The Isabelle Refinement Framework

Peter Lammich

University of Twente

May 2021
Introduction

• Peter Lammich
  • new assistant professor in FMT group
    • previously in Münster, Munich, Virginia Tech, Manchester
  • research: software verification
Introduction

- Peter Lammich
  - new assistant professor in FMT group
    - previously in Münster, Munich, Virginia Tech, Manchester
  - research: software verification
- if I’m not working: you’ll probably find me rock-climbing
Introduction

- Peter Lammich
  - new assistant professor in FMT group
    - previously in Münster, Munich, Virginia Tech, Manchester
  - research: software verification
- if I’m not working: you’ll probably find me rock-climbing
  - but I also enjoy hiking, biking (mtb, road, trek), racket sports (squash, badminton), ...
The Sloth, HVS 5a, at the Roaches in Peak District
Bull’s Crack, HVS 5a, at Heptonstall
Sport Climbing (somewhere in the Peaks)
Mountainbiking (at Lake Garda, after TransAlp)
Hiking in the Alps
... and now to the serious part: Software Verification

- Desirable properties of software
... and now to the serious part: Software Verification

- Desirable properties of software
  - correct
... and now to the serious part: Software Verification

- Desirable properties of software
  - correct (formally verified)
... and now to the serious part: Software Verification

- Desirable properties of software
  - correct (formally verified)
  - fast
... and now to the serious part: Software Verification

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation effort
and now to the serious part: Software Verification

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation and proof effort
… and now to the serious part: Software Verification

- Desirable properties of software
  - correct (formally verified)
  - fast
  - manageable implementation and proof effort
- Choose two!
... and now to the serious part: Software Verification

- 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

\[
\begin{align*}
\text{procedure} & \quad \text{augment} (g, f, p) \\
c_p & \leftarrow \min \{ g_f(u, v) \mid (u, v) \in p \} \\
\text{for all} & \, (u, v) \in p \, \text{do} \\
\text{if} & \, (u, v) \in g \, \text{then} \\
f(u, v) & \leftarrow f(u, v) + c_p \\
\text{else} & \\
f(v, u) & \leftarrow f(v, u) - c_p \\
\text{return} & \, f
\end{align*}
\]

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

\[
\begin{align*}
g & \text{: flow network} \\
s, t & \text{: source, target} \\
g_f & \text{: residual network}
\end{align*}
\]
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 \quad s, t: source, target \quad 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**

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

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

**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 \(g\)
Correctness

**procedure** EDOMONDS-KARP\((g, s, t)\)

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

**while** exists augmenting path in \( gf \) **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 \( gf \) 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** \texttt{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 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
    $p \leftarrow$ 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 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.
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\((g, s, t)\)

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

```cpp
textbook proof typically covers abstract algorithm.
```

```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;
}
```
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:
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($g$, $s$, $t$)
  $f \leftarrow \lambda(u, v) \cdot 0$
  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.
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\((g, s, t)\)
\[ f \leftarrow \lambda(u, v). 0 \]
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, ...
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 †,s †) ∈ node 64;

(t †,t †) ∈ node 64;

(g †,g †) ∈ adjl =⇒ (bfs s † t † g † , find shortest s t g) ∈ array

Shortcut notation:

(bfs, find shortest) ∈ node 64 → node 64 → adjl → array

• Implementations used for different parts must fit together!
Keeping it Manageable

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

Implementations used for different parts must fit together!
Keeping it Manageable

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

Implementation:

- **BFS implementation** uses adjacency lists.
- **EdmondsKarp** uses abstract graphs.
- Refinement relations between:
  - nodes and int64s (node 64);
  - adjacency lists and graphs (adjl);
  - arrays and paths (array).

Shortcut notation:

- \((bfs, find\_shortest) \in node^{64} \rightarrow node^{64} \rightarrow adjl \rightarrow array\)
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
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

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**
- 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 \textsc{EdmondsKarp}

• Data refinement
  • BFS implementation uses adjacency lists. \textsc{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 ($\text{node}_{64}$);
    - adjacency lists and graphs ($\text{adjl}$);
    - arrays and paths ($\text{array}$).
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).

\[(s \dagger, s) \in \text{node}_{64}; (t \dagger, t) \in \text{node}_{64}; (g \dagger, g) \in \text{adjl} \implies (\text{bfs } s \dagger, t \dagger, g \dagger, \text{find\_shortest } s \ t \ g) \in \text{array}\]
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 \texttt{EdmondsKarp}
- Data refinement
  - BFS implementation uses adjacency lists. \texttt{EdmondsKarp} used abstract graphs.
  - refinement relations between
    - nodes and int64s (\texttt{node64});
    - adjacency lists and graphs (\texttt{adjl});
    - arrays and paths (\texttt{array}).

\[(s^\dagger,s) \in \texttt{node64}; (t^\dagger,t) \in \texttt{node64}; (g^\dagger,g) \in \texttt{adjl} \implies (\texttt{bfs } s^\dagger t^\dagger g^\dagger, \texttt{find_shortest } s t g) \in \texttt{array}\]

Shortcut notation: \((\texttt{bfs,find_shortest}) \in \texttt{node64} \rightarrow \texttt{node64} \rightarrow \texttt{adjl} \rightarrow \texttt{array}\)
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).

\[(s_t, s) \in \text{node}_{64}; (t_t, t) \in \text{node}_{64}; (g_t, g) \in \text{adjl} \implies (\text{bfs } s_t, t_t, g_t, \text{ find}_\text{shortest } s, t, g) \in \text{array} \]

