the time in milliseconds. The chart shows the performance of sorting 100,000,000 uint64s on Intel Core i7-8665U CPU, 32GiB RAM]
Verified Sorting Algorithms: Pdqsort

Time/ms

Isabelle-LLVM
Boost (unverified)

Sorting $100 \cdot 10^6$ uint64s on Intel Core i7-8665U CPU, 32GiB RAM
Current (near Future) Projects

- Framework
- Scalable Sefref Tool
- Nested Containers
- Nice input language
- Support for Nres+Time
- Applications
  - SAT
  - Verified SAT Solver
  - Verified drat-trim
  - QBF certificate checking
- Graphs: Efficient Blossom Algorithm Implementation
- Sorting:
  - Branch-aware partitioning
  - Stable sorts
Current (near Future) Projects

- Framework
  - Scalable Sepref Tool

- Applications
  - SAT
    - Verified SAT Solver
    - Verified drat-trim
  - QBF certificate checking

- Graphs: Efficient Blossom Algorithm Implementation
- Sorting:
  - Branch-aware partitioning
  - Stable sorts
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers

• Applications
  • SAT
    • Verified SAT Solver
    • Verified drat-trim
  • QBF certificate checking

• Graphs: Efficient Blossom Algorithm Implementation
• Sorting:
  • Branch-aware partitioning
  • Stable sorts
Current (near Future) Projects

- Framework
  - Scalable Sepref Tool
  - Nested Containers
  - Nice input language

- Support for Nres+Time

- Applications
  - SAT
    - Verified SAT Solver
    - Verified drat-trim
  - QBF certificate checking

- Graphs: Efficient Blossom Algorithm Implementation

- Sorting:
  - Branch-aware partitioning
  - Stable sorts
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers
  • Nice input language
  • Support for Nres+Time

• Applications
  • SAT
  • Verified SAT Solver
  • Verified drat-trim
  • QBF certificate checking
  • Graphs: Efficient Blossom Algorithm Implementation
  • Sorting:
    • Branch-aware partitioning
    • Stable sorts
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers
  • Nice input language
  • Support for Nres+Time

• Applications
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers
  • Nice input language
  • Support for Nres+Time

• Applications
  • SAT
    • Verified SAT Solver
    • Verified drat-trim
    • QBF certificate checking
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers
  • Nice input language
  • Support for Nres+Time

• Applications
  • SAT
    • Verified SAT Solver
    • Verified drat-trim
    • QBF certificate checking
  • Graphs: Efficient Blossom Algorithm Implementation
Current (near Future) Projects

- Framework
  - Scalable Sepref Tool
  - Nested Containers
  - Nice input language
  - Support for Nres+Time

- Applications
  - SAT
    - Verified SAT Solver
    - Verified drat-trim
    - QBF certificate checking
  - Graphs: Efficient Blossom Algorithm Implementation
  - Sorting:
    - Branch-aware partitioning
    - Stable sorts
Current (near Future) Projects

• Framework
  • Scalable Sepref Tool
  • Nested Containers
  • Nice input language
  • Support for Nres+Time

• Applications
  • SAT
    • Verified SAT Solver
    • Verified drat-trim
    • QBF certificate checking
  • Graphs: Efficient Blossom Algorithm Implementation
  • Sorting:
    • Branch-aware partitioning
    • Stable sorts

• ...
Sepref Tool

- Synthesize imperative program from functional
  - sep-logic assertion relating concrete with abstract variables
• Synthesize imperative program from functional
  • sep-logic assertion relating concrete with abstract variables

```
f (l :: int list) {
  (int)set S = {}
  int c=0
  for (int i=0; i<l.length; ++i) {
    t1 = l[i]
    if (t1 ∉ S) {
      ∗: assert (c<l.length)
      ++c
      S={t1} ∪ S
    }
  }
}
```
• Synthesize imperative program from functional
  • sep-logic assertion relating concrete with abstract variables

**Sepref Tool**

```plaintext
f (l :: int list) {
  (int)set S = {}
  int c=0
  for (int i=0; i<|l|; ++i) {
    t1 = l[i]
    if (t1 ∉ S) {
      ∗: assert (c <|l|)
      ++c
      S={t1} ∪ S
    }
  }
}

f (l′ :: int64 array) {
  hashmap S′ = hm_empty()
  int64 c′=0
  for (int64 i′=0; i′<|l′|; ++i′) {
    t1′ = l[i′]
    if (!hm_member t1′ S′) {
      ∗:
      ++c′
      S′=hm_insert t1′ S′
    }
  }
}
```

