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

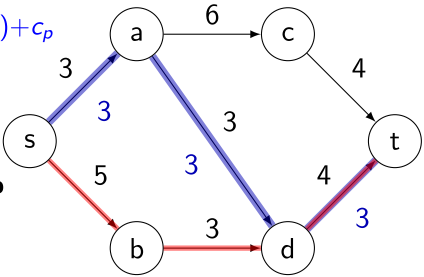
- What does it need to formally verify an algorithm?
  - E.g. maxflow algorithms

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

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

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

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

   return flow;
}
```

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

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

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

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

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

    return flow;
}
```

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

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

Keeping it Manageable

- A manageable proof needs modularization:

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

Keeping it Manageable

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

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

Keeping it Manageable

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

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

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

Keeping it Manageable

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

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

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

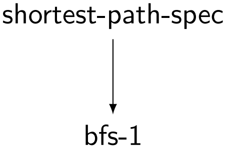
- Implementations used for different parts must fit together!

Refinement Architecture (simplified)

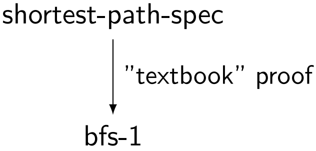
Refinement Architecture (simplified)

shortest-path-spec

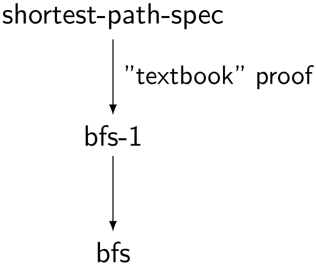
Refinement Architecture (simplified)



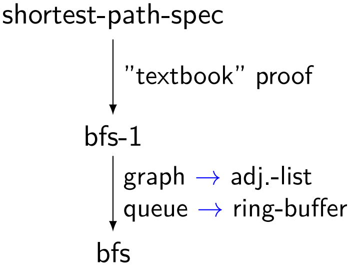
Refinement Architecture (simplified)



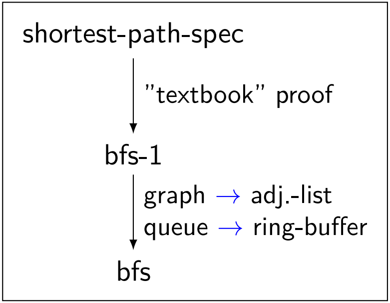
Refinement Architecture (simplified)



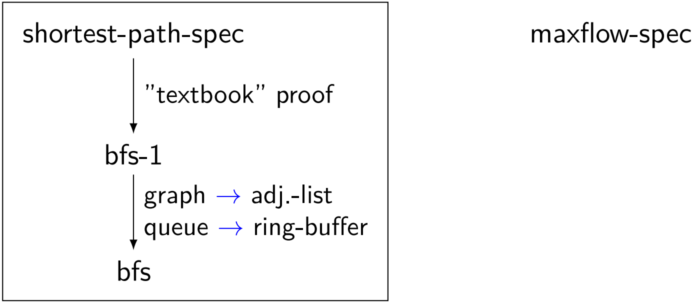
Refinement Architecture (simplified)



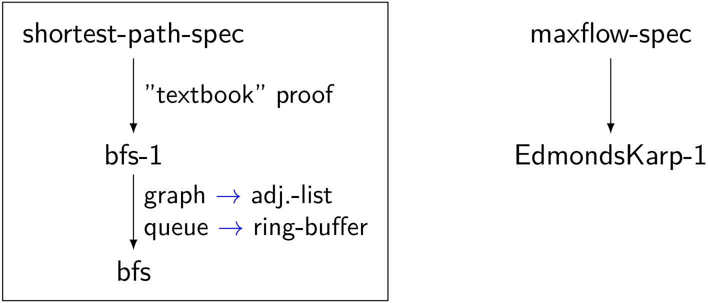
Refinement Architecture (simplified)



