Refinement based Algorithm Development

with Isabelle/HOL

Peter Lammich

Fakultät für Informatik
Technische Universität München

2018-8-30

Introduction

- Why Program Verification

Introduction

- Why Program Verification
  - See previous lectures ;)

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
  - Isabelle

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
  - Isabelle
  - Coq

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
    - Isabelle
    - Coq
    - Other

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
  - Isabelle
  - Coq
  - Other
- Maxflow Algorithms (eg Ford-Fulkerson, Edmonds-Karp, Push-Relabel)

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
  - Isabelle
  - Coq
  - Other
- Maxflow Algorithms (eg Ford-Fulkerson, Edmonds-Karp, Push-Relabel)
- Parametricity (eg Theorems for Free!)

Short Poll

Raise your hand if you know/ have heard of

- Monads (eg in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
    - Isabelle
    - Coq
    - Other
- Maxflow Algorithms (eg Ford-Fulkerson, Edmonds-Karp, Push-Relabel)
- Parametricity (eg Theorems for Free!)
- Separation Logic

Overview

- Refinement based approach to algorithm development
  - From pseudocode to implementation
- Everything verified within Isabelle/HOL
  - We do not trust any other tools
- Edmonds-Karp Maxflow algorithm as running example
  - Approach has been used for many formalizations. Highlights:
    MUNTA: Timed Automata Model Checker
    GRAT: SAT solver certification
    CAVA: LTL model checker

Lecture Material:
`http://www21.in.tum.de/~lammich/vtsa2018_isabelle.tgz`

Isabelle/HOL Theorem Prover

- LCF-style: Based on small trusted kernel
  - Only this kernel can prove theorems
  - Large set of tools on top of kernel
  - Errors in tools do not endanger soundness

Isabelle/HOL Theorem Prover

- LCF-style: Based on small trusted kernel
  - Only this kernel can prove theorems
  - Large set of tools on top of kernel
  - Errors in tools do not endanger soundness
- Interactive
  - Proving as interactive "game" between user and prover
  - Sophisticated proof search tools also available (sledgehammer)

Isabelle/HOL Theorem Prover

- LCF-style: Based on small trusted kernel
  - Only this kernel can prove theorems
  - Large set of tools on top of kernel
  - Errors in tools do not endanger soundness
- Interactive
  - Proving as interactive "game" between user and prover
  - Sophisticated proof search tools also available (sledgehammer)
- Archive of Formal Proofs `https://www.isa-afp.org/`
  - Large set of theories readily available
  - Maintained to run with latest Isabelle version

Isabelle/HOL

- This lecture: Not an Isabelle introduction

Isabelle/HOL

- This lecture: Not an Isabelle introduction
- Trying to present ideas independent of Isabelle

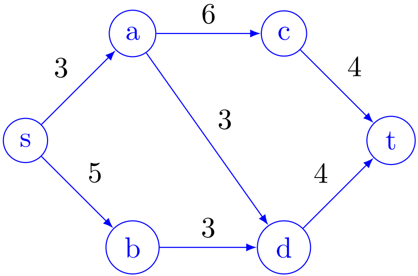
Isabelle/HOL

- This lecture: Not an Isabelle introduction
- Trying to present ideas independent of Isabelle
- But many examples and demos in Isabelle

Flow Networks and Flows

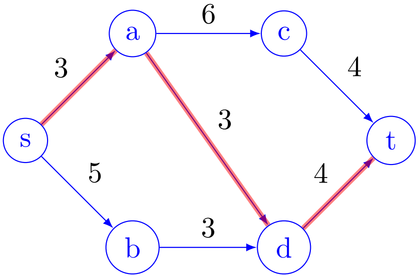
- Flow Network
  - Directed graph
  - Edges annotated with capacity
  - Distinguished source and sink node
- Flow
  - Generated only at source
  - Consumed only at sink
  - Must not exceed edge capacities

Finding Maximum Flow



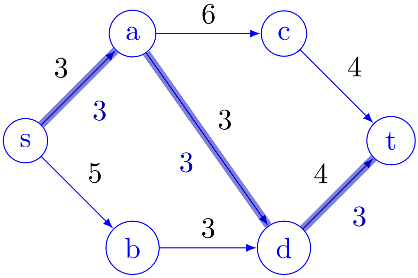
- Start with empty flow

Finding Maximum Flow



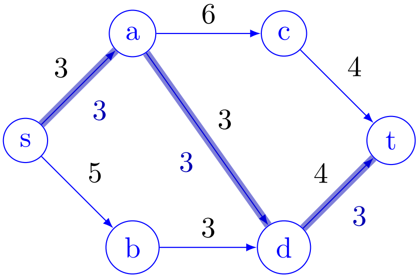
- Start with empty flow
- Find *augmenting path*

Finding Maximum Flow



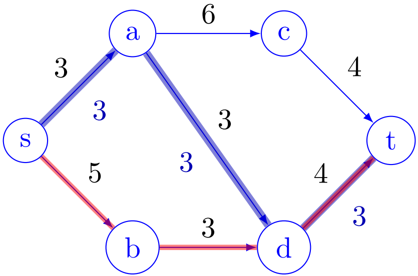
- Start with empty flow
- Find *augmenting path*
    - Increase flow

Finding Maximum Flow



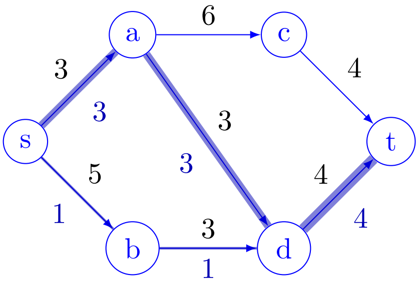
- Start with empty flow
- Find *augmenting path*
    - Increase flow
- Repeat

Finding Maximum Flow



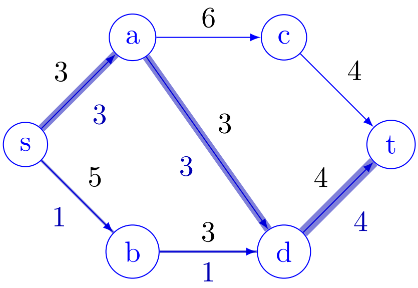
- Start with empty flow
- Find *augmenting path*
  - Increase flow
- Repeat

Finding Maximum Flow



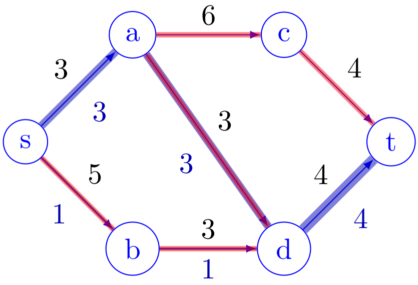
- Start with empty flow
- Find *augmenting path*
  - Increase flow
- Repeat

Finding Maximum Flow



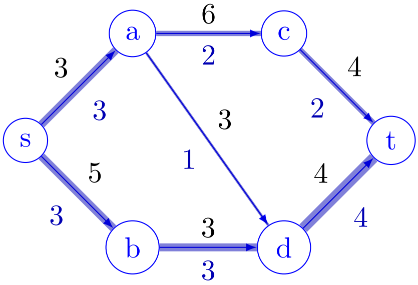
- Start with empty flow
- Find *augmenting path*
    - Increase flow
- Repeat
- May need to take back flow
    - To increase overall value

Finding Maximum Flow



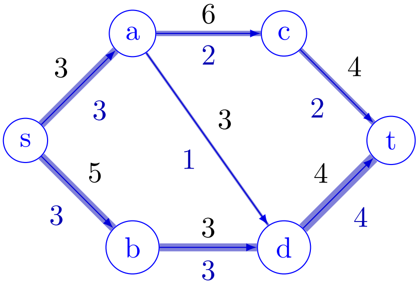
- Start with empty flow
- Find *augmenting path*
    - Increase flow
- Repeat
- May need to take back flow
    - To increase overall value

Finding Maximum Flow



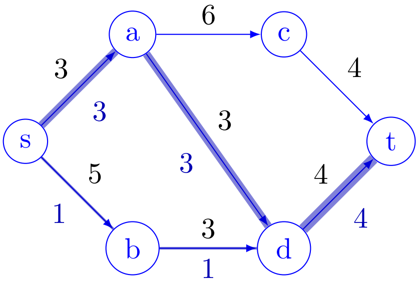
- Start with empty flow
- Find *augmenting path*
    - Increase flow
- Repeat
- May need to take back flow
    - To increase overall value

Finding Maximum Flow



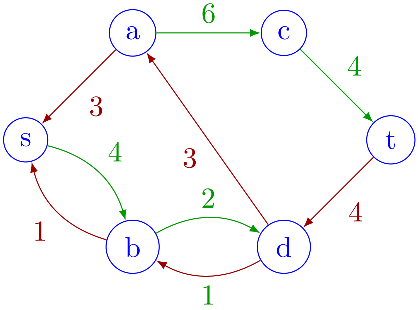
- Start with empty flow
- Find *augmenting path*
  - Increase flow
- Repeat
- May need to take back flow
  - To increase overall value
- Flow is maximal now

- Flow that can be moved between nodes
  - By increasing or taking back flow



Residual Graph
of Network and Flow

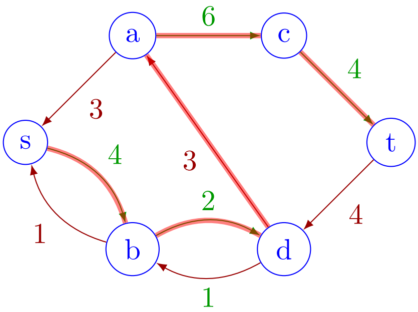
- Flow that can be moved between nodes
  - By increasing or taking back flow



Residual Graph

of Network and Flow

- Flow that can be moved between nodes
  - By increasing or taking back flow
- Augmenting path: s-t path in residual graph



Ford-Fulkerson Method

- Theorem: Flow is maximal iff $\nexists$ augmenting path
  - Corollary of Min-Cut/Max-Flow theorem

Ford-Fulkerson Method

- Theorem: Flow is maximal iff $\nexists$ augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

```
set flow to zero
while exists augmenting path
  augment flow along path
```

Ford-Fulkerson Method

- Theorem: Flow is maximal iff $\nexists$ augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

```
set flow to zero
while exists augmenting path
  augment flow along path
```

- Partial correctness: obvious

Ford-Fulkerson Method

- Theorem: Flow is maximal iff $\nexists$ augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

```
set flow to zero
while exists augmenting path
  augment flow along path
```

- Partial correctness: obvious
- Termination: only for integer/rational capacities

Ford-Fulkerson Method

- Theorem: Flow is maximal iff $\nexists$ augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

```
set flow to zero
while exists augmenting path
  augment flow along path
```

- Partial correctness: obvious
- Termination: only for integer/rational capacities
- Edmonds/Karp: choose shortest augmenting path
  - $O(VE)$ iterations for real-valued capacities
  - Using BFS to find path: $O(VE^2)$ algorithm

Implementation

- Work on residual graphs instead of flows
  - Augmentation can be done on residual graphs
  - Flow can be extracted from residual graph

Implementation

- Work on residual graphs instead of flows
  - Augmentation can be done on residual graphs
  - Flow can be extracted from residual graph
- Use BFS to find shortest path

Implementation

- Work on residual graphs instead of flows
  - Augmentation can be done on residual graphs
  - Flow can be extracted from residual graph
- Use BFS to find shortest path
- Data structures for graph, augmenting path, BFS algorithm

Implementation

- Work on residual graphs instead of flows
    - Augmentation can be done on residual graphs
    - Flow can be extracted from residual graph
- Use BFS to find shortest path
- Data structures for graph, augmenting path, BFS algorithm
- Write Standard-ML program (Or Scala, Haskell, Ocaml, ...)

Data Structures

- Residual Graph
  - Operations: successors of node, capacity of edge
  - Nodes by natural numbers from $\{0..<N\}$
  - Adjacency matrix by array ($\mathit{capacity}[N][N]$)
  - Adjacency map by array $(\mathit{node\ list})[N]$

Data Structures

- Residual Graph
  - Operations: successors of node, capacity of edge
  - Nodes by natural numbers from $\{0..<N\}$
  - Adjacency matrix by array ($\mathit{capacity}[N][N]$)
  - Adjacency map by array $(\mathit{node\ list})[N]$
- BFS algorithm

Data Structures

- Residual Graph
  - Operations: successors of node, capacity of edge
  - Nodes by natural numbers from $\{0..<N\}$
  - Adjacency matrix by array ($capacity[N][N]$)
  - Adjacency map by array $(node\ list)[N]$
- BFS algorithm
  - Predecessor map: Array ((node option)[N])
    - Or use $int[N]$ and map $None$ to -1

Data Structures

- Residual Graph
  - Operations: successors of node, capacity of edge
  - Nodes by natural numbers from $\{0..<N\}$
  - Adjacency matrix by array ($capacity[N][N]$)
  - Adjacency map by array $(node\ list)[N]$
- BFS algorithm
  - Predecessor map: Array ((node option)[N])
    - Or use $int[N]$ and map $None$ to -1
  - Current and next set: *node list*

Data Structures

- Residual Graph
  - Operations: successors of node, capacity of edge
  - Nodes by natural numbers from $\{0..<N\}$
  - Adjacency matrix by array ($\mathit{capacity}[N][N]$)
  - Adjacency map by array $(\mathit{node\ list})[N]$
- BFS algorithm
  - Predecessor map: Array ((node option)[N])
    - Or use $\mathit{int}[N]$ and map None to -1
  - Current and next set: *node list*
- Augmenting path: *node list*
  - Could use predecessor map directly
  - but converting to list gives cleaner interface, and is no bottleneck

How to Formally Verify Implementation?

- Informally, correctness argument given on abstract algorithm

```
set flow to zero
while exists augmenting path
  augment flow along path
```

  - Using rich background theory on network flows

How to Formally Verify Implementation?

- Informally, correctness argument given on abstract algorithm

```
set flow to zero
while exists augmenting path
  augment flow along path
```

  - Using rich background theory on network flows
- Then, we described how to implement the parts of the algorithm

How to Formally Verify Implementation?

- Informally, correctness argument given on abstract algorithm

```
set flow to zero
while exists augmenting path
  augment flow along path
```

  - Using rich background theory on network flows
- Then, we described how to implement the parts of the algorithm
- And concluded: Abstract algorithm is correct and implemented correctly
  $\Longrightarrow$ implementation is correct

Formal Verification

- Want to do same approach formally

Formal Verification

- Want to do same approach formally
- Give precise semantics to abstract algorithm

Formal Verification

- Want to do same approach formally
- Give precise semantics to abstract algorithm
- Prove that it returns maxflow

Formal Verification

- Want to do same approach formally
- Give precise semantics to abstract algorithm
- Prove that it returns maxflow
- Give semantics to implementation

Formal Verification

- Want to do same approach formally
- Give precise semantics to abstract algorithm
- Prove that it returns maxflow
- Give semantics to implementation
- Show that it corresponds to abstract algorithm

Formal Verification

- Want to do same approach formally
- Give precise semantics to abstract algorithm
- Prove that it returns maxflow
- Give semantics to implementation
- Show that it corresponds to abstract algorithm
- By transitivity, argue that implementation returns maxflow
  impl $\leq$ *abstract algo* $\leq$ *specification*

Modularity

- Standard modular design patterns apply
  - E.g. BFS implemented and proved correct independently of Edmonds-Karp algorithm
  - Only interface (graphs and paths) must match

Modularity

- Standard modular design patterns apply
  - E.g. BFS implemented and proved correct independently of Edmonds-Karp algorithm
  - Only interface (graphs and paths) must match
- Separation of concerns (abstract correctness, implementation)
  - Also done naturally in textbooks

Modularity

- Standard modular design patterns apply
  - E.g. BFS implemented and proved correct independently of Edmonds-Karp algorithm
  - Only interface (graphs and paths) must match
- Separation of concerns (abstract correctness, implementation)
  - Also done naturally in textbooks

Why should one care about BFS algorithm or non-overflow of adjacency matrix array access, when demonstrating the abstract idea of Edmonds-Karp algorithm?