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

• 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

EdmondsKarp-1

EdmondsKarp-2

EdmondsKarp
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-1

EdmondsKarp-2

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

bfs

maxflow-spec

EdmondsKarp-1

capacity,flow $\rightarrow$ array
shortest-path $\rightarrow$ bfs
Refinement Architecture (simplified)

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

maxflow-spec
  → "textbook" proof
    → EdmondsKarp-1
Refinement Architecture (simplified)

shortest-path-spec

"textbook" proof

bfs-1

graph $\rightarrow$ adj.-list

queue $\rightarrow$ 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

substantial ideas
requires interactive proof

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

straightforward

mainly automatic

maxflow-spec

"textbook" proof

EdmondsKarp-1

modify residual graph

EdmondsKarp-2

node → int

graph → adj.-list

capacity, flow → array

shortest-path → bfs

EdmondsKarp
The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Tools + Automation
The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Tools + Automation
- Libraries
The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Tools + Automation
- Libraries
- Down to Ocaml/Haskell/Scala/SML and LLVM
IRF Core

• Nondeterministic programs shallowly embedded in HOL
  • As monad
    \[ \alpha \ M = \text{FAIL} \mid \text{SPEC (} \alpha \Rightarrow \text{bool)} \]
    return, bind
IRF Core

- Nondeterministic programs shallowly embedded in HOL
  - As monad
    \[ \alpha \ M = \text{FAIL} \mid \text{SPEC} (\alpha \Rightarrow \text{bool}) \]
  - return, bind
  - + if-then-else, recursion (via flat ccpo)
IRF Core

- Nondeterministic programs shallowly embedded in HOL
  - As monad
    \[
    \alpha \ M = \text{FAIL} \mid \text{SPEC } (\alpha \Rightarrow \text{bool})
    \]
  - return, bind
  - + if-then-else, recursion (via flat cpo)
  - + derived constructs (while, foreach, ...
IRF Core

• Nondeterministic programs shallowly embedded in HOL
  • As monad
    \[ \alpha \, M = \text{FAIL} \mid \text{SPEC} (\alpha \Rightarrow \text{bool}) \]
    return, bind
  • + if-then-else, recursion (via flat ccpo)
  • + derived constructs (while, foreach, ...)
  • = usable programming language
IRF Core

• Nondeterministic programs shallowly embedded in HOL
  • As monad
    \[ \alpha \ M = \text{FAIL} \mid \text{SPEC} (\alpha \Rightarrow \text{bool}) \]
    return, bind
  • + if-then-else, recursion (via flat ccpo)
  • + derived constructs (while, foreach, ...)
  • = usable programming language

• Refinement Calculus for Program and Data Refinement
IRF Core

- Nondeterministic programs shallowly embedded in HOL
  - As monad
    \[ \alpha M = FAIL | \text{SPEC} (\alpha \Rightarrow \text{bool}) \]
    return, bind
  - + if-then-else, recursion (via flat ccpo)
  - + derived constructs (while, foreach, ...)
  - = usable programming language

- Refinement Calculus for Program and Data Refinement

- Automation: VCG, semi-automatic data refinement
Imperative-HOL Backend

- imperative + functional language
- code generation to Ocaml/Haskell/Scala/SML
- automatic refinement of functional to imperative DS
  - if used linearly
Isabelle-LLVM Backend

- only imperative + bounded integers
- automatic placement of destructors
- semi-automatic in-bound proofs (eg for int → int64)
Refinement with Time

- Prove correctness and complexity
- *Resource currencies* to structure complexity proofs along refinement
- Down to Imperative-HOL / LLVM
Libraries

- Functional and Imperative data structures
  - readily usable for your developments
- Functional:
  - hashtable, red-black-trees, tries, Finger-Trees, (Skew) binomial queues, ...
- Imperative:
  - dynarray, heap, matrix, linked-list, hashtable, bit-vector, union-find, ROBDDs, B-Trees, ...
Highlight Verifications

- CAVA model checker
  - fully fledged LTL model checker
  - developed independently by 3 groups
  - newer development: MUNTA for timed automata
Highlight Verifications

- CAVA model checker
  - fully fledged LTL model checker
  - developed independently by 3 groups
  - newer development: MUNTA for timed automata
- Maxflow: Edmonds-Karp and Push-Relabel
  - textbook-level abstract correctness proof
  - efficient implementation
Highlight Verifications

- CAVA model checker
  - fully fledged LTL model checker
  - developed independently by 3 groups
  - newer development: MUNTA for timed automata
- Maxflow: Edmonds-Karp and Push-Relabel
  - textbook-level abstract correctness proof
  - efficient implementation
- GRAT: SAT-Solver verification tool
  - faster than unverified state-of-the-art tool drat-trim
Highlight Verifications

- **CAVA model checker**
  - fully fledged LTL model checker
  - developed independently by 3 groups
  - newer development: MUNTA for timed automata

- **Maxflow: Edmonds-Karp and Push-Relabel**
  - textbook-level abstract correctness proof
  - efficient implementation

- **GRAT: SAT-Solver verification tool**
  - faster than unverified state-of-the-art tool drat-trim

- **Introsort + Pdqsort**
  - verified correctness and complexity
  - on par with C++ impls from GNU libstdc++ and Boost
Future Work

- Concurrency
- Consolidate frameworks and tools
- Interesting algorithms to verify
Conclusions

Isabelle Refinement Framework

- powerful interactive theorem prover
- stepwise refinement
- libraries for standard DS
- lot’s of automation
- efficient backend (LLVM)

= verified and efficient algorithms, at manageable effort