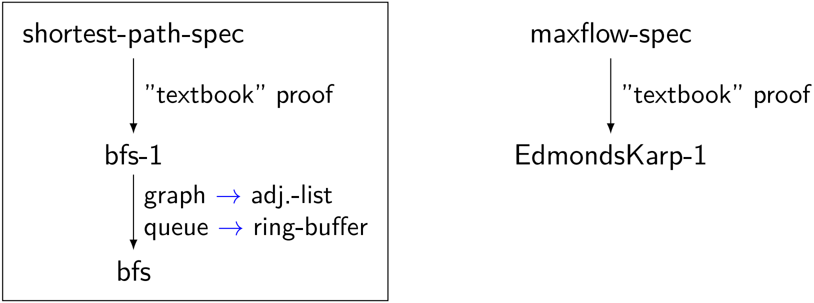
Refinement Architecture (simplified)



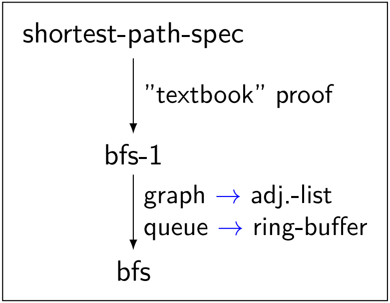
Refinement Architecture (simplified)

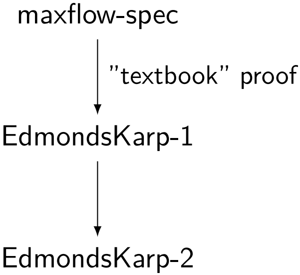


Refinement Architecture (simplified)

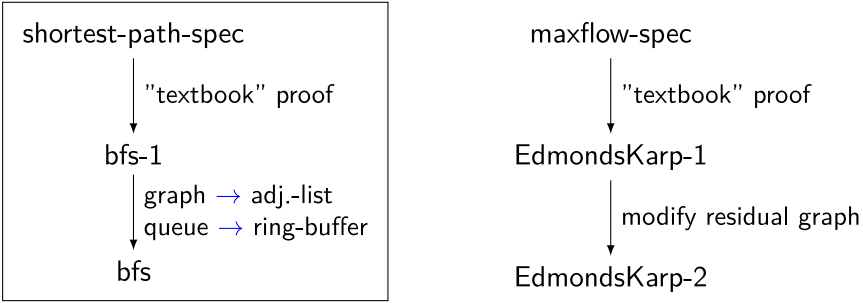


Refinement Architecture (simplified)