Features Required for Abstract Algorithm

- Standard control flow (if, recursion)

Features Required for Abstract Algorithm

- Standard control flow (if, recursion)
- Mathematical concepts (sets, functions, graphs)

Features Required for Abstract Algorithm

- Standard control flow (if, recursion)
- Mathematical concepts (sets, functions, graphs)
- Nondeterminism
  - Cannot determine actual shortest path in abstract algorithm
  - There may be many, and implementation details of BFS decide which one is returned
  - Abstractly: nondeterministically choose among all possibilities

Interlude: Monads for Programming

- A monad is a type $'a\ M$ with operations

  $\textbf{return} :: 'a \Rightarrow 'a\ M$

  $bind :: 'a\ M \Rightarrow ('a \Rightarrow 'b\ M) \Rightarrow 'b\ M$

Interlude: Monads for Programming

- A monad is a type $'a\ M$ with operations
  $\mathbf{return} :: {'a} \Rightarrow {'a}\ M$
  $bind :: {'a}\ M \Rightarrow ({'a} \Rightarrow {'b}\ M) \Rightarrow {'b}\ M$
- Intuition:
  return a value,
  bind the result of m_1 to variable x, then execute $m_2\ x$

Interlude: Monads for Programming

- A monad is a type $'a\ M$ with operations
  return $::\ 'a \Rightarrow\ 'a\ M$
  bind $::\ 'a\ M \Rightarrow (\,'a \Rightarrow\ 'b\ M) \Rightarrow\ 'b\ M$
- Intuition:
  return a value,
  bind the result of m_1 to variable x, then execute $m_2\ x$
- Syntax sugar

Interlude: Monads for Programming

- A monad is a type $'a\ M$ with operations
  $\textbf{return} :: 'a \Rightarrow 'a\ M$
  $bind :: 'a\ M \Rightarrow ('a \Rightarrow 'b\ M) \Rightarrow 'b\ M$
- Intuition:
  return a value,
  bind the result of m_1 to variable x, then execute $m_2\ x$
- Syntax sugar
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ m_2\ x\ \} = bind\ m_1\ (\lambda x.\ m_2\ x)$

Interlude: Monads for Programming

- A monad is a type $'a\ M$ with operations
  $\textbf{return} :: 'a \Rightarrow 'a\ M$
  $bind :: 'a\ M \Rightarrow ('a \Rightarrow 'b\ M) \Rightarrow 'b\ M$
- Intuition:
  return a value,
  bind the result of m_1 to variable x, then execute $m_2\ x$
- Syntax sugar
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ m_2\ x\ \} = bind\ m_1\ (\lambda x.\ m_2\ x)$
- Monad laws
  - $\textbf{do}\ \{\ x \leftarrow \textbf{return}\ v;\ f\ x\ \} = f\ v$
  - $\textbf{do}\ \{\ x \leftarrow m;\ \textbf{return}\ x\ \} = m$
  - $\textbf{do}\ \{\ y \leftarrow \textbf{do}\ \{x \leftarrow m;\ f\ x\ \};\ g\ y\ \} = \textbf{do}\ \{\ x \leftarrow m;\ y \leftarrow f\ x;\ g\ y\ \}$

Monad Syntax Sugar

- infix bind notation

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \qquad = \mathit{bind}\ m_1\ \ m_2$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \ggg\!= m_2 \quad = \mathit{bind}\ m_1\ \ m_2$
  - $m_1 \gg m_2 \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \quad = \mathit{bind}\ m_1\ \ m_2$
  - $m_1 \gg m_2 \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \qquad = bind\ m_1\ \ m_2$
  - $m_1 \gg m_2 \qquad = bind\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\textbf{do}\ \{\ m_1;\ m_2\ \} \qquad = bind\ m_1\ (\lambda\_.\ m_2)$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \qquad = \mathit{bind}\ m_1\ m_2$
  - $m_1 \gg m_2 \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\textbf{do}\ \{\ m_1;\ m_2\ \} \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2\ x;\ z \leftarrow m_3\ x\ y;\ \ldots\ \}$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \qquad = \mathit{bind}\ m_1\ \ m_2$
  - $m_1 \gg m_2 \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\textbf{do}\ \{\ m_1;\ m_2\ \} \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2\ x;\ z \leftarrow m_3\ x\ y;\ \dots\ \}$
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2;\ m_3;\ z \leftarrow m_4\ x;\ \dots\ \}$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \quad = \mathit{bind}\ m_1\ m_2$
  - $m_1 \gg m_2 \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\mathbf{do}\ \{\ m_1;\ m_2\ \} \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
  - $\mathbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2\ x;\ z \leftarrow m_3\ x\ y;\ \ldots\ \}$
  - $\mathbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2;\ m_3;\ z \leftarrow m_4\ x;\ \ldots\ \}$
  - $\mathbf{do}\ \{\ \ldots;\ (x_1, x_2, x_3) \leftarrow m;\ \ldots\ \}$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \gg\!= m_2 \qquad = \mathit{bind}\ m_1\ \ m_2$
  - $m_1 \gg m_2 \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\textbf{do}\ \{\ m_1;\ m_2\ \} \qquad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2\ x;\ z \leftarrow m_3\ x\ y;\ \ldots\ \}$
  - $\textbf{do}\ \{\ x \leftarrow m_1;\ y \leftarrow m_2;\ m_3;\ z \leftarrow m_4\ x;\ \ldots\ \}$
  - $\textbf{do}\ \{\ \ldots;\ (x_1,x_2,x_3) \leftarrow m;\ \ldots\ \}$
  - $\textbf{do}\ \{\ \ldots;\ \textbf{let}\ (x_1,x_2,x_3) = a;\ \ldots\ \}$

Monad Syntax Sugar

- infix bind notation
  - $m_1 \ggg= m_2 \quad = \mathit{bind}\ m_1\ m_2$
  - $m_1 \gg m_2 \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
- do-notation
  - $\textbf{do}\ \{\ m_1;\ m_2\ \} \quad = \mathit{bind}\ m_1\ (\lambda\_.\ m_2)$
  - $\textbf{do}\ \{\ x {\leftarrow} m_1;\ y {\leftarrow} m_2\ x;\ z \leftarrow m_3\ x\ y;\ \ldots\ \}$
  - $\textbf{do}\ \{\ x {\leftarrow} m_1;\ y {\leftarrow} m_2;\ m_3;\ z {\leftarrow} m_4\ x;\ \ldots\ \}$
  - $\textbf{do}\ \{\ \ldots;\ (x_1, x_2, x_3) {\leftarrow} m;\ \ldots\ \}$
  - $\textbf{do}\ \{\ \ldots;\ \textbf{let}\ (x_1, x_2, x_3) = a;\ \ldots\ \}$
- Any HOL function, and its syntax

$$
\begin{array}{l}
\textbf{do}\ \{ \\
\quad x {\leftarrow} m; \\
\quad \textbf{if}\ x{<}0\ \textbf{then return}\ (-1\ ) \\
\quad \textbf{else if}\ x{=}0\ \textbf{then return}\ 0 \\
\quad \textbf{else if}\ x{>}0\ \textbf{then return}\ 1 \\
\}
\end{array}
$$

Monads for Programming

- Monad models sequential execution
  - **do** $\{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f

Monads for Programming

- Monad models sequential execution
  - **do** $\{ x {\leftarrow} m;\ f\ x \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$

Monads for Programming

- Monad models sequential execution
  - **do** $\{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$
  - Computations that can fail: $M = option$

    return $x = Some\ x$
    $bind\ m_1\ m_2 = \textbf{case}\ m_1\ \textbf{of}\ None \Rightarrow None \mid Some\ x \Rightarrow m_2\ x$
    $fail = None$

Monads for Programming

- Monad models sequential execution
  - **do** $\{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$
  - Computations that can fail: $M{=}option$

    $\textbf{return}\ x = Some\ x$
    $bind\ m_1\ m_2 = \textbf{case}\ m_1\ \textbf{of}\ None \Rightarrow None \mid Some\ x \Rightarrow m_2\ x$
    $fail = None$

    - Example: $\textbf{do}\ \{\ \textbf{if}\ x{=}0\ \textbf{then}\ fail\ \textbf{else return}\ (1\ /\ x)\ \}$

Monads for Programming

- Monad models sequential execution
  - **do** $\{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$
  - Computations that can fail: $M{=}option$

    $\textbf{return}\ x = Some\ x$
    $bind\ m_1\ m_2 = \textbf{case}\ m_1\ \textbf{of}\ None \Rightarrow None \mid Some\ x \Rightarrow m_2\ x$
    $fail = None$

    - Example: **do** $\{\ \textbf{if}\ x{=}0\ \textbf{then}\ fail\ \textbf{else}\ \textbf{return}\ (1\ /\ x)\ \}$
  - Nondeterministic computations $M{=}set$

    $\textbf{return}\ x = \{x\}$
    $bind\ m_1\ m_2 = \bigcup \{\ m_2\ x \mid x.\ x{\in}m_1\ \}$
    $choose\ x.\ \Phi\ x = \{\ x.\ \Phi\ x\ \}$

Monads for Programming

- Monad models sequential execution
  - **do** $\{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$
  - Computations that can fail: $M = option$

    $\mathbf{return}\ x = Some\ x$
    $bind\ m_1\ m_2 = \mathbf{case}\ m_1\ \mathbf{of}\ None \Rightarrow None \mid Some\ x \Rightarrow m_2\ x$
    $fail = None$

    - Example: **do** $\{\ \mathbf{if}\ x{=}0\ \mathbf{then}\ fail\ \mathbf{else}\ \mathbf{return}\ (1\ /\ x)\ \}$
  - Nondeterministic computations $M = set$

    $\mathbf{return}\ x = \{x\}$
    $bind\ m_1\ m_2 = \bigcup \{\ m_2\ x \mid x.\ x \in m_1\ \}$
    $choose\ x.\ \Phi\ x = \{\ x.\ \Phi\ x\ \}$

    - Example: **do** $\{\ e \leftarrow choose\ (u,v).\ c(u,v) > 0;\ \ldots\ \}$

Monads for Programming

- Monad models sequential execution
  - $\textbf{do}\ \{\ x \leftarrow m;\ f\ x\ \}$ First execute m, then f
- More functionality can be added by structure of type $'a\ M$
  - Computations that can fail: $M = option$

    $\textbf{return}\ x = Some\ x$
    $bind\ m_1\ m_2 = \textbf{case}\ m_1\ \textbf{of}\ None \Rightarrow None \mid Some\ x \Rightarrow m_2\ x$
    $fail = None$

    - Example: $\textbf{do}\ \{\ \textbf{if}\ x{=}0\ \textbf{then}\ fail\ \textbf{else}\ \textbf{return}\ (1\ /\ x)\ \}$
  - Nondeterministic computations $M = set$

    $\textbf{return}\ x = \{x\}$
    $bind\ m_1\ m_2 = \bigcup \{\ m_2\ x \mid x.\ x \in m_1\ \}$
    $choose\ x.\ \Phi\ x = \{\ x.\ \Phi\ x\ \}$

    - Example: $\textbf{do}\ \{\ e \leftarrow choose\ (u,v).\ c(u,v) > 0;\ \ldots\ \}$
  - Many more: exceptions, state, output, probability, ...

The *nres* Monad

- Combines failure and nondeterminism monad. $M = nres$ where

  $\textbf{datatype}\ 'a\ nres = FAIL \mid RES\ 'a\ set$

  $$\begin{aligned}
  &\textbf{return}\ x &&= RES\ \{x\} \\
  &bind\ FAIL\ f &&= FAIL \\
  &bind\ (RES\ X)\ f &&= Sup\ \{f\,x \mid x.\ x{\in}X\}
  \end{aligned}$$

  where

  $Sup\ X = (\textbf{if}\ FAIL \in X\ \textbf{then}\ FAIL\ \textbf{else}\ RES\ (\bigcup\{Y.\ RES\ Y \in X\}))$

The *nres* Monad

- Combines failure and nondeterminism monad. $M = \mathit{nres}$ where
  $\textbf{datatype}\ {'a}\ \mathit{nres} = \mathit{FAIL} \mid \mathit{RES}\ {'a}\ \mathit{set}$

  $$\begin{aligned}
  \textbf{return}\ x &= \mathit{RES}\ \{x\} \\
  \mathit{bind}\ \mathit{FAIL}\ f &= \mathit{FAIL} \\
  \mathit{bind}\ (\mathit{RES}\ X)\ f &= \mathit{Sup}\ \{f\,x \mid x.\ x{\in}X\}
  \end{aligned}$$

  where

  $\mathit{Sup}\ X = (\textbf{if}\ \mathit{FAIL} \in X\ \textbf{then}\ \mathit{FAIL}\ \textbf{else}\ \mathit{RES}\ (\bigcup\{Y.\ \mathit{RES}\ Y \in X\}))$

- Derived combinators

  $$\begin{aligned}
  \textbf{spec}\ x.\ \Phi\ x &= \mathit{RES}\ \{x.\ \Phi\ x\} \\
  \textbf{assert}\ \Phi &= \textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL} \\
  \textbf{select}\ p.\ \Phi\ p &= \textbf{if}\ \exists x.\ \Phi\ x\ \textbf{then}\ \mathit{RES}\ \{\mathit{Some}\ x \mid x.\ \Phi\ x\}\ \textbf{else return}\ \mathit{None}
  \end{aligned}$$

Structural Recursion

- Generalized fold combinator

  $\mathit{nfoldli}\ []\ c\ f\ s = \textbf{return}\ s$

  $\mathit{nfoldli}\ (x \# \mathit{ls})\ c\ f\ s =$

  $(\textbf{if}\ c\ s\ \textbf{then}\ f\ x\ s \gg\!= (\lambda s.\ \mathit{nfoldli}\ \mathit{ls}\ c\ f\ s)\ \textbf{else return}\ s)$

Structural Recursion

- Generalized fold combinator

  $\mathit{nfoldli}\ []\ c\ f\ s = \textbf{return}\ s$
  $\mathit{nfoldli}\ (x \# ls)\ c\ f\ s =$
  $(\textbf{if}\ c\ s\ \textbf{then}\ f\ x\ s \gg= (\lambda s.\ \mathit{nfoldli}\ ls\ c\ f\ s)\ \textbf{else}\ \textbf{return}\ s)$

- Iteration over set

  $\textbf{foreach}\ S\ f\ \sigma = \textbf{do}\ \{$
  $\quad \textbf{assert}\ (\mathit{finite}\ X);$
  $\quad l \leftarrow \textbf{spec}\ l.\ \mathit{distinct}\ l \wedge S = \mathit{set}\ l;$
  $\quad \mathit{nfoldli}\ l\ (\lambda\_.\ \mathit{True})\ f\ \sigma$
  $\}$

Structural Recursion

- Generalized fold combinator

  $\mathit{nfoldli}\ []\ c\ f\ s = \mathbf{return}\ s$
  $\mathit{nfoldli}\ (x \# \mathit{ls})\ c\ f\ s =$
  $(\mathbf{if}\ c\ s\ \mathbf{then}\ f\ x\ s \gg\!= (\lambda s.\ \mathit{nfoldli}\ \mathit{ls}\ c\ f\ s)\ \mathbf{else\ return}\ s)$

- Iteration over set

  $\mathbf{foreach}\ S\ f\ \sigma = \mathbf{do}\ \{$
  $\quad \mathbf{assert}\ (\mathit{finite}\ X);$
  $\quad l \leftarrow \mathbf{spec}\ l.\ \mathit{distinct}\ l \wedge S = \mathit{set}\ l;$
  $\quad \mathit{nfoldli}\ l\ (\lambda\_.\ \mathit{True})\ f\ \sigma$
  $\}$

  - Warning: Actual implementation of **foreach** and friends suffers from legacy problems. But works nicely at the surface!

Arbitrary Recursion

- Recursion via fixed-point construction

  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$

Arbitrary Recursion

- Recursion via fixed-point construction
  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$
  - Yields *FAIL* if there is a nonterminating execution (total correctness)

Arbitrary Recursion

- Recursion via fixed-point construction
  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$
  - Yields FAIL if there is a nonterminating execution (total correctness)
    - $\mathbf{rec}\ D.\ B\ D$ ignores nonterminating executions (partial correctness)

Arbitrary Recursion

- Recursion via fixed-point construction
  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$
  - Yields FAIL if there is a nonterminating execution (total correctness)
    - $\mathbf{rec}\ D.\ B\ D$ ignores nonterminating executions (partial correctness)
  - Monotonicity of function body follows by construction from monad combinators!

