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?

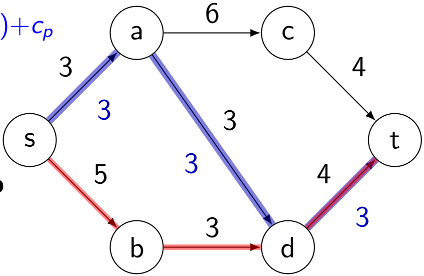
Introduction

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

Introduction

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

```
procedure AUGMENT($g, f, p$)
    $c_p \leftarrow \min\{g_f(u, v) \mid (u, v) \in p\}$
    for all $(u, v) \in p$ do
        if $(u, v) \in g$ then $f(u, v) \leftarrow f(u, v) + c_p$
        else $f(v, u) \leftarrow f(v, u) - c_p$
    return $f$
procedure EDMONDS-KARP($g, s, t$)
    $f \leftarrow \lambda(u, v).\ 0$
    while exists augmenting path in $g_f$ do
        $p \leftarrow$ shortest augmenting path
        $f \leftarrow$AUGMENT($g, f, p$)
```



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

Correctness

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

Correctness

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

Theorem (Ford-Fulkerson)

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

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

Correctness

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

Theorem (Ford-Fulkerson)

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

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

Proof.

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

□

Correctness

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

Theorem (Ford-Fulkerson)

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

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

Proof.

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

Correctness

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

Theorem

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

- *either δ_f decreases*
- *δ_f remains the same, and the number of edges in g_f that 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 δ_f *be the length of a shortest* s, t *- path in* g_f.
When augmenting with a shortest path,

- *either* δ_f *decreases*
- δ_f *remains the same, and the number of edges in* g_f *that 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
        $p \leftarrow$ shortest augmenting path
        $f \leftarrow$AUGMENT($g, f, p$)
```

Theorem

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

- *either δ_f decreases*
- *δ_f remains the same, and the number of edges in g_f that 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($g, s, t$)
    $f \leftarrow \lambda(u, v).\ 0$
    while exists augmenting path in $g_f$ do
        $p \leftarrow$ shortest augmenting path
        $f \leftarrow$ AUGMENT($g, f, p$)
```

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

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

   return flow;
}
```

textbook proof typically covers abstract algorithm.

Implementation

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

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

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

   return flow;
}
```

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

Implementation

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

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

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

   return flow;
}
```

textbook proof typically covers abstract algorithm.

but this is quite far from implementation. Still missing:

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

Implementation

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

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

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

    return flow;
}
```

textbook proof typically covers abstract algorithm.

but this is quite far from implementation. Still missing:

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

Implementation

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

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

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

    return flow;
}
```

textbook proof typically covers abstract algorithm.

but this is quite far from implementation. Still missing:

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

Implementation

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

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

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

   return flow;
}
```

textbook proof typically covers abstract algorithm.

but this is quite far from implementation. Still missing:

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

Keeping it Manageable

- A manageable proof needs modularization:

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

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

Keeping it Manageable

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

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

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

Keeping it Manageable

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

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

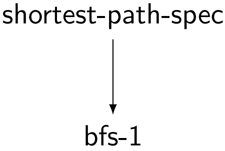
  Shortcut notation: $(\mathrm{bfs}, \mathrm{find\_shortest}) \in \mathrm{node}_{64} \rightarrow \mathrm{node}_{64} \rightarrow \mathrm{adjl} \rightarrow \mathrm{array}$
- Implementations used for different parts must fit together!

Refinement Architecture (simplified)

Refinement Architecture (simplified)

shortest-path-spec

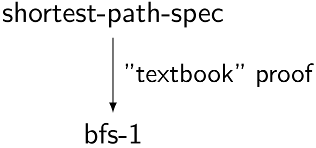
Refinement Architecture (simplified)



shortest-path-spec

bfs-1

Refinement Architecture (simplified)

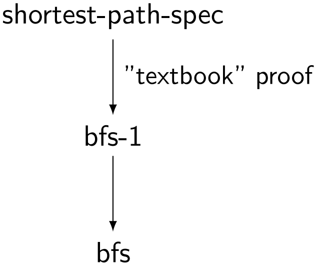


shortest-path-spec

"textbook" proof

bfs-1

Refinement Architecture (simplified)