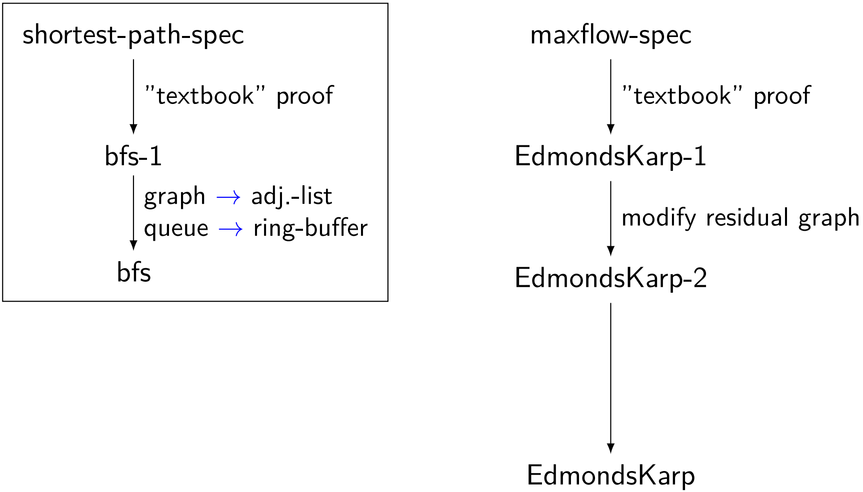




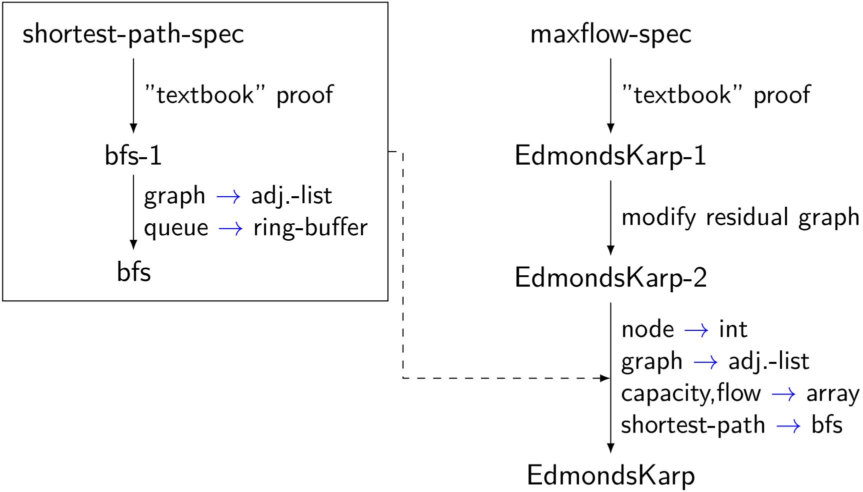
Refinement Architecture (simplified)



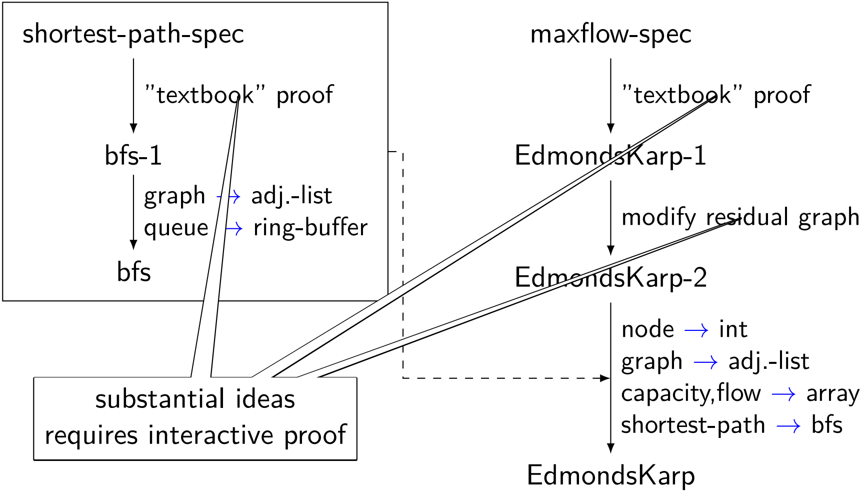
Refinement Architecture (simplified)



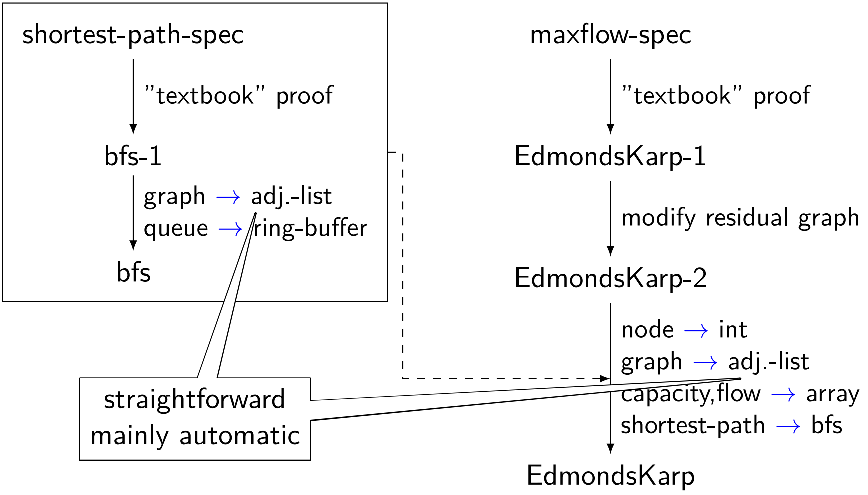
Refinement Architecture (simplified)



Refinement Architecture (simplified)



Refinement Architecture (simplified)