Arbitrary Recursion

- Recursion via fixed-point construction
  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$
  - Yields FAIL if there is a nonterminating execution (total correctness)
    - $\mathbf{rec}\ D.\ B\ D$ ignores nonterminating executions (partial correctness)
  - Monotonicity of function body follows by construction from monad combinators!
  - skipping gory details in this lecture.

Arbitrary Recursion

- Recursion via fixed-point construction
  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$
  - Yields FAIL if there is a nonterminating execution (total correctness)
    - $\mathbf{rec}\ D.\ B\ D$ ignores nonterminating executions (partial correctness)
  - Monotonicity of function body follows by construction from monad combinators!
  - skipping gory details in this lecture.
- From these primitives, define more advanced combinators
  $\mathbf{while}_T\ b\ f \equiv \mathbf{rec}_T\ W.\ (\lambda s.\ \mathbf{if}\ b\ s\ \mathbf{then}\ f\ s \gg\!= W\ \mathbf{else}\ \mathbf{return}\ s)$

Arbitrary Recursion

- Recursion via fixed-point construction

  $\mathit{trimono}\ B \Longrightarrow (\mathbf{rec}_T\ D.\ B\ D) = B\ (\mathbf{rec}_T\ D.\ B\ D)$

  - Yields FAIL if there is a nonterminating execution (total correctness)
    - $\mathbf{rec}\ D.\ B\ D$ ignores nonterminating executions (partial correctness)
  - Monotonicity of function body follows by construction from monad combinators!
  - skipping gory details in this lecture.
- From these primitives, define more advanced combinators

  $\mathbf{while}_T\ b\ f \equiv \mathbf{rec}_T\ W.\ (\lambda s.\ \mathbf{if}\ b\ s\ \mathbf{then}\ f\ s \gg= W\ \mathbf{else\ return}\ s)$
- Program is ordinary term in HOL (shallow embedding)

Examples

Select item from (non-empty) set S :

Examples

Select item from (non-empty) set S : $\textbf{spec}\ x.\ x \in S$

Examples

Select item from (non-empty) set S : $\textbf{spec}\ x.\ x \in S$
Iterate until set is empty, sum up elements

Examples

Select item from (non-empty) set S : $\textbf{spec}\ x.\ x \in S$
Iterate until set is empty, sum up elements

$$
\begin{array}{l}
\mathit{sum\_up}\ S = \textbf{do}\ \{ \\
\quad (S,a) \leftarrow \textbf{while}_T\ (\lambda(S,a).\ S{\neq}\{\})\ (\lambda(S,a).\ \textbf{do}\ \{ \\
\quad\ \ x{\leftarrow}\textbf{spec}\ x.\ x{\in}S; \\
\quad\ \ \textbf{return}\ (S{-}\{x\},a{+}x) \\
\quad \})\ (S,0); \\
\quad \textbf{return}\ a \\
\ \}
\end{array}
$$

Examples

Select item from (non-empty) set S : **spec** $x.\ x \in S$
Iterate until set is empty, sum up elements

sum_up S = **do** {
  $(S,a) \leftarrow$ **while**$_T$ $(\lambda(S,a).\ S \neq \{\})$ $(\lambda(S,a).$ **do** {
    $x \leftarrow$**spec** $x.\ x \in S$;
    return $(S - \{x\}, a + x)$
  }) $(S,0)$;
  return a
}

Only works for finite sets.

Examples

Check if $42 \in S$, S finite. Iterate over S.

Examples

Check if $42 \in S$, S finite. Iterate over S.

$is\_42in\ S = \textbf{do}\ \{$
$\quad (S,f) \leftarrow \textbf{while}_T\ (\lambda(S,f).\ S \neq \{\} \wedge \neg f)\ (\lambda(S,a).\ \textbf{do}\ \{$
$\qquad x \leftarrow \textbf{spec}\ x.\ x \in S;$
$\qquad \textbf{return}\ (S - \{x\}, x = 42)$
$\quad \})\ (S, False);$
$\quad \textbf{return}\ f$
$\}$

Check if $42 \in S$, S finite. Iterate over S.

$$
\begin{array}{l}
\mathit{is\_42in}\ S = \textbf{do}\ \{ \\
\quad (S,f) \leftarrow \textbf{while}_T\ (\lambda(S,f).\ S \neq \{\} \wedge \neg f)\ (\lambda(S,a).\ \textbf{do}\ \{ \\
\quad\quad x \leftarrow \textbf{spec}\ x.\ x \in S; \\
\quad\quad \textbf{return}\ (S - \{x\}, x = 42) \\
\quad \})\ (S, \mathit{False}); \\
\quad \textbf{return}\ f \\
\}
\end{array}
$$

We have emulated a *break* by Boolean flag f.

Edmonds-Karp Algorithm

```
let f = (λ_. 0);
(f,_) ← while_T
  (λ(f,brk). ¬brk)
  (λ(f,_). do {
    p ← select p. Graph.isShortestPath (residualGraph c f) s p t;
    case p of
      None ⇒ return (f,True)
    | Some p ⇒ do {
        let f = NFlow.augment_with_path c f p;
        return (f, False)
      }
  })
  (f,False);
return f
```

Comments

- Locale *Network* fixes flow network

```
locale Network
  fixes c :: nat × nat ⇒ 'capacity
    and s :: nat
    and t :: nat
  assumes Network c s t
```

Comments

- Locale *Network* fixes flow network

  locale *Network*
    fixes $c :: nat \times nat \Rightarrow {}'capacity$
      and $s :: nat$
      and $t :: nat$
    assumes *Network c s t*

- Break from while loop not (yet) supported
  - Using Boolean flag to emulate

Comments

- Locale *Network* fixes flow network

  locale *Network*
    fixes $c :: nat \times nat \Rightarrow {'capacity}$
      and $s :: nat$
      and $t :: nat$
    assumes *Network c s t*

- Break from while loop not (yet) supported
  - Using Boolean flag to emulate
- **select** $p.\ \Phi\ p$: Nondeterministically select value that satisfies Φ

Comments

- Locale *Network* fixes flow network

  locale *Network*
    fixes $c :: nat \times nat \Rightarrow 'capacity$
      and $s :: nat$
      and $t :: nat$
    assumes *Network c s t*

- Break from while loop not (yet) supported
  - Using Boolean flag to emulate
- **select** $p.\ \Phi\ p$: Nondeterministically select value that satisfies Φ
  - Returns *None* if there is no such term

Refinement

- Program m *refines* m', if results of m are also results of m'

  $\_ \leq FAIL$

  $RES\ X \leq RES\ Y\ iff\ X \subseteq Y$

Refinement

- Program m *refines* m', if results of m are also results of m'
  $\_ \leq \mathit{FAIL}$
  $\mathit{RES}\ X \leq \mathit{RES}\ Y\ \mathit{iff}\ X \subseteq Y$
- Special case: correctness of program m wrt. specification Φ

Refinement

- Program m *refines* m', if results of m are also results of m'
  $\_ \leq FAIL$
  $RES\ X \leq RES\ Y \text{ iff } X \subseteq Y$
- Special case: correctness of program m wrt. specification Φ
  - All possible results of m satisfy Φ

Refinement

- Program m *refines* m', if results of m are also results of m'
  $\_ \le FAIL$
  $RES\ X \le RES\ Y$ *iff* $X \subseteq Y$
- Special case: correctness of program m wrt. specification Φ
  - All possible results of m satisfy Φ
  - $m \le RES\ (Collect\ \Phi)$ (notation: $m \le (\textbf{spec}\ x.\ \Phi\ x)$)

Refinement

- Program m *refines* m', if results of m are also results of m'
  $\_ \leq FAIL$
  $RES\ X \leq RES\ Y\ iff\ X \subseteq Y$
- Special case: correctness of program m wrt. specification Φ
  - All possible results of m satisfy Φ
  - $m \leq RES\ (Collect\ \Phi)$ (notation: $m \leq (\textbf{spec}\ x.\ \Phi\ x)$)
- As Hoare-triple $\{P\}\ f\ \{Q\}$
  $P\ x \Longrightarrow f\ x \leq (\textbf{spec}\ r.\ Q\ r)$

Examples

- $qsort :: int\ list \Rightarrow list\ nres$ correct

Examples

- $\mathit{qsort} :: \mathit{int\ list} \Rightarrow \mathit{list\ nres}$ correct
  $\mathit{qsort}\ l \leq (\textbf{spec}\ l'.\ \mathit{mset}\ l' = \mathit{mset}\ l \land \mathit{sorted}\ l')$

Examples

- $\mathit{qsort} :: \mathit{int\ list} \Rightarrow \mathit{list\ nres}$ correct
  $\mathit{qsort}\ l \leq (\textbf{spec}\ l'.\ \mathit{mset}\ l' = \mathit{mset}\ l \wedge \mathit{sorted}\ l')$
- $\mathit{sum\_up}$ correct:

Examples

- $\mathit{qsort} :: \mathit{int\ list} \Rightarrow \mathit{list\ nres}$ correct
  $\mathit{qsort}\ l \le (\mathbf{spec}\ l'.\ \mathit{mset}\ l' = \mathit{mset}\ l \wedge \mathit{sorted}\ l')$
- $\mathit{sum\_up}$ correct:
  $\mathit{finite}\ S \Longrightarrow \mathit{sum\_up}\ S \le (\mathbf{spec}\ a.\ a = \sum S)$

Examples

- $\mathit{qsort} :: \mathit{int\ list} \Rightarrow \mathit{list\ nres}$ correct
  $\mathit{qsort}\ l \leq (\textbf{spec}\ l'.\ \mathit{mset}\ l' = \mathit{mset}\ l \wedge \mathit{sorted}\ l')$
- $\mathit{sum\_up}$ correct:
  $\mathit{finite}\ S \Longrightarrow \mathit{sum\_up}\ S \leq (\textbf{spec}\ a.\ a = \sum S)$
- $\mathit{get\_min} :: \mathit{int\ set} \Rightarrow \mathit{int\ nres}$ correct

Examples

- $\mathit{qsort} :: \mathit{int\ list} \Rightarrow \mathit{list\ nres}$ correct
  $\mathit{qsort}\ l \le (\textbf{spec}\ l'.\ \mathit{mset}\ l' = \mathit{mset}\ l \wedge \mathit{sorted}\ l')$
- $\mathit{sum\_up}$ correct:
  $\mathit{finite}\ S \Longrightarrow \mathit{sum\_up}\ S \le (\textbf{spec}\ a.\ a = \sum S)$
- $\mathit{get\_min} :: \mathit{int\ set} \Rightarrow \mathit{int\ nres}$ correct
  $S \neq \{\} \Longrightarrow \mathit{get\_min}\ S \le (\textbf{spec}\ x.\ x \in S \wedge (\forall y{\in}S.\ x \le y))$

Correctness of Edmonds-Karp

- Prove $\mathit{edmonds\_karp} \leq (\textbf{spec}\ f.\ \mathit{isMaxFlow}\ f)$
  - In Network context (precondition!)

Correctness of Edmonds-Karp

- Prove $\mathit{edmonds\_karp} \leq (\textbf{spec}\ f.\ \mathit{isMaxFlow}\ f)$
  - In Network context (precondition!)
- How to prove such lemmas?

Correctness of Edmonds-Karp

- Prove $\mathit{edmonds\_karp} \le (\textbf{spec}\ f.\ \mathit{isMaxFlow}\ f)$
  - In Network context (precondition!)
- How to prove such lemmas?
- Use verification condition generator!

Reminder: Weakest Preconditions

- Weakest precondition: $wp\ c\ Q$ means: program c terminates with result that satisfies Q

Reminder: Weakest Preconditions

- Weakest precondition: $wp\ c\ Q$ means: program c terminates with result that satisfies Q
    - Nontermination not correct

Reminder: Weakest Preconditions

- Weakest precondition: $wp\ c\ Q$ means: program c terminates with result that satisfies Q
  - Nontermination not correct
- Weakest liberal precondition: $wlp\ (c\ s)\ Q$ means: If program c terminates, then result satisfies Q.

Reminder: Weakest Preconditions

- Weakest precondition: $wp\ c\ Q$ means: program c terminates with result that satisfies Q
  - Nontermination not correct
- Weakest liberal precondition: $wlp\ (c\ s)\ Q$ means: If program c terminates, then result satisfies Q.
  - Nontermination is correct

Reminder: Weakest Preconditions

- Weakest precondition: $wp\ c\ Q$ means: program c terminates with result that satisfies Q
    - Nontermination not correct
- Weakest liberal precondition: $wlp\ (c\ s)\ Q$ means: If program c terminates, then result satisfies Q.
    - Nontermination is correct
- We will use wp here!

Standard Rules for *wp*

- Sequential composition / bind:

  $\mathit{wp}\ m_1\ (\lambda x.\ \mathit{wp}\ (m_2\ x)\ Q) \Longrightarrow \mathit{wp}\ (x \leftarrow m_1;\ m_2)\ Q$

Standard Rules for *wp*

- Sequential composition / bind:

  $wp\ m_1\ (\lambda x.\ wp\ (m_2\ x)\ Q) \Longrightarrow wp\ (x \leftarrow m_1;\ m_2)\ Q$

- If-then-else

  $[\![\ b \Longrightarrow wp\ c_1\ Q;\ \neg b \Longrightarrow wp\ c_2\ Q\ ]\!] \Longrightarrow wp\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ Q$

Standard Rules for *wp*

- Sequential composition / bind:

  $wp\ m_1\ (\lambda x.\ wp\ (m_2\ x)\ Q) \Longrightarrow wp\ (x \leftarrow m_1;\ m_2)\ Q$

- If-then-else

  $[\![\ b \Longrightarrow wp\ c_1\ Q;\ \neg b \Longrightarrow wp\ c_2\ Q\ ]\!] \Longrightarrow wp\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ Q$

- While

Standard Rules for *wp*

- Sequential composition / bind:

  $wp\ m_1\ (\lambda x.\ wp\ (m_2\ x)\ Q) \Longrightarrow wp\ (x \leftarrow m_1;\ m_2)\ Q$

- If-then-else

  $[\![\ b \Longrightarrow wp\ c_1\ Q;\ \neg b \Longrightarrow wp\ c_2\ Q\ ]\!] \Longrightarrow wp\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ Q$

- While

  - Partial correctness:

    $[\![\ I\ s_0;\ \bigwedge s.\ [\![\ b\ s;\ I\ s\ ]\!] \Longrightarrow wlp\ (c\ s)\ I\ ]\!]$
    $\Longrightarrow wlp\ (\mathbf{while}\ b\ c\ s_0)\ (\lambda s.\ I\ s \wedge \neg b\ s)$

Standard Rules for *wp*

- Sequential composition / bind:

  $wp\ m_1\ (\lambda x.\ wp\ (m_2\ x)\ Q) \Longrightarrow wp\ (x \leftarrow m_1;\ m_2)\ Q$

- If-then-else

  $[\![\ b \Longrightarrow wp\ c_1\ Q;\ \neg b \Longrightarrow wp\ c_2\ Q\ ]\!] \Longrightarrow wp\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ Q$

- While

  - Partial correctness:

    $[\![\ I\ s_0;\ \bigwedge s.\ [\![\ b\ s;\ I\ s\ ]\!] \Longrightarrow wlp\ (c\ s)\ I\ ]\!]$
    $\Longrightarrow wlp\ (\mathbf{while}\ b\ c\ s_0)\ (\lambda s.\ I\ s \wedge \neg b\ s)$

  - Total correctness:

    $[\![\ wf <;\ I\ s_0;\ \bigwedge s.\ [\![b\ s;\ I\ s]\!] \Longrightarrow wp\ (c\ s)\ (\lambda s'.\ I\ s' \wedge s' < s)\ ]\!]$
    $\Longrightarrow wp\ (\mathbf{while}\ b\ c\ s_0)\ (\lambda s.\ I\ s \wedge \neg b\ s)$

Standard Rules for *wp*

- Sequential composition / bind:

  $wp\ m_1\ (\lambda x.\ wp\ (m_2\ x)\ Q) \Longrightarrow wp\ (x{\leftarrow}m_1;\ m_2)\ Q$
- If-then-else

  $\llbracket\ b{\Longrightarrow}wp\ c_1\ Q;\ \neg b{\Longrightarrow}wp\ c_2\ Q\ \rrbracket \Longrightarrow wp\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ Q$
- While
  - Partial correctness:

    $\llbracket\ I\ s_0;\ \bigwedge s.\ \llbracket\ b\ s;\ I\ s\ \rrbracket \Longrightarrow wlp\ (c\ s)\ I\ \rrbracket$
    $\Longrightarrow wlp\ (\textbf{while}\ b\ c\ s_0)\ (\lambda s.\ I\ s \wedge \neg b\ s)$
  - Total correctness:

    $\llbracket\ wf\ {<};\ I\ s_0;\ \bigwedge s.\ \llbracket b\ s;\ I\ s\rrbracket \Longrightarrow wp\ (c\ s)\ (\lambda s'.\ I\ s' \wedge s'{<}s)\ \rrbracket$
    $\Longrightarrow wp\ (\textbf{while}\ b\ c\ s_0)\ (\lambda s.\ I\ s \wedge \neg b\ s)$
  - add consequence rule

    $\llbracket\ wf\ {<};\ I\ s_0;\ \bigwedge s.\ \llbracket b\ s;\ I\ s\rrbracket \Longrightarrow wp\ (c\ s)\ (\lambda s'.\ I\ s' \wedge s'{<}s);$
    $\quad \bigwedge s.\ I\ s \wedge \neg b\ s \Longrightarrow Q\ s\ \rrbracket$
    $\Longrightarrow\ wp\ (\textbf{while}\ b\ c\ s_0)\ Q$

Rules for Nres-Monad

$\Phi\ x \Longrightarrow \textbf{return}\ x \leq (\textbf{spec}\ x.\ \Phi\ x)$

Rules for Nres-Monad

$\Phi\ x \Longrightarrow \textbf{return}\ x \leq (\textbf{spec}\ x.\ \Phi\ x)$

$m \leq (\textbf{spec}\ x.\ f\,x \leq (\textbf{spec}\ y.\ \Phi\ y)) \Longrightarrow m \gg\!= (\lambda x.\ f\,x) \leq (\textbf{spec}\ y.\ \Phi\ y)$

Rules for Nres-Monad

$\Phi\ x \Longrightarrow \mathbf{return}\ x \leq (\mathbf{spec}\ x.\ \Phi\ x)$

$m \leq (\mathbf{spec}\ x.\ f\,x \leq (\mathbf{spec}\ y.\ \Phi\ y)) \Longrightarrow m \ggg= (\lambda x.\ f\,x) \leq (\mathbf{spec}\ y.\ \Phi\ y)$

$[\![b \Longrightarrow m_1 \leq (\mathbf{spec}\ x.\ \Phi\ x);\ \neg\ b \Longrightarrow m_2 \leq (\mathbf{spec}\ x.\ \Phi\ x)]\!]$
$\Longrightarrow (\mathbf{if}\ b\ \mathbf{then}\ m_1\ \mathbf{else}\ m_2) \leq (\mathbf{spec}\ x.\ \Phi\ x)$

Rules for Nres-Monad

$\Phi\ x \Longrightarrow \textbf{return}\ x \leq (\textbf{spec}\ x.\ \Phi\ x)$

$m \leq (\textbf{spec}\ x.\ f\ x \leq (\textbf{spec}\ y.\ \Phi\ y)) \Longrightarrow m \ggg (\lambda x.\ f\ x) \leq (\textbf{spec}\ y.\ \Phi\ y)$

$[\![b \Longrightarrow m_1 \leq (\textbf{spec}\ x.\ \Phi\ x);\ \neg\ b \Longrightarrow m_2 \leq (\textbf{spec}\ x.\ \Phi\ x)]\!]$
$\Longrightarrow (\textbf{if}\ b\ \textbf{then}\ m_1\ \textbf{else}\ m_2) \leq (\textbf{spec}\ x.\ \Phi\ x)$

$[\![wf\ R;\ I\ s;\ \bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq (\textbf{spec}\ s'.\ I\ s' \wedge (s',\ s) \in R);$
$\bigwedge s.\ [\![I\ s;\ \neg\ b\ s]\!] \Longrightarrow \Phi\ s]\!]$
$\Longrightarrow \textbf{while}_T\ b\ f\ s \leq (\textbf{spec}\ s.\ \Phi\ s)$

Rules for Nres-Monad

$$\Phi\ x \Longrightarrow \textbf{return}\ x \leq (\textbf{spec}\ x.\ \Phi\ x)$$

$$m \leq (\textbf{spec}\ x.\ f\,x \leq (\textbf{spec}\ y.\ \Phi\ y)) \Longrightarrow m \ggg\!= (\lambda x.\ f\,x) \leq (\textbf{spec}\ y.\ \Phi\ y)$$

$$[\![b \Longrightarrow m_1 \leq (\textbf{spec}\ x.\ \Phi\ x);\ \neg\ b \Longrightarrow m_2 \leq (\textbf{spec}\ x.\ \Phi\ x)]\!]$$
$$\Longrightarrow (\textbf{if}\ b\ \textbf{then}\ m_1\ \textbf{else}\ m_2) \leq (\textbf{spec}\ x.\ \Phi\ x)$$

$$[\![wf\ R;\ I\ s;\ \bigwedge s.\ [\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq (\textbf{spec}\ s'.\ I\ s' \wedge (s', s) \in R);$$
$$\bigwedge s.\ [\![I\ s;\ \neg\ b\ s]\!] \Longrightarrow \Phi\ s]\!]$$
$$\Longrightarrow \textbf{while}_T\ b\ f\ s \leq (\textbf{spec}\ s.\ \Phi\ s)$$

...

Verification Condition Generator

- Apply rules repeatedly.

Verification Condition Generator

- Apply rules repeatedly.
- Rule to be applied determined by topmost statement
  - Automatically!
  - May have to apply consequence rule before
    $[\![m \leq (\textbf{spec}\ x.\ \Phi\ x);\ \bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x]\!] \Longrightarrow m \leq (\textbf{spec}\ x.\ \Psi\ x)$

Verification Condition Generator

- Apply rules repeatedly.
- Rule to be applied determined by topmost statement
  - Automatically!
  - May have to apply consequence rule before
    $\llbracket m \leq (\textbf{spec}\ x.\ \Phi\ x);\ \bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x \rrbracket \Longrightarrow m \leq (\textbf{spec}\ x.\ \Psi\ x)$
- Stop when no rule applies
  - Subgoal is not of shape $\_ \leq \textbf{spec}\ \_.\ \_$ (cf. *RETURN_rule*)
  - Missing rule, e.g. for user-defined function

Verification Condition Generator

- Apply rules repeatedly.
- Rule to be applied determined by topmost statement
  - Automatically!
  - May have to apply consequence rule before
    $[\![m \leq (\textbf{spec}\ x.\ \Phi\ x);\ \bigwedge x.\ \Phi\ x \Longrightarrow \Psi\ x]\!] \Longrightarrow m \leq (\textbf{spec}\ x.\ \Psi\ x)$
- Stop when no rule applies
  - Subgoal is not of shape $\_ \leq \textbf{spec}\ \_.\ \_$ (cf. *RETURN_rule*)
  - Missing rule, e.g. for user-defined function
- Prove the generated VCs
  - Using Isabelle's standard proof methods

Invariants

- Finding good invariant is usually most creative task

Invariants

- Finding good invariant is usually most creative task
- Recall: Invariant for while-loop must

Invariants

- Finding good invariant is usually most creative task
- Recall: Invariant for while-loop must
  - Hold initially ($I\ s_0$)

Invariants

- Finding good invariant is usually most creative task
- Recall: Invariant for while-loop must
  - Hold initially ($I\ s_0$)
  - Be preserved by loop iteration ($[\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq \textbf{spec}\ I$)

Invariants

- Finding good invariant is usually most creative task
- Recall: Invariant for while-loop must
  - Hold initially ($I\ s_0$)
  - Be preserved by loop iteration ($[\![I\ s;\ b\ s]\!] \Longrightarrow f\ s \leq \mathbf{spec}\ I$)
  - Imply postcondition when loop terminates ($[\![I\ s;\ \neg b\ s]\!] \Longrightarrow \Phi\ s$)

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant:

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0\ -\ S)$

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0 - S)$ sufficient?

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S$; $S{=}S{-}\{x\}$; $a{=}a{+}x$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0 - S) \qquad\qquad \wedge\ S{\subseteq}S_0$

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum(S_0 - S) \qquad\qquad \wedge\ S{\subseteq}S_0$
- $f{=}\mathit{False}$; **while** $\neg f$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ f{=}(x{=}{=}42)\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow f = 42{\in}S_0$
  Invariant:

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum(S_0\ -\ S) \qquad\qquad \wedge\ S{\subseteq}S_0$
- $f{=}\mathit{False}$; **while** $\neg f$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ f{=}(x{=}{=}42)\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow f = 42{\in}S_0$
  Invariant: $f = x{\in}(S_0{-}S)\ \wedge\ S{\subseteq}S_0$

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ a{=}a{+}x\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0\ -\ S) \qquad\qquad \wedge\ S{\subseteq}S_0$
- $f{=}\mathit{False}$; **while** $\neg f$ **do** $\{\ x{\leftarrow}$**spec** $x.\ x{\in}S;\ S{=}S{-}\{x\};\ f{=}(x{==}42)\ \}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow f = 42{\in}S_0$
  Invariant: $f = x{\in}(S_0{-}S)\ \wedge\ S{\subseteq}S_0$
- **while** $a{\neq}b$ **do** $\{$**if** $a{<}b$ **then** $b{=}b{-}a$ **else** $a{=}a{-}b\}$
  Specification: $[\![a_0{>}0;\ b_0{>}0]\!] \Longrightarrow a = \mathit{gcd}\ a_0\ b_0$
  Invariant:

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S$; $S{=}S{-}\{x\}$; $a{=}a{+}x$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0 - S) \qquad\qquad \wedge\ S{\subseteq}S_0$
- $f{=}\mathit{False}$; **while** $\neg f$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S$; $S{=}S{-}\{x\}$; $f{=}(x{==}42)$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow f = 42{\in}S_0$
  Invariant: $f = x{\in}(S_0{-}S) \wedge S{\subseteq}S_0$
- **while** $a{\neq}b$ **do** $\{$**if** $a{<}b$ **then** $b{=}b{-}a$ **else** $a{=}a{-}b\}$
  Specification: $[\![a_0{>}0;\ b_0{>}0]\!] \Longrightarrow a = \mathit{gcd}\ a_0\ b_0$
  Invariant: $\mathit{gcd}\ a\ b = \mathit{gcd}\ a_0\ b_0$

Invariant Examples

Note: informal syntax!

- $a{=}0$; **while** $S{\neq}\{\}$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S$; $S{=}S{-}\{x\}$; $a{=}a{+}x$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow a = \sum S_0$
  Invariant: $a = \sum (S_0 - S) \quad\quad \wedge\ S{\subseteq}S_0$
- $f{=}\mathit{False}$; **while** $\neg f$ **do** $\{$ $x{\leftarrow}$**spec** $x.\ x{\in}S$; $S{=}S{-}\{x\}$; $f{=}(x{==}42)$ $\}$
  Specification: $\mathit{finite}\ S_0 \Longrightarrow f = 42{\in}S_0$
  Invariant: $f = x{\in}(S_0{-}S) \wedge S{\subseteq}S_0$
- **while** $a{\neq}b$ **do** $\{$**if** $a{<}b$ **then** $b{=}b{-}a$ **else** $a{=}a{-}b\}$
  Specification: $[\![a_0{>}0;\ b_0{>}0]\!] \Longrightarrow a = \mathit{gcd}\ a_0\ b_0$
  Invariant: $\mathit{gcd}\ a\ b = \mathit{gcd}\ a_0\ b_0\ \wedge a{>}0 \wedge b{>}0$

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals
  - *auto*, *simp*, ... Try to solve, present unsolvable parts

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals
  - *auto*, *simp*, ... Try to solve, present unsolvable parts
- Other useful tools

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals
  - *auto*, *simp*, ... Try to solve, present unsolvable parts
- Other useful tools
  - *sledgehammer* Run SMT solvers, replay proof in Isabelle kernel

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals
  - *auto*, *simp*, . . . Try to solve, present unsolvable parts
- Other useful tools
  - *sledgehammer* Run SMT solvers, replay proof in Isabelle kernel
  - *quickcheck*, *nitpick* Find counterexamples

Backwards Verification in Isabelle

- Prove lemma: discharge subgoals
  - Initially: One subgoal, proposition of the lemma
  - Apply rules, which may discharge/produce subgoals
- Rule $[\![P_1 ; \ldots ; P_n]\!] \Longrightarrow Q$
  - Replace subgoal Q with new subgoals P_1, ..., P_n (unification!)
- Other proof methods to work on subgoals
  - *auto*, *simp*, ... Try to solve, present unsolvable parts
- Other useful tools
  - *sledgehammer* Run SMT solvers, replay proof in Isabelle kernel
  - *quickcheck*, *nitpick* Find counterexamples
- No subgoals left: lemma proved!

Simple_Invar_Demo.thy

Simple Invariants

`Edka_Abstract_Demo.thy`

Proving Correctness of Abstract Edmonds-Karp Algorithm

Conclusions (so far)

- Prove correctness of abstract algorithm first
  - can be modeled in nres-monad, shallowly embedded in HOL
- Proof by VCG + abstract theorems from background theory
  - VCG is almost automatic
  - Background theory can require considerable manual work

A Complete Example: State-Space Search

- Given directed edges E and a start node s, compute the set of reachable nodes

workset $W = \{s\}$; *visited set* $V = \{\}$

while $W \neq \{\}$ **do**
  remove some node u from W
  if $u \notin V$ **then**
    $V = V \cup \{u\}$
    $W = W \cup \{\ v.\ (u,v) \in E\ \}$

return V

A Complete Example: State-Space Search

- Given directed edges E and a start node s, compute the set of reachable nodes

  workset $W = \{s\}$; *visited set* $V = \{\}$

  while $W \neq \{\}$ **do**
  remove some node u from W
  if $u \notin V$ **then**
  $V = V \cup \{u\}$
  $W = W \cup \{ v.\ (u,v) \in E \}$

  return V

- BFS, DFS, Best-First, ... are instances of this generic scheme!

Correctness

workset $W = \{s\}$; *visited set* $V = \{\}$

while $W \neq \{\}$ **do**
  remove some node u from W
  if $u \notin V$ **then**
    $V = V \cup \{u\}$
    $W = W \cup \{\ v.\ (u,v) \in E\ \}$

return V

- Clearly, only reachable nodes are added to W or V
  - $V \subseteq$ *reachable* at end of loop

Correctness

workset $W = \{s\}$; *visited set* $V = \{\}$

while $W \neq \{\}$ **do**
  remove some node u from W
  if $u \notin V$ **then**
    $V = V \cup \{u\}$
    $W = W \cup \{\ v.\ (u,v) \in E\ \}$

return V

- Clearly, only reachable nodes are added to W or V
    - $V \subseteq \mathit{reachable}$ at end of loop
- Outgoing edges from V always end in $W \cup V$ (search frontier)

Correctness

workset $W = \{s\}$; *visited set* $V = \{\}$

while $W \neq \{\}$ **do**
  remove some node u from W
  if $u \notin V$ **then**
    $V = V \cup \{u\}$
    $W = W \cup \{\ v.\ (u,v) \in E\ \}$

return V

- Clearly, only reachable nodes are added to W or V
  - $V \subseteq$ *reachable* at end of loop
- Outgoing edges from V always end in $W \cup V$ (search frontier)
- Finally, $W = \{\}$. Thus V closed under edges,
  - As start node is in V upon termination, we get $V \supseteq$ *reachable*

Correctness

workset $W = \{s\}$; *visited set* $V = \{\}$

while $W \neq \{\}$ **do**
  remove some node u *from* W
  if $u \notin V$ **then**
    $V = V \cup \{u\}$
    $W = W \cup \{\, v.\ (u,v) \in E \,\}$

return V

- Clearly, only reachable nodes are added to W or V
  - $V \subseteq \mathit{reachable}$ at end of loop
- Outgoing edges from V always end in $W \cup V$ (search frontier)
- Finally, $W = \{\}$. Thus V closed under edges,
  - As start node is in V upon termination, we get $V \supseteq \mathit{reachable}$
- Termination: Only if set of reachable nodes is finite
  - $V \subseteq \mathit{reachable}$ increases, or V remains unchanged and $W \subseteq \mathit{reachable}$ decreases.