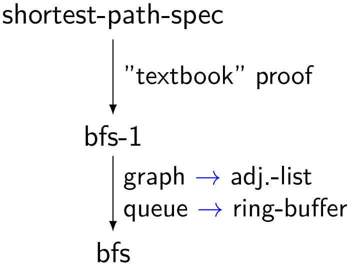
shortest-path-spec

"textbook" proof

bfs-1

bfs

Refinement Architecture (simplified)



shortest-path-spec

”textbook” proof

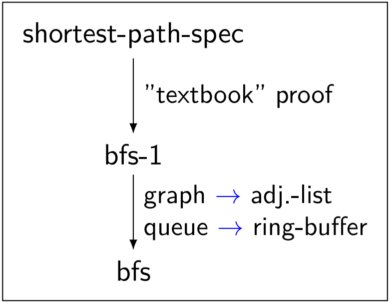
bfs-1

graph → adj.-list

queue → ring-buffer

bfs

Refinement Architecture (simplified)



shortest-path-spec

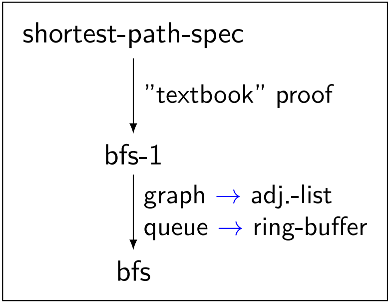
"textbook" proof

bfs-1

graph → adj.-list
queue → ring-buffer

bfs

Refinement Architecture (simplified)



shortest-path-spec

"textbook" proof

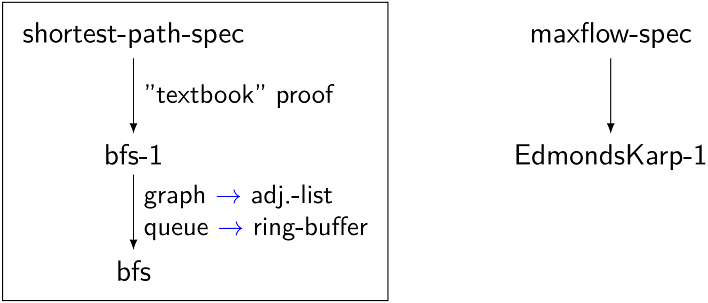
bfs-1

graph → adj.-list
queue → ring-buffer

bfs

maxflow-spec

Refinement Architecture (simplified)



shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list

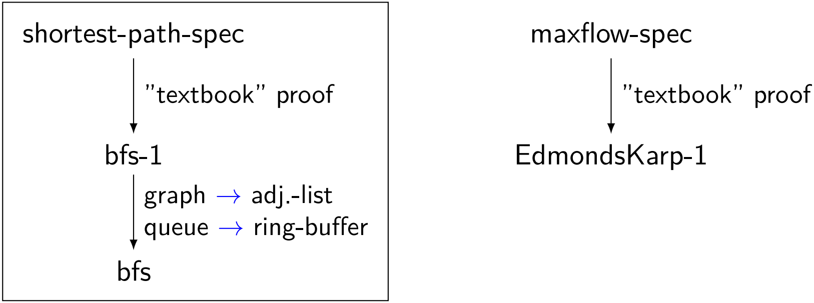
queue → ring-buffer

bfs

maxflow-spec

EdmondsKarp-1

Refinement Architecture (simplified)



shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list

queue → ring-buffer

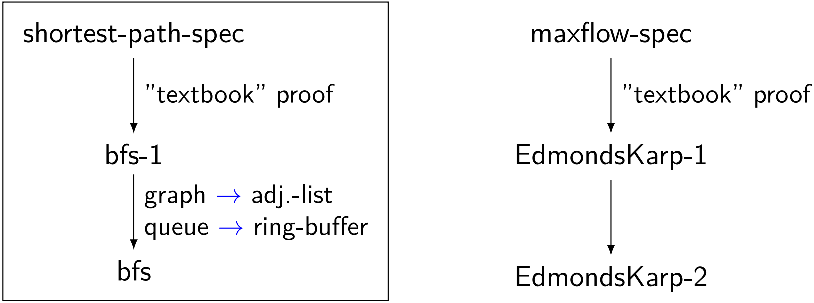
bfs

maxflow-spec

"textbook" proof

EdmondsKarp-1

Refinement Architecture (simplified)