The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL

The Isabelle Refinement Framework

- Formalization of Refinement in Isabelle/HOL
- Batteries included

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

The Isabelle Refinement Framework

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

Formalizing Refinement

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

Formalizing Refinement

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

Formalizing Refinement

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

Code Generation

Translate HEAP to compilable code

Code Generation

Translate HEAP to compilable code

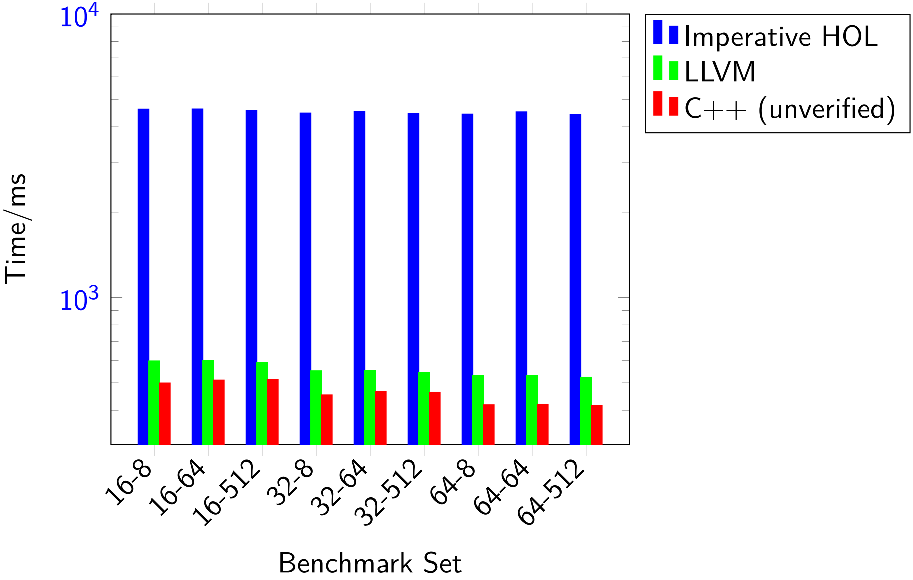
1. Imperative-HOL:
   - based on Isabelle's code generator
   - OCaml,SML,Haskell,Scala (using imp. features)
   - results cannot compete with optimized C/C++

Code Generation

Translate HEAP to compilable code

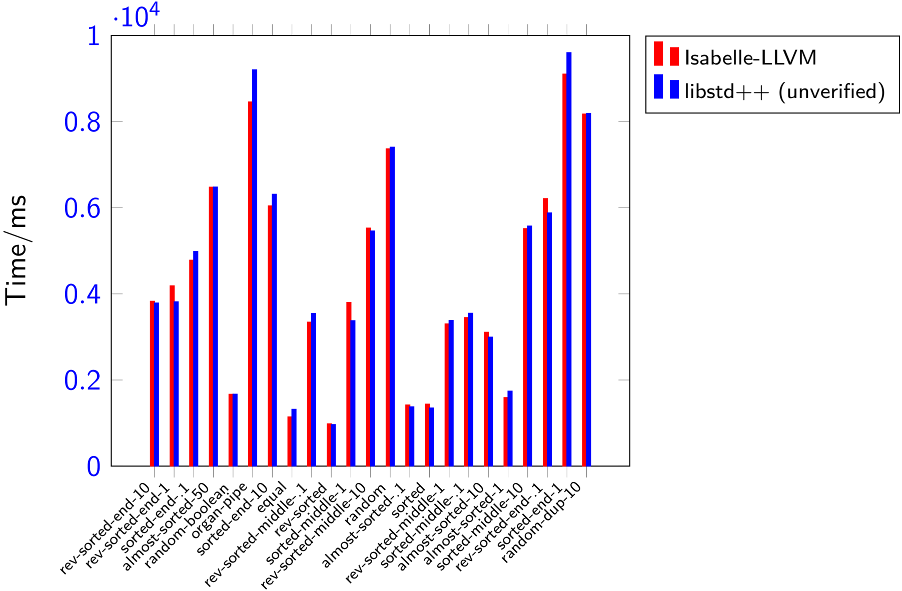
1. Imperative-HOL:
   - based on Isabelle's code generator
   - OCaml,SML,Haskell,Scala (using imp. features)
   - results cannot compete with optimized C/C++
2. NEW!: Isabelle-LLVM
   - shallow embedding of fragment of LLVM-IR
   - pretty-print to actual LLVM IR text
   - then use LLVM optimizer and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