`Workset_Demo.thy`

Proving Correctness of State-Space Search

Structural Refinement

- Combinators of nres-monad are monotonic

Structural Refinement

- Combinators of nres-monad are monotonic
- If we have $m_1 \leq m_2$, we can replace m_1 by m_2 in any context

Structural Refinement

- Combinators of nres-monad are monotonic
- If we have $m_1 \leq m_2$, we can replace m_1 by m_2 in any context
- Eg. $BFS\ g\ s\ t \leq \textbf{select}\ p.\ Graph.isShortestPath\ g\ s\ p\ t$

Structural Refinement

- Combinators of nres-monad are monotonic
- If we have $m_1 \le m_2$, we can replace m_1 by m_2 in any context
- Eg. $\mathit{BFS}\ g\ s\ t \le \textbf{select}\ p.\ \mathit{Graph.isShortestPath}\ g\ s\ p\ t$
  - Note: $(\textbf{select}\ p.\ \Phi\ p) = (\textbf{spec}\ r.\ \textbf{case}\ r\ \textbf{of}\ \mathit{None} \Rightarrow \nexists\, p.\ \Phi\ p \mid \mathit{Some}\ p \Rightarrow \Phi\ p)$

Structural Refinement

- Combinators of nres-monad are monotonic
- If we have $m_1 \leq m_2$, we can replace m_1 by m_2 in any context
- Eg. $\mathit{BFS}\ g\ s\ t \leq \textbf{select}\ p.\ \mathit{Graph.isShortestPath}\ g\ s\ p\ t$
  - Note: $(\textbf{select}\ p.\ \Phi\ p) = (\textbf{spec}\ r.\ \textbf{case}\ r\ \textbf{of}\ \mathit{None} \Rightarrow \nexists\, p.\ \Phi\ p \mid \mathit{Some}\ p \Rightarrow \Phi\ p)$
- That's easy!

Structural Refinement

- Combinators of nres-monad are monotonic
- If we have $m_1 \leq m_2$, we can replace m_1 by m_2 in any context
- Eg. $\mathit{BFS}\ g\ s\ t \leq \textbf{select}\ p.\ \mathit{Graph.isShortestPath}\ g\ s\ p\ t$
  - Note: $(\textbf{select}\ p.\ \Phi\ p) = (\textbf{spec}\ r.\ \textbf{case}\ r\ \textbf{of}\ \mathit{None} \Rightarrow \nexists\, p.\ \Phi\ p \mid \mathit{Some}\ p \Rightarrow \Phi\ p)$
- That's easy! But what if data representation changes?

Using Residual Graph

- Our current *edmonds_karp* computes residual graph in each iteration

  $p \leftarrow \textbf{select}\ p.\ Graph.isShortestPath\ (residualGraph\ c\ f)\ s\ p\ t$

Using Residual Graph

- Our current *edmonds_karp* computes residual graph in each iteration

  $p \leftarrow \textbf{select}\ p.\ \mathit{Graph.isShortestPath}\ (\mathit{residualGraph}\ c\ f)\ s\ p\ t$

- Nice for correctness proof.

Using Residual Graph

- Our current *edmonds_karp* computes residual graph in each iteration

  $p \leftarrow \textbf{select}\ p.\ \mathit{Graph.isShortestPath}\ (\mathit{residualGraph}\ c\ f)\ s\ p\ t$

- Nice for correctness proof. But very inefficient!

Using Residual Graph

- Our current *edmonds_karp* computes residual graph in each iteration

  $p \leftarrow$ **select** $p.$ *Graph.isShortestPath* (*residualGraph c f*) *s p t*

- Nice for correctness proof. But very inefficient!
- Instead of flow, we can update residual graph

Using Residual Graph

- Our current *edmonds_karp* computes residual graph in each iteration

  $p \leftarrow$ **select** *p. Graph.isShortestPath (residualGraph c f) s p t*

- Nice for correctness proof. But very inefficient!
- Instead of flow, we can update residual graph
  - Upon termination: compute flow from residual graph

Refined Algorithm

Original:

```
let f = (λ_. 0);
(f,_) ← while_T
  (λ(f,brk). ¬brk)
  (λ(f,_). do {
    p ← select p. Graph.isShortestPath (residualGraph c f) s p t;
    case p of
      None ⇒ return (f,True)
    | Some p ⇒ do {
        let f = NFlow.augment_with_path c f p;
        return (f, False)
      }
  })
  (f,False);
return f
```

Refined Algorithm

Refined:

$\textbf{let } \mathit{cf} = c;$
$(\mathit{cf},\_) \leftarrow \textbf{while}_T$
$\quad (\lambda(\mathit{cf},\mathit{brk}).\ \neg \mathit{brk})$
$\quad (\lambda(\mathit{cf},\_).\ \textbf{do}\ \{$
$\quad\quad p \leftarrow \textbf{select}\ p.\ \mathit{Graph.isShortestPath\ cf\ s\ p\ t};$
$\quad\quad \textbf{case}\ p\ \textbf{of}$
$\quad\quad\quad \mathit{None} \Rightarrow \textbf{return}\ (\mathit{cf},\mathit{True})$
$\quad\quad \mid \mathit{Some}\ p \Rightarrow \textbf{do}\ \{$
$\quad\quad\quad\quad \textbf{let}\ \mathit{cf} = \mathit{Graph.augment\_cf\ cf}\ (\mathit{set}\ p)\ (\mathit{resCap\_cf\ cf\ p});$
$\quad\quad\quad\quad \textbf{return}\ (\mathit{cf},\ \mathit{False})$
$\quad\quad\quad \}$
$\quad \})$
$\quad (\mathit{cf},\mathit{False});$
$\textbf{return}\ (\mathit{flow\_of\_cf\ cf})$

Correctness

- How to prove this correct?

Correctness

- How to prove this correct? w/o repeating abstract proof!

Correctness

- How to prove this correct? w/o repeating abstract proof!
- Relate *edmonds_karp2* to *edmonds_karp*

Correctness

- How to prove this correct? w/o repeating abstract proof!
- Relate *edmonds_karp2* to *edmonds_karp*
- It's the same algorithm!
    - only flow has been exchanged by residual graph

Data Refinement

- Relate residual graph and flow

Data Refinement

- Relate residual graph and flow
- Show that operations on residual graph and flow are consistent

Data Refinement

- Relate residual graph and flow
- Show that operations on residual graph and flow are consistent
- Use structural rules to infer relation between programs

Relation between Residual Graph and Flow

- $\mathit{flow\_of\_cf}::(\mathit{nat} \times \mathit{nat} \Rightarrow {}'\mathit{capacity}) \Rightarrow \mathit{nat} \times \mathit{nat} \Rightarrow {}'\mathit{capacity}$
  convert residual graph to flow

Relation between Residual Graph and Flow

- $\mathit{flow\_of\_cf}::(\mathit{nat} \times \mathit{nat} \Rightarrow {'\mathit{capacity}}) \Rightarrow \mathit{nat} \times \mathit{nat} \Rightarrow {'\mathit{capacity}}$
  convert residual graph to flow
- $\mathit{cfi\_rel} = \{(\mathit{cf}, \mathit{flow\_of\_cf}\ \mathit{cf})\ |\mathit{cf}.\ \mathit{RGraph}\ c\ s\ t\ \mathit{cf}\}$

Relation between Residual Graph and Flow

- $\textit{flow\_of\_cf} :: (\textit{nat} \times \textit{nat} \Rightarrow {'}\textit{capacity}) \Rightarrow \textit{nat} \times \textit{nat} \Rightarrow {'}\textit{capacity}$
  convert residual graph to flow
- $\textit{cfi\_rel} = \{(\textit{cf}, \textit{flow\_of\_cf cf}) \mid \textit{cf}.\ \textit{RGraph c s t cf}\}$
  - Relation consists of
    abstraction function (*flow_of_cf*)
    and *invariant* (*RGraph c s t*)

Relation between Residual Graph and Flow

- $\mathit{flow\_of\_cf} :: (\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{'capacity}) \Rightarrow \mathit{nat} \times \mathit{nat} \Rightarrow \mathit{'capacity}$
  convert residual graph to flow
- $\mathit{cfi\_rel} = \{(\mathit{cf}, \mathit{flow\_of\_cf}\ \mathit{cf}) \mid \mathit{cf}.\ \mathit{RGraph}\ c\ s\ t\ \mathit{cf}\}$
  - Relation consists of
    abstraction function (*flow_of_cf*)
    and *invariant* (*RGraph c s t*)
  - This pattern occurs frequently. Shortcut:
    $\mathit{cfi\_rel} \equiv \mathit{br}\ \mathit{flow\_of\_cf}\ (\mathit{RGraph}\ c\ s\ t)$

Relating Operations

- Show, for each operation: inputs related $\Longrightarrow$ outputs related

Relating Operations

- Show, for each operation: inputs related $\Longrightarrow$ outputs related
- Initial flow: Straightforward

  $(c,\ \lambda\_.\ 0{::}'capacity) \in cfi\_rel$

Relating Operations

- Show, for each operation: inputs related $\Longrightarrow$ outputs related
- Initial flow: Straightforward

  $(c,\ \lambda\_.\ 0::'capacity) \in cfi\_rel$

- Augmentation of flow:

  $[\![(cf,\ f) \in cfi\_rel;\ NPreflow.isAugmentingPath\ c\ s\ t\ f\ p]\!]$
  $\Longrightarrow (Graph.augment\_cf\ cf\ (set\ p)\ (resCap\_cf\ cf\ p),$
  $\quad NFlow.augment\_with\_path\ c\ f\ p)$
  $\quad \in cfi\_rel$

Relating Operations

- Show, for each operation: inputs related $\Longrightarrow$ outputs related
- Initial flow: Straightforward

  $(c, \lambda\_.\ 0{::}'capacity) \in cfi\_rel$

- Augmentation of flow:

  $[\![(cf, f) \in cfi\_rel;\ NPreflow.isAugmentingPath\ c\ s\ t\ f\ p]\!]$
  $\Longrightarrow (Graph.augment\_cf\ cf\ (set\ p)\ (resCap\_cf\ cf\ p),$
  $\quad NFlow.augment\_with\_path\ c\ f\ p)$
  $\quad \in cfi\_rel$

  - Note the *isAugmentingPath* precondition!

Relating Operations

- Show, for each operation: inputs related $\Longrightarrow$ outputs related
- Initial flow: Straightforward

  $(c, \lambda\_.\ 0{::}'capacity) \in cfi\_rel$

- Augmentation of flow:

  $[\![(cf, f) \in cfi\_rel;\ NPreflow.isAugmentingPath\ c\ s\ t\ f\ p]\!]$
  $\Longrightarrow (Graph.augment\_cf\ cf\ (set\ p)\ (resCap\_cf\ cf\ p),$
  $NFlow.augment\_with\_path\ c\ f\ p)$
  $\in cfi\_rel$

  - Note the *isAugmentingPath* precondition!
  - Will later show how to take care of

Relating Programs

- Representation of program result can also change

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph
- We need to lift relation on result types to relation on *nres*

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph
- We need to lift relation on result types to relation on *nres*
  - Each concrete result related to some abstract result

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph
- We need to lift relation on result types to relation on *nres*
  - Each concrete result related to some abstract result
  - Given relation $R :: ('c,'a)\ set$, and $m_1 :: 'c\ nres$, $m_2 :: 'a\ nres$

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph
- We need to lift relation on result types to relation on *nres*
  - Each concrete result related to some abstract result
  - Given relation $R :: ('c,'a)\ set$, and $m_1 :: 'c\ nres$, $m_2 :: 'a\ nres$
  - m_1 related to m_2, if
    - $m_2 = FAIL$, or
    - $m_1 = RES\ X$, $m_2 = RES\ Y$, and $\forall x \in X.\ \exists y \in Y.\ (x,y) \in R$

Relating Programs

- Representation of program result can also change
  - E.g, while loop: flow $\rightarrow$ residual graph
- We need to lift relation on result types to relation on *nres*
  - Each concrete result related to some abstract result
  - Given relation $R :: ('c,'a)\ set$, and $m_1::'c\ nres$, $m_2::'a\ nres$
  - m_1 related to m_2, if
    - $m_2 = FAIL$, or
    - $m_1 = RES\ X$, $m_2 = RES\ Y$, and $\forall x{\in}X.\ \exists y{\in}Y.\ (x,y){\in}R$
- Can be expressed as refinement
  - $m_1 \le \Downarrow R\ m_2$, where

    $\Downarrow R\ FAIL = FAIL$

    $\Downarrow R\ (RES\ Y) = (\textbf{spec}\ x.\ \exists y{\in}Y.\ (x,\ y) \in R)$

Proving Relation Between Programs

- Combinators are parametric
  - Control flow doesn't care about the data
  - as long as conditions evaluate the same

Proving Relation Between Programs

- Combinators are parametric
  - Control flow doesn't care about the data
  - as long as conditions evaluate the same
- Some rules

$(x_1, x_2) \in R \Longrightarrow \textbf{return}\ x_1 \leq \Downarrow R\ (\textbf{return}\ x_2)$

Proving Relation Between Programs

- Combinators are parametric
  - Control flow doesn't care about the data
  - as long as conditions evaluate the same
- Some rules

  $(x_1, x_2) \in R \Longrightarrow \textbf{return}\ x_1 \leq \Downarrow R\ (\textbf{return}\ x_2)$

  $[\![m_1 \leq \Downarrow R\ m_2;\ \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2)]\!]$
  $\Longrightarrow m_1 \gg\!= f_1 \leq \Downarrow R'\ (m_2 \gg\!= f_2)$

Proving Relation Between Programs

- Combinators are parametric
  - Control flow doesn't care about the data
  - as long as conditions evaluate the same
- Some rules

$(x_1, x_2) \in R \Longrightarrow \textbf{return}\ x_1 \leq \Downarrow R\ (\textbf{return}\ x_2)$

$[\![m_1 \leq \Downarrow R\ m_2; \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2)]\!]$
$\Longrightarrow m_1 \gg\!= f_1 \leq \Downarrow R'\ (m_2 \gg\!= f_2)$

$[\![(x, x') \in R; \bigwedge x\ x'.\ (x, x') \in R \Longrightarrow b\ x = b'\ x';$
$\bigwedge x\ x'.\ [\![(x, x') \in R; b\ x; b'\ x']\!] \Longrightarrow f\ x \leq \Downarrow R\ (f'\ x')]\!]$
$\Longrightarrow \textbf{while}\ b\ f\ x \leq \Downarrow R\ (\textbf{while}\ b'\ f'\ x')$

- Prove $m_1 \leq\Downarrow R\ m_2$, where m_1 and m_2 have same structure

Automation

- Prove $m_1 \leq\Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators

Automation

- Prove $m_1 \leq\Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators
  - Remaining goals are refinement of operations

Automation

- Prove $m_1 \leq\Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators
  - Remaining goals are refinement of operations
- Additional challenges in practice

Automation

- Prove $m_1 \leq\Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators
  - Remaining goals are refinement of operations
- Additional challenges in practice
  - Unknown relations

Automation

- Prove $m_1 \leq \Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators
  - Remaining goals are refinement of operations
- Additional challenges in practice
  - Unknown relations
  - Side conditions

Automation

- Prove $m_1 \leq \Downarrow R\ m_2$, where m_1 and m_2 have same structure
- Repeatedly apply rules for combinators
  - Remaining goals are refinement of operations
- Additional challenges in practice
  - Unknown relations
  - Side conditions
  - Program structure only almost equal

Unknown Relations

- Recall bind-refine rule

  $[\![m_1 \le \Downarrow R\ m_2;\ \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \le \Downarrow R'\ (f_2\ x_2)]\!]$
  $\Longrightarrow m_1 \gg\!= f_1 \le \Downarrow R'\ (m_2 \gg\!= f_2)$

Unknown Relations

- Recall bind-refine rule

  $[\![m_1 \leq \Downarrow R\ m_2;\ \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2)]\!]$
  $\Longrightarrow m_1 \gg\!= f_1 \leq \Downarrow R'\ (m_2 \gg\!= f_2)$

- What relation R should we choose?