shortest-path-spec

"textbook" proof

bfs-1

graph → adj.-list

queue → ring-buffer

bfs

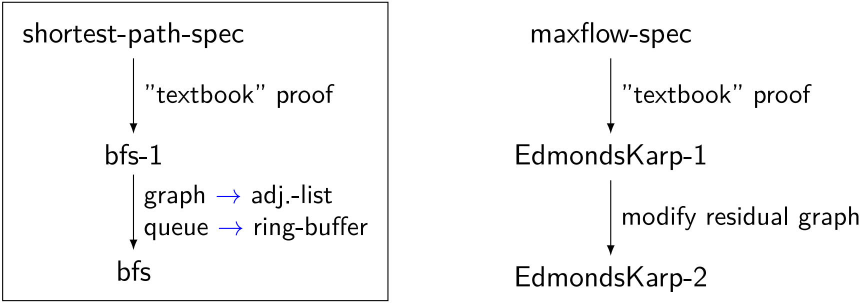
maxflow-spec

"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

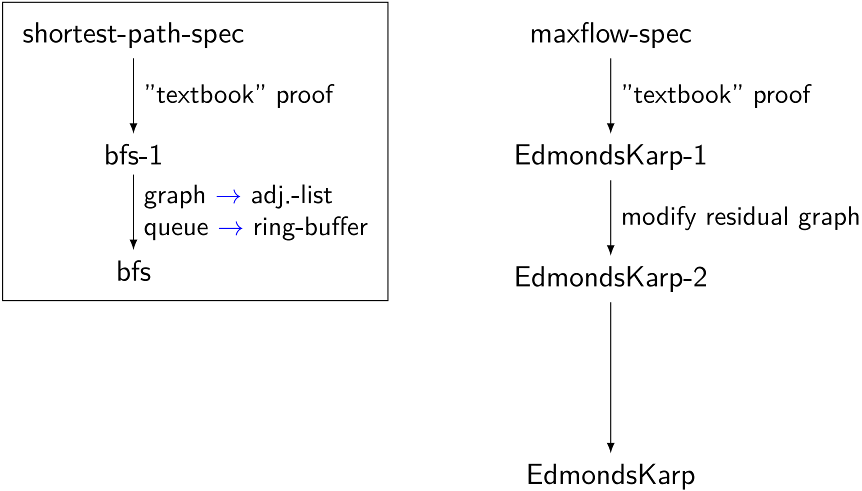
"textbook" proof

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

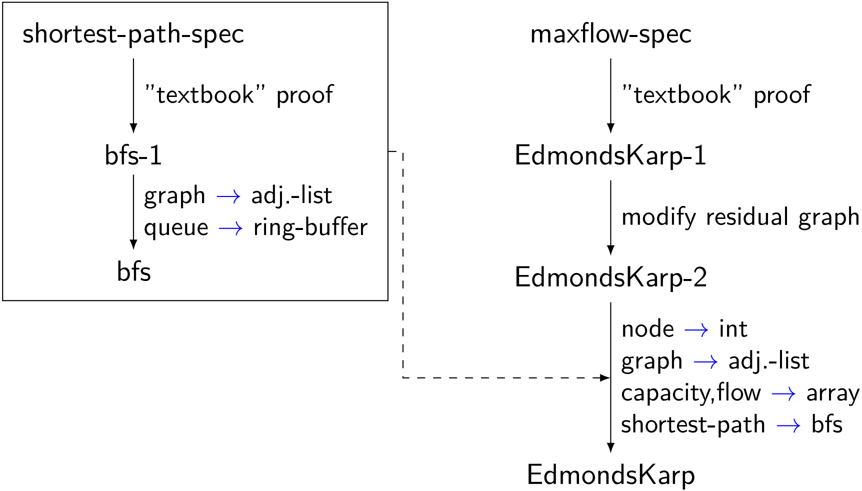
EdmondsKarp-1

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

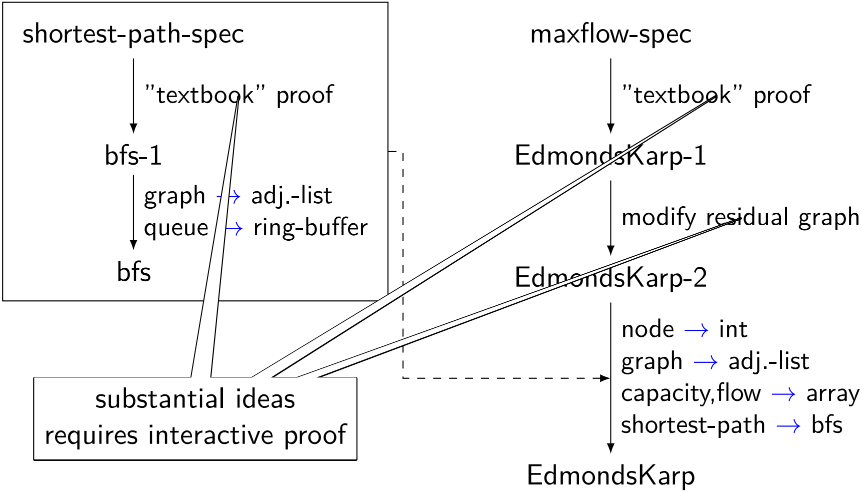
modify residual graph

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

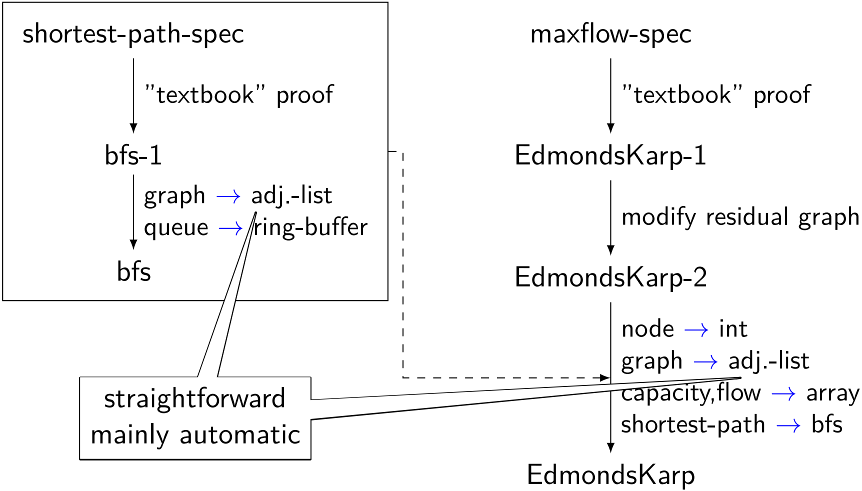
EdmondsKarp-2

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

EdmondsKarp

substantial ideas
requires interactive proof

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

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

- Nondetermistic programs shallowly embedded in HOL
  - As monad

    α M = FAIL | SPEC ($\alpha \Rightarrow$ bool)

    return, bind

IRF Core

- Nondetermistic programs shallowly embedded in HOL
  - As monad

    α M = FAIL | SPEC ($\alpha \Rightarrow$ bool)

    return, bind
  - + if-then-else, recursion (via flat ccpo)