...
Sepref Tool

- Synthesize imperative program from functional
  - sep-logic assertion relating concrete with abstract variables

\[
\begin{align*}
\text{f (l :: int list) \{} & \text{f (l' :: int64 array) \{} \\
\text{  (int) set } S = \\{\} & \text{ hashmap } S' = \text{hm_empty()} \\
\text{  int } c=0 & \text{ int64 } c'=0 \\
\text{  for (int } i=0; i<|l|; ++i) \{ & \text{ for (int64 } i'=0; i'|<|l'|; ++i') \{ \\
\text{    t_1 = l[i] } & \text{ t_1' = l'[i'] } \\
\text{    if (t_1 \notin S) \{ } & \text{ if (\neg \text{hm_member } t_1'S') \{ } \\
\text{      \*: assert (c<|l|) } & \text{      \*: } \\
\text{      ++c } & \text{      ++c' } \\
\text{      S=\{t_1\} \cup S } & \text{      S'=\text{hm_insert } t_1'S' } \\
\text{    \} } } } & \text{ } } free \ S' \} \\
\text{  \} } } \\
\text{\} } } \\
\end{align*}
\]

At *: \text{array } i64 \text{ l l' } * \text{hm } i64 \text{ S S'} * \text{...}
Nested Containers

Hoare-Rule for array-index:

\[
\{ \text{array A l l'} \ast i64 i i' \ast i < \|l\| } \ r' = l'[i] \ \{ \text{array A l l'} \ast i64 i i' \ast A (l[i]) r' \}
\]

where

\[
\text{array A l p} = \exists l'. p + 0 \mapsto l'[0] \ast \ldots \ast p + n \mapsto l'[n] \\
* A l[0] l'[0] \ast \ldots \ast A l[n] l'[n]
\]
Nested Containers

Hoare-Rule for array-index:

\[
\begin{align*}
\{&\text{array } A \mid l' \ast i64 \ast i \ast i < |l|\} \quad r' = l'[i] \quad \{ \text{array } A \mid l' \ast i64 \ast i' \ast A \mid l[i] \} \quad r' \\
\text{where} \\
\text{array } A \mid p = \exists l'. \ p + 0 \rightarrow l'[0] \ast \ldots \ast p + n \rightarrow l'[n] \\
&\ast A \mid l[0] \mid l'[0] \ast \ldots \ast A \mid l[n] \mid l'[n]
\end{align*}
\]

Problem: Does not work for \textit{array (array i64)}! (result is shared)
Nested Containers

Hoare-Rule for array-index:

\[
\{ \text{array } A \mid l' \ast i64 \mid i' \ast i < |l| \} \quad r' = l'[i] \quad \{ \text{array } A \mid l' \ast i64 \mid i' \ast A \ (l[i]) \} \quad r' \}
\]

where

\[
\text{array } A \mid p = \exists l'. \ p + 0 \leftrightarrow l'[0] \ast \ldots \ast p + n \leftrightarrow l'[n] \\
\ast A \mid l[0] \mid l[0] \ast \ldots \ast A \mid l[n] \mid l[n]
\]

Problem: Does not work for array \( \text{array } i64 \)! (result is shared)

- current approach: abstract data type: \( \alpha \) option list
  - None: element not in array
  - Manual ownership management

• future:
  - read-only sharing (fractional sep-logic?)
  - automation (as far as possible)
  - maybe inspiration from Rust.
Nested Containers

Hoare-Rule for array-index:

\{ \text{array A l l' * i64 i i' * i < |l|} \} \ r' = l[i'] \ { \text{array A l l' * i64 i i' * A (l[i])} \} \ r' \}

where

array A l p = \exists l'. p + 0 \mapsto l[0] * \ldots * p + n \mapsto l[n]

* A l[0] l'[0] * \ldots * A l[n] l'[n]

Problem: Does not work for \textit{array (array i64)}! (result is shared)

- current approach: abstract data type: \texttt{α option list}
  - None: element not in array
  - Manual ownership management

- future:
  - read-only sharing (fractional sep-logic?)
  - automation (as far as possible)
  - maybe inspiration from Rust.
Conclusions

Isabelle Refinement Framework

- powerful interactive theorem prover
- stepwise refinement
- libraries for standard DS and algorithms
- lot’s of automation
- efficient backend (LLVM)
= verified and efficient algorithms, at manageable effort

https://github.com/lammich/isabelle_llvm