Unknown Relations

- Recall bind-refine rule

  $[\![m_1 \leq \Downarrow R\ m_2; \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2)]\!]$
  $\Longrightarrow m_1 \gg\!= f_1 \leq \Downarrow R'\ (m_2 \gg\!= f_2)$
- What relation R should we choose?
  - We don't know!

Unknown Relations

- Recall bind-refine rule
  $\llbracket m_1 \leq \Downarrow R\ m_2; \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2) \rrbracket$
  $\Longrightarrow m_1 \ggg f_1 \leq \Downarrow R'\ (m_2 \ggg f_2)$
- What relation R should we choose?
  - We don't know! (yet)

Unknown Relations

- Recall bind-refine rule

  $[\![m_1 \leq \Downarrow R\ m_2;\ \bigwedge x_1\ x_2.\ (x_1, x_2) \in R \Longrightarrow f_1\ x_1 \leq \Downarrow R'\ (f_2\ x_2)]\!]$
  $\Longrightarrow m_1 \ggg f_1 \leq \Downarrow R'\ (m_2 \ggg f_2)$
- What relation R should we choose?
  - We don't know! (yet)
- In practice: Guess relation from type

Side Conditions

- Recall theorem for augmentation refinement

$[\![(cf, f) \in \mathit{cfi\_rel}; \mathit{NPreflow.isAugmentingPath}\ c\ s\ t\ f\ p]\!]$
$\Longrightarrow (\mathit{Graph.augment\_cf}\ cf\ (\mathit{set}\ p)\ (\mathit{resCap\_cf}\ cf\ p),$
$\quad \mathit{NFlow.augment\_with\_path}\ c\ f\ p)$
$\quad \in \mathit{cfi\_rel}$

Side Conditions

- Recall theorem for augmentation refinement

  $\llbracket (\mathit{cf}, f) \in \mathit{cfi\_rel};\ \mathit{NPreflow.isAugmentingPath}\ c\ s\ t\ f\ p \rrbracket$
  $\Longrightarrow (\mathit{Graph.augment\_cf}\ \mathit{cf}\ (\mathit{set}\ p)\ (\mathit{resCap\_cf}\ \mathit{cf}\ p),$
  $\quad \mathit{NFlow.augment\_with\_path}\ c\ f\ p)$
  $\quad \in \mathit{cfi\_rel}$

- We do not refine paths at all

Side Conditions

- Recall theorem for augmentation refinement

  $[\![(\mathit{cf}, f) \in \mathit{cfi\_rel};\ \mathit{NPreflow.isAugmentingPath}\ c\ s\ t\ f\ p]\!]$
  $\Longrightarrow (\mathit{Graph.augment\_cf}\ \mathit{cf}\ (\mathit{set}\ p)\ (\mathit{resCap\_cf}\ \mathit{cf}\ p),$
  $\quad \mathit{NFlow.augment\_with\_path}\ c\ f\ p)$
  $\quad \in \mathit{cfi\_rel}$

- We do not refine paths at all
  - All we have is $(p,p) \in \mathit{Id}$!

Side Conditions

- Recall theorem for augmentation refinement

  $[\![(\mathit{cf}, f) \in \mathit{cfi\_rel}; \mathit{NPreflow.isAugmentingPath}\ c\ s\ t\ f\ p]\!]$
  $\Longrightarrow (\mathit{Graph.augment\_cf}\ \mathit{cf}\ (\mathit{set}\ p)\ (\mathit{resCap\_cf}\ \mathit{cf}\ p),$
  $\quad \mathit{NFlow.augment\_with\_path}\ c\ f\ p)$
  $\quad \in \mathit{cfi\_rel}$

- We do not refine paths at all
  - All we have is $(p,p) \in \mathit{Id}$!
- Solution: assertions and congruence rules

Assertions

- **assert** $\Phi =$ (**if** Φ **then return** () **else** *FAIL*)

Assertions

- **assert** $\Phi = ($**if** Φ **then return** $()$ **else** *FAIL*$)$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow$ **assert** $\Phi \leq$ *SPEC* Ψ

Assertions

- $\textbf{assert}\ \Phi = (\textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL})$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow \textbf{assert}\ \Phi \leq \mathit{SPEC}\ \Psi$
  - Insert $\textbf{assert}\ (\mathit{isAugmentingPath}\ c\ s\ t\ f\ p)$ to abstract algorithm

Assertions

- **assert** $\Phi = (\textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL})$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow \textbf{assert}\ \Phi \leq \mathit{SPEC}\ \Psi$
  - Insert **assert** $(\mathit{isAugmentingPath}\ c\ s\ t\ f\ p)$ to abstract algorithm
    - Easy to prove there!

Assertions

- $\textbf{assert}\ \Phi = (\textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL})$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow \textbf{assert}\ \Phi \leq \mathit{SPEC}\ \Psi$
  - Insert $\textbf{assert}\ (\mathit{isAugmentingPath}\ c\ s\ t\ f\ p)$ to abstract algorithm
    - Easy to prove there!
- $(\Phi \Longrightarrow m_1 \leq \Downarrow R\ m_2) \Longrightarrow m_1 \leq \Downarrow R\ (\mathit{assert}\ \Phi \gg\!= (\lambda\_.\ m_2))$

Assertions

- $\textbf{assert}\ \Phi = (\textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL})$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow \textbf{assert}\ \Phi \leq \mathit{SPEC}\ \Psi$
  - Insert $\textbf{assert}\ (\mathit{isAugmentingPath}\ c\ s\ t\ f\ p)$ to abstract algorithm
    - Easy to prove there!
- $(\Phi \Longrightarrow m_1 \leq \Downarrow R\ m_2) \Longrightarrow m_1 \leq \Downarrow R\ (\mathit{assert}\ \Phi \gg\!= (\lambda\_.\ m_2))$
  - During refinement, we can assume Φ

Assertions

- $\textbf{assert}\ \Phi = (\textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ \mathit{FAIL})$
  - $[\![\Phi; \Phi \Longrightarrow \Psi\ ()]\!] \Longrightarrow \textbf{assert}\ \Phi \leq \mathit{SPEC}\ \Psi$
  - Insert $\textbf{assert}\ (\mathit{isAugmentingPath}\ c\ s\ t\ f\ p)$ to abstract algorithm
    - Easy to prove there!
- $(\Phi \Longrightarrow m_1 \leq \Downarrow R\ m_2) \Longrightarrow m_1 \leq \Downarrow R\ (\mathit{assert}\ \Phi \gg= (\lambda\_.\ m_2))$
  - During refinement, we can assume Φ
- Assertions transport knowledge down the refinement chain

Congruences

- Consider **if** b **then** m_1 **else** m_2

Congruences

- Consider **if** b **then** m_1 **else** m_2
  - We can assume b $/\neg b$ for refinement of m_1/m_2

Congruences

- Consider **if** b **then** m_1 **else** m_2
  - We can assume b /$\neg b$ for refinement of m_1/m_2
  - Similar for $\textbf{while}_T$, $\textbf{foreach}_T$, **case**, *bind*, **assert**, ...

Congruences

- Consider **if** b **then** m_1 **else** m_2
  - We can assume b /$\neg b$ for refinement of m_1/m_2
  - Similar for $\textbf{while}_T$, $\textbf{foreach}_T$, **case**, *bind*, **assert**, ...
- Requires strengthened refinement rules for these combinators

Congruences

- Consider $\textbf{if } b \textbf{ then } m_1 \textbf{ else } m_2$
  - We can assume b / $\neg b$ for refinement of m_1/m_2
  - Similar for $\textbf{while}_T$, $\textbf{foreach}_T$, $\textbf{case}$, bind, $\textbf{assert}$, ...
- Requires strengthened refinement rules for these combinators

$$\llbracket b = b';\ \llbracket b;\ b' \rrbracket \Longrightarrow S1 \leq \Downarrow R\ S1';\ \llbracket \neg\ b;\ \neg\ b' \rrbracket \Longrightarrow S2 \leq \Downarrow R\ S2' \rrbracket$$
$$\Longrightarrow (\textbf{if } b \textbf{ then } S1 \textbf{ else } S2) \leq \Downarrow R\ (\textbf{if } b' \textbf{ then } S1' \textbf{ else } S2')$$

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x{=}a*b$ *in return* $(x{,}x+1)$

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x=a*b$ *in return* $(x,x+1)$
  - Application of rules will get stuck

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x=a*b$ *in return* $(x,x+1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x=a*b$ *in return* $(x,x+1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x{=}a*b$ *in return* $(x{,}x+1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG

Slightly Different Program Structure

- Consider **return** $(a{*}b,\ a{*}b{+}1)$ and *let* $x{=}a{*}b$ *in return* $(x{,}x{+}1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG
  - eg
    $f\ x \le \Downarrow R\ M' \Longrightarrow Let\ x\ f \le \Downarrow R\ M'$

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x{=}a*b$ *in return* $(x{,}x{+}1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG
  - eg
    $f\ x \le \Downarrow R\ M' \Longrightarrow Let\ x\ f \le \Downarrow R\ M'$
- Convert any refinement goal to first-order formula and solve

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and *let* $x{=}a*b$ *in return* $(x{,}x+1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG
  - eg
    $f\ x \leq \Downarrow R\ M' \Longrightarrow Let\ x\ f \leq \Downarrow R\ M'$
- Convert any refinement goal to first-order formula and solve
  - Needs to be invoked manually

Slightly Different Program Structure

- Consider **return** $(a*b,\ a*b+1)$ and $\mathit{let}\ x{=}a*b\ \mathit{in}\ \mathit{return}\ (x,x+1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG
  - eg
    $f\,x \le \Downarrow R\ M' \Longrightarrow \mathit{Let}\ x\ f \le \Downarrow R\ M'$
- Convert any refinement goal to first-order formula and solve
  - Needs to be invoked manually
  - Works well for non-recursive programs

Slightly Different Program Structure

- Consider **return** $(a{*}b,\ a{*}b{+}1)$ and *let* $x{=}a{*}b$ *in return* $(x{,}x{+}1)$
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful *subst* and *rewrite*
- Set of recovery rules built in VCG
  - eg
    $f\ x \leq \Downarrow R\ M' \Longrightarrow Let\ x\ f \leq \Downarrow R\ M'$
- Convert any refinement goal to first-order formula and solve
  - Needs to be invoked manually
  - Works well for non-recursive programs
  - May explode

Excursion: Natural Relators

- Given $R_1 :: ('c_1 \times 'a_1)\ set$ and $R_2 :: ('c_2 \times 'a_2)\ set$

Excursion: Natural Relators

- Given $R_1 :: ('c_1 \times 'a_1)\ set$ and $R_2 :: ('c_2 \times 'a_2)\ set$
- How are pairs $'c_1 \times 'c_2$ related to pairs $'a_1 \times 'a_2$?
  - Product relation:
    $R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2).\ (c_1, a_1) \in R_1 \wedge (c_2, a_2) \in R_2\}$

Excursion: Natural Relators

- Given $R_1 :: ('c_1 \times 'a_1)\ set$ and $R_2 :: ('c_2 \times 'a_2)\ set$
- How are pairs $'c_1 \times 'c_2$ related to pairs $'a_1 \times 'a_2$?
  - Product relation:
    $R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2).\ (c_1, a_1) \in R_1 \wedge (c_2, a_2) \in R_2\}$
- Many types have such a natural relator

Excursion: Natural Relators

- Given R_1::($'c_1 \times {}'a_1$) *set* and R_2::($'c_2 \times {}'a_2$) *set*
- How are pairs $'c_1 \times {}'c_2$ related to pairs $'a_1 \times {}'a_2$?
  - Product relation:
    $R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2).\ (c_1, a_1) \in R_1 \wedge (c_2, a_2) \in R_2\}$
- Many types have such a natural relator
  - Algebraic datatypes: Same structure, elements related
    - e.g. $\langle R \rangle$*list_rel* for $'a$ *list* , $\langle R_1, R_2 \rangle$*sum_rel* for $'a + {}'b$

Excursion: Natural Relators

- Given R_1::($'c_1 \times {}'a_1$) *set* and R_2::($'c_2 \times {}'a_2$) *set*
- How are pairs $'c_1 \times {}'c_2$ related to pairs $'a_1 \times {}'a_2$?
  - Product relation:
    $R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2).\ (c_1, a_1) \in R_1 \wedge (c_2, a_2) \in R_2\}$
- Many types have such a natural relator
  - Algebraic datatypes: Same structure, elements related
    - e.g. $\langle R \rangle list\_rel$ for $'a\ list$, $\langle R_1, R_2 \rangle sum\_rel$ for $'a + {}'b$
  - Functions: Argument related $\Longrightarrow$ result related
    - $((f_1, f_2) \in R \rightarrow S) = (\forall (x_1, x_2) \in R.\ (f_1\ x_1, f_2\ x_2) \in S)$

Excursion: Natural Relators

- Given R_1::$('c_1 \times 'a_1)$ *set* and R_2::$('c_2 \times 'a_2)$ *set*
- How are pairs $'c_1 \times 'c_2$ related to pairs $'a_1 \times 'a_2$?
  - Product relation:
    $R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2).\ (c_1, a_1) \in R_1 \wedge (c_2, a_2) \in R_2\}$
- Many types have such a natural relator
  - Algebraic datatypes: Same structure, elements related
    - e.g. $\langle R\rangle \mathit{list\_rel}$ for $'a\ \mathit{list}$, $\langle R_1, R_2\rangle \mathit{sum\_rel}$ for $'a + 'b$
  - Functions: Argument related $\Longrightarrow$ result related
    - $((f_1, f_2) \in R \rightarrow S) = (\forall (x_1, x_2)\in R.\ (f_1\ x_1, f_2\ x_2) \in S)$
  - Nondeterministic results $'a\ \mathit{nres}$
    - $((m_1, m_2) \in \langle R\rangle \mathit{nres\_rel}) = (m_1 \leq \Downarrow R\ m_2)$

Parametricity Examples

- First element of pair: $fst :: 'a \times 'b \Rightarrow 'a$

Parametricity Examples

- First element of pair: $fst :: {}'a \times {}'b \Rightarrow {}'a$
  $(fst, fst) \in A \times_r B \rightarrow A$

Parametricity Examples

- First element of pair: $\mathit{fst} :: {'a} \times {'b} \Rightarrow {'a}$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \to A$
- Append two lists: $(@) :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$

Parametricity Examples

- First element of pair: $\mathit{fst}{::}'a \times {}'b \Rightarrow {}'a$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \rightarrow A$
- Append two lists: $(@){::}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$
  $((@), (@)) \in \langle A\rangle \mathit{list\_rel} \rightarrow \langle A\rangle \mathit{list\_rel} \rightarrow \langle A\rangle \mathit{list\_rel}$

Parametricity Examples

- First element of pair: $\mathit{fst} :: {}'a \times {}'b \Rightarrow {}'a$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \rightarrow A$
- Append two lists: $(@) :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$
  $((@), (@)) \in \langle A \rangle \mathit{list\_rel} \rightarrow \langle A \rangle \mathit{list\_rel} \rightarrow \langle A \rangle \mathit{list\_rel}$
- Limitations in HOL

Parametricity Examples

- First element of pair: $\mathit{fst}::'a \times 'b \Rightarrow 'a$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \to A$
- Append two lists: $(@)::'a\ \mathit{list} \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$
  $((@), (@)) \in \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel}$
- Limitations in HOL
  - Partiality/underdefinedness: $\mathit{hd}::'a\ \mathit{list} \Rightarrow 'a$. $(\mathit{hd},\mathit{hd}) \in ?$