Knuth Morris Pratt



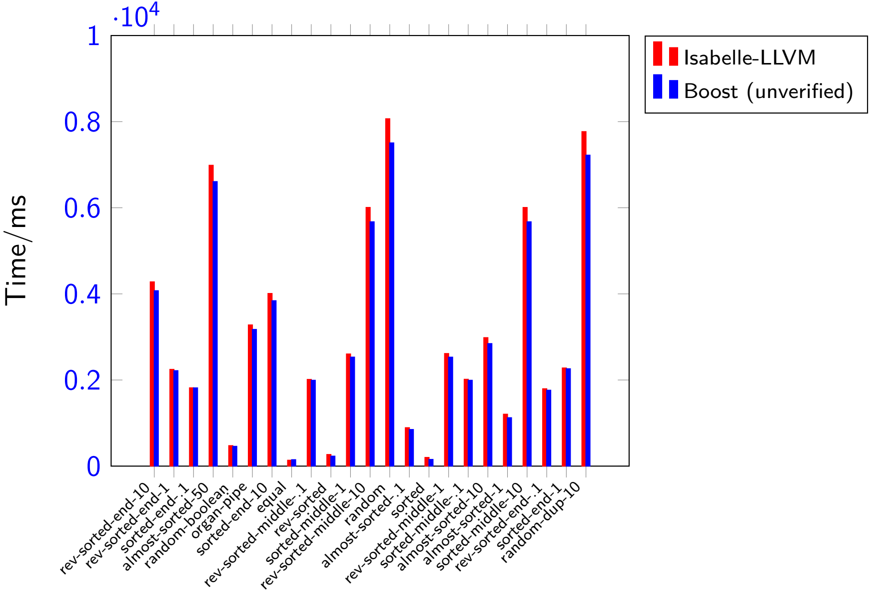
Execute *a*-*l* benchmark set from StringBench. Stop at first match.

Verified Sorting Algorithms: Introsort



Sorting $100 \cdot 10^6$ `uint64s` on Intel Core i7-8665U CPU, 32GiB RAM

Verified Sorting Algorithms: Pdqsort



Sorting $100 \cdot 10^6$ `uint64s` on Intel Core i7-8665U CPU, 32GiB RAM

Current (near Future) Projects

- Framework

Current (near Future) Projects

- Framework
    - Scalable Sepref Tool

Current (near Future) Projects

- Framework
  - Scalable Sepref Tool
  - Nested Containers

Current (near Future) Projects

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

Current (near Future) Projects

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

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

Sepref Tool

- 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|; ++i) {
  t1 = l[i]
  if (t1 ∉ S) {
   *: assert (c<|l|)
   ++c
   S={t1} ∪ S
  } } }
```

Sepref Tool

- 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|; ++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'
   } } free S' }
```

Sepref Tool

- 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|; ++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′
    } } free S′ }
```

At *: *array i64 l l′ * hm i64 S S′ * ...*

Nested Containers

Hoare-Rule for array-index:

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

Nested Containers

Hoare-Rule for array-index:

$\{$array A l l$'$ $*$ i64 i i$'$ $*$ i$<|$l$|\}$ r$'$=l$'$[i$'$] $\{$ 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 *array* (*array i64*)! (result is shared)

Nested Containers

Hoare-Rule for array-index:

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

where

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

Problem: Does not work for *array* (*array i64*)! (result is shared)

- current approach: abstract data type: α *option list*
  - None: element not in array
  - Manual ownership management

Nested Containers

Hoare-Rule for array-index:

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

where

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

Problem: Does not work for *array* (*array i64*)! (result is shared)

- current approach: abstract data type: 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