IRF Core

- Nondetermistic programs shallowly embedded in HOL
    - As monad

      α M = FAIL | SPEC ($\alpha \Rightarrow$ bool)

      return, bind
    - + if-then-else, recursion (via flat ccpo)
    - + derived constructs (while, foreach, ...)

IRF Core

- Nondetermistic programs shallowly embedded in HOL
  - As monad

    α M = FAIL | SPEC ($\alpha \Rightarrow$ bool)

    return, bind
  - + if-then-else, recursion (via flat ccpo)
  - + derived constructs (while, foreach, ...)
  - = usable programming language

IRF Core

- Nondetermistic programs shallowly embedded in HOL
  - As monad

    α M = FAIL | SPEC ($\alpha \Rightarrow$ 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

- Nondetermistic programs shallowly embedded in HOL
  - As monad

    α M = FAIL | SPEC ($\alpha \Rightarrow$ 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

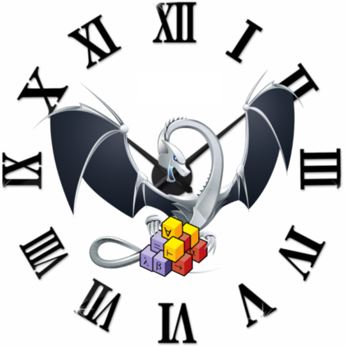
- imperative + functional language
- code generation to Ocaml/Haskell/Scala/SML
- automatic refinement of functional to imperative DS
    - if used linearly

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