Parametricity Examples

- First element of pair: $\mathit{fst}::'a \times 'b \Rightarrow 'a$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \to A$
- Append two lists: $(@)::'a\ \mathit{list} \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$
  $((@), (@)) \in \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel}$
- Limitations in HOL
  - Partiality/underdefinedness: $\mathit{hd}::'a\ \mathit{list} \Rightarrow 'a$. $(\mathit{hd},\mathit{hd}) \in ?$
  - equality/type-classes: $\mathit{List.member}::'a\ \mathit{list} \Rightarrow 'a \Rightarrow \mathit{bool}$. $(\mathit{List.member},\mathit{List.member}) \in ?$

Parametricity Examples

- First element of pair: $\mathit{fst}{::}'a \times {}'b \Rightarrow {}'a$
  $(\mathit{fst}, \mathit{fst}) \in A \times_r B \rightarrow A$
- Append two lists: $(@){::}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$
  $((@), (@)) \in \langle A \rangle \mathit{list\_rel} \rightarrow \langle A \rangle \mathit{list\_rel} \rightarrow \langle A \rangle \mathit{list\_rel}$
- Limitations in HOL
  - Partiality/underdefinedness: $\mathit{hd}{::}'a\ \mathit{list} \Rightarrow {}'a$. $(\mathit{hd},\mathit{hd}) \in ?$
  - equality/type-classes: $\mathit{List.member}{::}'a\ \mathit{list} \Rightarrow {}'a \Rightarrow \mathit{bool}$. $(\mathit{List.member},\mathit{List.member}) \in ?$
- Solution: Preconditions, generalization

Parametricity Examples

- First element of pair: $\mathit{fst}{::}'a \times {}'b \Rightarrow {}'a$
  $(\mathit{fst},\ \mathit{fst}) \in A \times_r B \to A$
- Append two lists: $(@){::}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$
  $((@),\ (@)) \in \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel}$
- Limitations in HOL
  - Partiality/underdefinedness: $\mathit{hd}{::}'a\ \mathit{list} \Rightarrow {}'a$. $(\mathit{hd},\mathit{hd}) \in ?$
  - equality/type-classes: $\mathit{List.member}{::}'a\ \mathit{list} \Rightarrow {}'a \Rightarrow \mathit{bool}$. $(\mathit{List.member},\mathit{List.member}) \in ?$
- Solution: Preconditions, generalization
  - $[\![l \neq [\,];\ (l',\ l) \in \langle A\rangle \mathit{list\_rel}]\!] \Longrightarrow (\mathit{hd}\ l',\ \mathit{hd}\ l) \in A$

Parametricity Examples

- First element of pair: $\mathit{fst}{::}'a \times 'b \Rightarrow 'a$
  $(\mathit{fst},\ \mathit{fst}) \in A \times_r B \to A$
- Append two lists: $(@){::}'a\ \mathit{list} \Rightarrow 'a\ \mathit{list} \Rightarrow 'a\ \mathit{list}$
  $((@),\ (@)) \in \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel} \to \langle A\rangle \mathit{list\_rel}$
- Limitations in HOL
  - Partiality/underdefinedness: $\mathit{hd}{::}'a\ \mathit{list} \Rightarrow 'a$. $(\mathit{hd},\mathit{hd}) \in ?$
  - equality/type-classes: $\mathit{List.member}{::}'a\ \mathit{list} \Rightarrow 'a \Rightarrow \mathit{bool}$. $(\mathit{List.member},\mathit{List.member}) \in ?$
- Solution: Preconditions, generalization
  - $[\![l \neq [\,];\ (l',\ l) \in \langle A\rangle \mathit{list\_rel}]\!] \Longrightarrow (\mathit{hd}\ l',\ \mathit{hd}\ l) \in A$
  - $\mathit{glist\_member}{::}('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{bool}$. $(\mathit{glist\_member},\ \mathit{glist\_member}) \in (Ra \to Ra \to \mathit{bool\_rel}) \to Ra \to \langle Ra\rangle \mathit{list\_rel} \to \mathit{bool\_rel}$

Summary

- Apply refinement rules for combinators
  - Use some fallback rules to recover (slight) structural changes
  - Manually align bigger changes
- Guess unknown relations
  - By type, using natural relators for structured types
  - Manually
- Show refinements between operators
  - Insert enough assertions into abstract program to prove preconditions

Edka_Refine_Demo.thy

Edmonds-Karp on Residual Graphs

`Workset_Demo.thy`

Implementing Graph by Successor Function

Getting Executable Code

- Iterate refinement until program is deterministic

Getting Executable Code

- Iterate refinement until program is deterministic
    - Program can be extracted into option-monad or plain function

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures
  - Concrete Program can be synthesized automatically in many cases

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures
  - Concrete Program can be synthesized automatically in many cases
- Use Isabelle Code Generator to generate ML/Scala/OCaml/Haskell

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures
  - Concrete Program can be synthesized automatically in many cases
- Use Isabelle Code Generator to generate ML/Scala/OCaml/Haskell
- Caveat: Only allows for purely functional code

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures
  - Concrete Program can be synthesized automatically in many cases
- Use Isabelle Code Generator to generate ML/Scala/OCaml/Haskell
- Caveat: Only allows for purely functional code
  - But this is slow!

Getting Executable Code

- Iterate refinement until program is deterministic
  - Program can be extracted into option-monad or plain function
  - Isabelle Collection Framework: library of verified data structures
  - Concrete Program can be synthesized automatically in many cases
- Use Isabelle Code Generator to generate ML/Scala/OCaml/Haskell
- Caveat: Only allows for purely functional code
  - But this is slow! We want imperative code.

Imperative/HOL

- Model imperative program by state monad

Imperative/HOL

- Model imperative program by state monad

  datatype $('a,'h)\ M = M\ (run\colon \langle 'h \Rightarrow ('a \times 'h) \rangle)$

  return $x = M\ (\lambda s.\ (x,s))$
  $bind\ m_1\ m_2 = M\ (\lambda s.$ **let** $(x,s') = run\ m_1\ s\ in\ run\ (m_2\ x)\ s')$
  $get = M\ (\lambda s.\ (s,s))$
  $put\ s = M\ (\lambda\_.\ ((),s))$

Imperative/HOL

- Model imperative program by state monad

  $\textbf{datatype}\ ('a,'h)\ M = M\ (run\colon \langle 'h \Rightarrow ('a \times 'h)\rangle)$

  $\textbf{return}\ x = M\ (\lambda s.\ (x,s))$
  $bind\ m_1\ m_2 = M\ (\lambda s.\ \textbf{let}\ (x,s') = run\ m_1\ s\ in\ run\ (m_2\ x)\ s')$
  $get = M\ (\lambda s.\ (s,s))$
  $put\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state

Imperative/HOL

- Model imperative program by state monad

  $\textbf{datatype}\ ('a,'h)\ M = M\ (run: \langle 'h \Rightarrow ('a \times 'h) \rangle)$

  $\textbf{return}\ x = M\ (\lambda s.\ (x,s))$
  $bind\ m_1\ m_2 = M\ (\lambda s.\ \textbf{let}\ (x,s') = run\ m_1\ s\ in\ run\ (m_2\ x)\ s')$
  $get = M\ (\lambda s.\ (s,s))$
  $put\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state

- State can be used to model a heap with pointers

Imperative/HOL

- Model imperative program by state monad

  datatype $('a,'h)\ M = M\ (\mathit{run}\colon \langle 'h \Rightarrow ('a \times 'h)\rangle)$

  return $x = M\ (\lambda s.\ (x,s))$
  $\mathit{bind}\ m_1\ m_2 = M\ (\lambda s.$ **let** $(x,s') = \mathit{run}\ m_1\ s\ \mathit{in}\ \mathit{run}\ (m_2\ x)\ s')$
  $\mathit{get} = M\ (\lambda s.\ (s,s))$
  $\mathit{put}\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state

- State can be used to model a heap with pointers
  - Requires some trickery in HOL, but works!

Imperative/HOL

- Model imperative program by state monad

  $\textbf{datatype}\ ('a,'h)\ M = M\ (run\colon \langle 'h \Rightarrow ('a \times 'h)\rangle)$

  $\textbf{return}\ x = M\ (\lambda s.\ (x,s))$
  $bind\ m_1\ m_2 = M\ (\lambda s.\ \textbf{let}\ (x,s') = run\ m_1\ s\ in\ run\ (m_2\ x)\ s')$
  $get = M\ (\lambda s.\ (s,s))$
  $put\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state
- State can be used to model a heap with pointers
  - Requires some trickery in HOL, but works!
- Code generator generates

Imperative/HOL

- Model imperative program by state monad

  $\textbf{datatype}\ ('a,'h)\ M = M\ (\mathit{run}\colon \langle 'h \Rightarrow ('a \times 'h)\rangle)$

  $\textbf{return}\ x = M\ (\lambda s.\ (x,s))$
  $\mathit{bind}\ m_1\ m_2 = M\ (\lambda s.\ \textbf{let}\ (x,s') = \mathit{run}\ m_1\ s\ \mathit{in}\ \mathit{run}\ (m_2\ x)\ s')$
  $\mathit{get} = M\ (\lambda s.\ (s,s))$
  $\mathit{put}\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state
- State can be used to model a heap with pointers
  - Requires some trickery in HOL, but works!
- Code generator generates
  - SML, OCaml, Scala — has explicit imperative constructs

Imperative/HOL

- Model imperative program by state monad

  $\textbf{datatype}\ ('a,'h)\ M = M\ (\mathit{run}\!:\ \langle 'h \Rightarrow ('a \times 'h)\rangle)$

  $\textbf{return}\ x = M\ (\lambda s.\ (x,s))$
  $\mathit{bind}\ m_1\ m_2 = M\ (\lambda s.\ \textbf{let}\ (x,s') = \mathit{run}\ m_1\ s\ \mathit{in}\ \mathit{run}\ (m_2\ x)\ s')$
  $\mathit{get} = M\ (\lambda s.\ (s,s))$
  $\mathit{put}\ s = M\ (\lambda\_.\ ((),s))$

  - Program takes state and returns result and new state
- State can be used to model a heap with pointers
  - Requires some trickery in HOL, but works!
- Code generator generates
  - SML, OCaml, Scala — has explicit imperative constructs
  - Haskell — has efficient heap monad

- Imperative/HOL comes with Hoare-Logic, VCG, etc.

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly
  - But we want to refine abstract programs

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
    - Nice to prove (pointer) programs directly
    - But we want to refine abstract programs
- Ideally, we want:

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly
  - But we want to refine abstract programs
- Ideally, we want:
  1. Specify (imperative) data structures for abstract types

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly
  - But we want to refine abstract programs
- Ideally, we want:
  1. Specify (imperative) data structures for abstract types
  2. Synthesize Imperative/HOL program and refinement proof automatically

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly
  - But we want to refine abstract programs
- Ideally, we want:
  1. Specify (imperative) data structures for abstract types
  2. Synthesize Imperative/HOL program and refinement proof automatically
- The Sepref Tool does exactly that!

Refinement

- Imperative/HOL comes with Hoare-Logic, VCG, etc.
  - Nice to prove (pointer) programs directly
  - But we want to refine abstract programs
- Ideally, we want:
  1. Specify (imperative) data structures for abstract types
  2. Synthesize Imperative/HOL program and refinement proof automatically
- The Sepref Tool does exactly that!
  - And has large collection of readily available data structures

Imperative Refinement Basics

- Establish relation between imperative-program c and nres-program a

  $\mathit{hn\_refine}\ \Gamma\ c\ \Gamma'\ R\ a$

Γ	Heap content before execution
Γ'	Heap content after execution
R	Relation for result

Imperative Refinement Basics

- Establish relation between imperative-program c and nres-program a

  $\mathit{hn\_refine}\ \Gamma\ c\ \Gamma'\ R\ a$

Γ	Heap content before execution
Γ'	Heap content after execution
R	Relation for result

- Formally

  $\mathit{hn\_refine}\ \Gamma\ c\ \Gamma'\ R\ m \equiv$
  $\mathit{nofail}\ m \longrightarrow <\Gamma>\ c\ <\lambda r.\ \Gamma' * (\exists_A x.\ R\ x\ r * \uparrow(\mathbf{return}\ x \leq m))>_t$

  - where $<P>\ c\ <\lambda r.\ Q\ r>_t$ is Hoare-triple for Imperative/HOL programs

Separation Logic

- Refinement relations now also cover heap content.

Separation Logic

- Refinement relations now also cover heap content.
- Examples:

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn} :: \mathit{int}\ \mathit{list} \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn} :: \mathit{int}\ \mathit{list} \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers
  - $\mathit{amtx\_assn}\ M\ N\ \mathit{int\_assn} :: (\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}) \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Function $\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}$ by $M \times N$ array

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn} :: \mathit{int}\ \mathit{list} \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers
  - $\mathit{amtx\_assn}\ M\ N\ \mathit{int\_assn} :: (\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}) \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Function $\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}$ by $M{\times}N$ array
  - $\mathit{int\_assn} = (\lambda i\ i'.\ \uparrow(i' = i))$ — Pure assertion, no heap content.

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn} :: \mathit{int\ list} \Rightarrow \mathit{int\ Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers
  - $\mathit{amtx\_assn}\ M\ N\ \mathit{int\_assn} :: (\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}) \Rightarrow \mathit{int\ Heap.array} \Rightarrow \mathit{assn}$ — Function $\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}$ by $M \times N$ array
  - $\mathit{int\_assn} = (\lambda i\ i'.\ \uparrow (i' = i))$ — Pure assertion, no heap content.
- Assertions separated by $*$: They do not alias!

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn}::\mathit{int}\ \mathit{list} \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers
  - $\mathit{amtx\_assn}\ M\ N\ \mathit{int\_assn}::(\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}) \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Function $\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}$ by $M \times N$ array
  - $\mathit{int\_assn} = (\lambda i\ i'.\ \uparrow (i' = i))$ — Pure assertion, no heap content.
- Assertions separated by $*$: They do not alias!
  - $\mathit{array\_assn}\ \mathit{int\_assn}\ l_1\ a_1 * \mathit{array\_assn}\ \mathit{int\_assn}\ l_2\ a_2$

Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - $\mathit{array\_assn}\ \mathit{int\_assn} :: \mathit{int}\ \mathit{list} \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Implements list of integers by array of integers
  - $\mathit{amtx\_assn}\ M\ N\ \mathit{int\_assn} :: (\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}) \Rightarrow \mathit{int}\ \mathit{Heap.array} \Rightarrow \mathit{assn}$ — Function $\mathit{nat} \times \mathit{nat} \Rightarrow \mathit{int}$ by $M \times N$ array
  - $\mathit{int\_assn} = (\lambda i\ i'.\ \uparrow (i' = i))$ — Pure assertion, no heap content.
- Assertions separated by $*$: They do not alias!
  - $\mathit{array\_assn}\ \mathit{int\_assn}\ l_1\ a_1 * \mathit{array\_assn}\ \mathit{int\_assn}\ l_2\ a_2$
  - Arrays a_1 and a_2 do not overlap

Frame Rule

$$<P>\ c\ <Q> \Longrightarrow\ <P * F>\ c\ <\lambda r.\ Q\ r * F>_t$$

Frame Rule

$<P>\ c\ <Q> \Longrightarrow\ <P * F>\ c\ <\lambda r.\ Q\ r * F>_t$

- Program behaviour does not change if other stuff added to the heap

Frame Rule

$<P>\ c\ <Q> \Longrightarrow\ <P * F>\ c\ <\lambda r.\ Q\ r * F>_t$

- Program behaviour does not change if other stuff added to the heap
  Rule for lookup in hashtable:
  $<\mathit{is\_hashmap}\ m\ ht>\ \mathit{hm\_lookup}\ k\ ht\ <\lambda r.\ \mathit{is\_hashmap}\ m\ ht * \uparrow(r = m\ k)>$

Frame Rule

$<P>\ c\ <Q> \Longrightarrow\ <P * F>\ c\ <\lambda r.\ Q\ r * F>_t$

- Program behaviour does not change if other stuff added to the heap

  Rule for lookup in hashtable:

  $<is\_hashmap\ m\ ht>\ hm\_lookup\ k\ ht\ <\lambda r.\ is\_hashmap\ m\ ht * \uparrow (r = m\ k)>$

  Of course, this still holds if also an array-list is on the heap

  $<is\_hashmap\ m_1\ ht_1 *$
  $\quad is\_array\_list\ l_2$
  $\quad\ al_2>\ hm\_lookup\ k$
  $\qquad\qquad ht_1\ <\lambda r.\ is\_hashmap\ m_1\ ht_1 * is\_array\_list\ l_2\ al_2 * \uparrow (r = m_1\ k)>_t$

Operation Refinement

$<$*is_hashmap m ht*$>$ *hm_lookup k ht* $<\lambda$*r. is_hashmap m ht* $* \uparrow (r = m\ k)>$

Operation Refinement

$<\mathit{is\_hashmap}\ m\ ht>\ \mathit{hm\_lookup}\ k\ ht\ <\lambda r.\ \mathit{is\_hashmap}\ m\ ht * \uparrow (r = m\ k)>$

The hashtable still exists after operation

Operation Refinement

$<\mathit{is\_hashmap}\ m\ \mathit{ht}>\ \mathit{hm\_lookup}\ k\ \mathit{ht}\ <\lambda r.\ \mathit{is\_hashmap}\ m\ \mathit{ht} * \uparrow (r = m\ k)>$

The hashtable still exists after operation

$<\mathit{is\_hashmap}\ m\ \mathit{ht}>\ \mathit{hm\_update}\ k\ v\ \mathit{ht}\ <\lambda r.\ \mathit{is\_hashmap}\ (m(k \mapsto v))\ r>_t$

Operation Refinement

$<\textit{is\_hashmap}\ m\ ht>\ \textit{hm\_lookup}\ k\ ht\ <\lambda r.\ \textit{is\_hashmap}\ m\ ht * \uparrow (r = m\ k)>$

The hashtable still exists after operation

$<\textit{is\_hashmap}\ m\ ht>\ \textit{hm\_update}\ k\ v\ ht\ <\lambda r.\ \textit{is\_hashmap}\ (m(k \mapsto v))\ r>_t$

Original hashtable is gone (destructive update)

Operation Refinement

$<\mathit{is\_hashmap}\ m\ ht>\ \mathit{hm\_lookup}\ k\ ht\ <\lambda r.\ \mathit{is\_hashmap}\ m\ ht * \uparrow (r = m\ k)>$

The hashtable still exists after operation

$<\mathit{is\_hashmap}\ m\ ht>\ \mathit{hm\_update}\ k\ v\ ht\ <\lambda r.\ \mathit{is\_hashmap}\ (m(k \mapsto v))\ r>_t$

Original hashtable is gone (destructive update)

Shortcut notations

$$
\begin{array}{lll}
(\mathit{hm\_lookup}, \lambda k\ m.\ m\ k) & \in \mathit{id}^k * \mathit{is\_hashmap}^k & \to \mathit{id} \\
(\mathit{hm\_update}, \lambda k\ v\ m.\ m(k{\mapsto}v)) & \in \mathit{id}^k * \mathit{id}^k * \mathit{is\_hashmap}^d & \to \mathit{is\_hashmap}
\end{array}
$$

Operation Refinement

$<\textit{is\_hashmap m ht}> \textit{hm\_lookup k ht} <\lambda r.\ \textit{is\_hashmap m ht} * \uparrow (r = m\ k)>$

The hashtable still exists after operation

$<\textit{is\_hashmap m ht}> \textit{hm\_update k v ht} <\lambda r.\ \textit{is\_hashmap}\ (m(k \mapsto v))\ r>_t$

Original hashtable is gone (destructive update)

Shortcut notations

$$\begin{array}{lll} (\textit{hm\_lookup}, \lambda k\ m.\ m\ k) & \in \textit{id}^k * \textit{is\_hashmap}^k & \to \textit{id} \\ (\textit{hm\_update}, \lambda k\ v\ m.\ m(k{\mapsto}v)) & \in \textit{id}^k * \textit{id}^k * \textit{is\_hashmap}^d & \to \textit{is\_hashmap} \end{array}$$

k — keep

d — destroy

Synthesis of Program

Identify operations

do $\{$
  assert $(Q \neq \{\})$;
  $v \leftarrow$ **spec** $x.\ x \in Q \wedge (\forall y \in Q.\ x \leq y)$;
  case $m\ v$ **of**
    $None \Rightarrow$ **return** $(m(v \mapsto True))$
  $|\ Some\ \_ \Rightarrow$ **return** m
$\}$

Synthesis of Program

Identify operations

```
do {
  assert (Q≠{});
  v ← pq_get_min Q;
  case map_lookup v m of
    None ⇒ return (map_update v True m)
  | Some _ ⇒ return m
}
```

Synthesis of Program

Monadify (flatten expressions)

$$
\begin{array}{l}
\textbf{do}\ \{\\
\quad \textbf{assert}\ (Q \neq \{\});\\
\quad v \leftarrow \mathit{pq\_get\_min}\ Q;\\
\quad \textbf{case}\ \mathit{map\_lookup}\ v\ m\ \textbf{of}\\
\qquad \mathit{None} \Rightarrow \textbf{return}\ (\mathit{map\_update}\ v\ \mathit{True}\ m)\\
\quad \mid \mathit{Some}\ \_ \Rightarrow \textbf{return}\ m\\
\}
\end{array}
$$

Synthesis of Program

Monadify (flatten expressions)

```
do {
  assert (Q≠{});
  v ← pq_get_min Q;
  t₁ ← return (map_lookup v m)
  case t₁ of
    None ⇒ do {t₂ ← return True; return (map_update v t₂ m)}
  | Some _ ⇒ return m
}
```

Synthesis of Program

Initialize heap content

```
do {
  assert (Q≠{});
  v ← pq_get_min Q;
  t₁ ← return (map_lookup v m)
  case t₁ of
    None ⇒ do {t₂ ← return True; return (map_update v t₂ m)}
  | Some _ ⇒ return m
}
```

Synthesis of Program

Initialize heap content

$$
\begin{array}{l}
\textbf{do}\ \{ \\
\quad <\ \mathit{minheap}\ \mathit{Qi}\ Q * \mathit{hashmap}\ \mathit{mi}\ m\ > \\
\quad \textbf{assert}\ (Q \neq \{\}); \\
\quad v \leftarrow \mathit{pq\_get\_min}\ Q; \\
\quad t_1 \leftarrow \textbf{return}\ (\mathit{map\_lookup}\ v\ m) \\
\quad \textbf{case}\ t_1\ \textbf{of} \\
\qquad \mathit{None} \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ \mathit{True}; \textbf{return}\ (\mathit{map\_update}\ v\ t_2\ m)\} \\
\quad \mid \mathit{Some}\ \_ \Rightarrow \textbf{return}\ m \\
\}
\end{array}
$$

Synthesis of Program

Symbolic forward execution

$$
\begin{array}{l}
\textbf{do } \{ \\
\quad < \mathit{minheap\ Qi\ Q} * \mathit{hashmap\ mi\ m} > \\
\quad \textbf{assert } (Q \neq \{\}); \\
\quad v \leftarrow \mathit{pq\_get\_min}\ Q; \\
\quad t_1 \leftarrow \textbf{return } (\mathit{map\_lookup}\ v\ m) \\
\quad \textbf{case } t_1 \textbf{ of} \\
\qquad \mathit{None} \Rightarrow \textbf{do } \{t_2 \leftarrow \textbf{return } \mathit{True}; \textbf{return } (\mathit{map\_update}\ v\ t_2\ m)\} \\
\quad \mid \mathit{Some}\ \_ \Rightarrow \textbf{return } m \\
\}
\end{array}
$$

Synthesis of Program

Assert becomes no-op

$\textbf{do}\ \{$
$\quad (*\ \textbf{assert}\ (Q{\neq}\{\});\ *)$
$\quad <\ \mathit{minheap}\ \mathit{Qi}\ Q * \mathit{hashmap}\ \mathit{mi}\ m\ >\ |\ Q{\neq}[]$
$\quad v \leftarrow \mathit{pq\_get\_min}\ Q;$
$\quad t_1 \leftarrow \textbf{return}\ (\mathit{map\_lookup}\ v\ m)$
$\quad \textbf{case}\ t_1\ \textbf{of}$
$\qquad \mathit{None} \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ \mathit{True};\ \textbf{return}\ (\mathit{map\_update}\ v\ t_2\ m)\}$
$\quad |\ \mathit{Some}\ \_ \Rightarrow \textbf{return}\ m$
$\}$

Synthesis of Program

$(\mathit{minheap\_get\_min}, \mathit{pq\_get\_min}) \in \mathit{minheap}^k \rightarrow \mathit{int}$

$\textbf{do}\ \{$
$\quad (*\ \textbf{assert}\ (Q{\neq}\{\});\ *)$
$\quad <\ \mathit{minheap}\ \mathit{Qi}\ Q * \mathit{hashmap}\ \mathit{mi}\ m\ >\ |\ Q{\neq}[]$
$\quad v \leftarrow \mathit{pq\_get\_min}\ Q;$
$\quad t_1 \leftarrow \textbf{return}\ (\mathit{map\_lookup}\ v\ m)$
$\quad \textbf{case}\ t_1\ \textbf{of}$
$\quad\quad \mathit{None} \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ \mathit{True};\ \textbf{return}\ (\mathit{map\_update}\ v\ t_2\ m)\}$
$\quad |\ \mathit{Some}\ \_ \Rightarrow \textbf{return}\ m$
$\}$

Synthesis of Program

Result now also bound.

$\textbf{do}\ \{$
$\quad (*\ \textbf{assert}\ (Q{\neq}\{\});\ *)$
$\quad \mathit{vi} \leftarrow \mathit{minheap\_get\_min}\ \mathit{Qi};$
$\quad <\ \mathit{minheap}\ Q\ \mathit{Qi} * \mathit{hashmap}\ m\ \mathit{mi} * \mathit{int}\ v\ \mathit{vi}\ >\ |\ Q{\neq}[]$
$\quad t_1 \leftarrow \textbf{return}\ (\mathit{map\_lookup}\ v\ m)$
$\quad \textbf{case}\ t_1\ \textbf{of}$
$\qquad \mathit{None} \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ \mathit{True};\ \textbf{return}\ (\mathit{map\_update}\ v\ t_2\ m)\}$
$\quad |\ \mathit{Some}\ \_ \Rightarrow \textbf{return}\ m$
$\}$

Synthesis of Program

$(\textit{hm\_lookup}, \textit{map\_lookup}) \in \textit{int}^k * \textit{hashmap}^k \rightarrow (\textit{bool})\textit{option}$

$\textbf{do}\ \{$
$\quad (*\ \textbf{assert}\ (Q \neq \{\});\ *)$
$\quad vi \leftarrow \textit{minheap\_get\_min}\ Qi;$
$\quad <\ \textit{minheap}\ Q\ Qi * \textit{hashmap}\ m\ mi * \textit{int}\ v\ vi >\ |\ Q \neq []$
$\quad t_1 \leftarrow \textbf{return}\ (\textit{map\_lookup}\ v\ m)$
$\quad \textbf{case}\ t_1\ \textbf{of}$
$\qquad \textit{None} \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ \textit{True};\ \textbf{return}\ (\textit{map\_update}\ v\ t_2\ m)\}$
$\quad |\ \textit{Some}\ \_ \Rightarrow \textbf{return}\ m$
$\}$

Synthesis of Program

$$
\begin{array}{l}
\textbf{do}\ \{\\
\quad (*\ \textbf{assert}\ (Q \neq \{\});\ *)\\
\quad vi \leftarrow minheap\_get\_min\ Qi;\\
\quad ti_1 \leftarrow hm\_lookup\ vi\ mi;\\
\quad <\ minheap\ Q\ Qi * hashmap\ m\ mi * int\ v\ vi * (bool)option\ t_1\ ti_1\ >\ |\ Q \neq []\\
\quad \textbf{case}\ t_1\ \textbf{of}\\
\qquad None \Rightarrow \textbf{do}\ \{t_2 \leftarrow \textbf{return}\ True;\ \textbf{return}\ (map\_update\ v\ t_2\ m)\}\\
\quad |\ Some\ \_ \Rightarrow \textbf{return}\ m\\
\}
\end{array}
$$

Synthesis of Program

Split: Translate both branches separately

do $\{$
  (* **assert** ($Q \neq \{\}$); *)
  $vi \leftarrow minheap\_get\_min\ Qi$;
  $ti_1 \leftarrow hm\_lookup\ vi\ mi$;
  $<minheap\ Q\ Qi * hashmap\ m\ mi * int\ v\ vi * (bool) option\ t_1\ ti_1> \mid Q \neq []$
  case t_1 **of**
    $None \Rightarrow$ **do** $\{t_2 \leftarrow$ **return** $True$; **return** $(map\_update\ v\ t_2\ m)\}$
  $\mid Some\ \_ \Rightarrow$ **return** m
$\}$

Synthesis of Program

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      < minheap Q Qi * hashmap m mi * int v vi > | Q≠[], t₁=None
      t₂ ← return True;
      return (map_update v t₂ m)
      }
  | Some _ ⇒
      < minheap Q Qi * hashmap m mi * int v vi * bool t₃ ti₃ > | Q≠[],
t₁=Some t₃
      return m
}
```

Synthesis of Program

$(\textbf{return } \mathit{True}, \textbf{return } \mathit{True}) \in - \rightarrow \mathit{bool}$

do $\{$
  $(*$ **assert** $(Q \neq \{\}); *)$
  $\mathit{vi} \leftarrow \mathit{minheap\_get\_min}\ \mathit{Qi};$
  $\mathit{ti}_1 \leftarrow \mathit{hm\_lookup}\ \mathit{vi}\ \mathit{mi};$
  case ti_1 **of**
    $\mathit{None} \Rightarrow$ **do** $\{$
      $< \mathit{minheap}\ Q\ \mathit{Qi} * \mathit{hashmap}\ m\ \mathit{mi} * \mathit{int}\ v\ \mathit{vi} > \mid Q \neq [],\ t_1 = \mathit{None}$
      $t_2 \leftarrow$ **return** $\mathit{True};$
      return $(\mathit{map\_update}\ v\ t_2\ m)$
      $\}$
  $\mid \mathit{Some}\ \_ \Rightarrow$
      $< \mathit{minheap}\ Q\ \mathit{Qi} * \mathit{hashmap}\ m\ \mathit{mi} * \mathit{int}\ v\ \mathit{vi} * \mathit{bool}\ t_3\ \mathit{ti}_3 > \mid Q \neq [],$
$t_1 = \mathit{Some}\ t_3$
      return m
$\}$

Synthesis of Program

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      ti₂ ← return True;
     < minheap Q Qi ∗ hashmap m mi ∗ int v vi ∗ bool ti₂ t₂ > | Q≠[], t₁=None
      return (map_update v t₂ m)
      }
  | Some _ ⇒
      < minheap Q Qi ∗ hashmap m mi ∗ int v vi ∗ bool t₃ ti₃ > | Q≠[],
t₁=Some t₃
      return m
}
```

Synthesis of Program

$(\mathit{hm\_update},\mathit{map\_update}) \in \mathit{int}^k * \mathit{bool}^k * \mathit{hashmap}^d \to \mathit{hashmap}$

do $\{$
$(*$ **assert** $(Q \neq \{\}); *)$
$\mathit{vi} \leftarrow \mathit{minheap\_get\_min}\ \mathit{Qi};$
$\mathit{ti}_1 \leftarrow \mathit{hm\_lookup}\ \mathit{vi}\ \mathit{mi};$
case ti_1 **of**
$\mathit{None} \Rightarrow$ **do** $\{$
$\mathit{ti}_2 \leftarrow$ **return** $\mathit{True};$
$< \mathit{minheap}\ Q\ \mathit{Qi} * \mathit{hashmap}\ m\ \mathit{mi} * \mathit{int}\ v\ \mathit{vi} * \mathit{bool}\ \mathit{ti}_2\ t_2 > \mid Q \neq [], t_1 = \mathit{None}$
return $(\mathit{map\_update}\ v\ t_2\ m)$
$\}$
$\mid \mathit{Some}\ \_ \Rightarrow$
$< \mathit{minheap}\ Q\ \mathit{Qi} * \mathit{hashmap}\ m\ \mathit{mi} * \mathit{int}\ v\ \mathit{vi} * \mathit{bool}\ t_3\ \mathit{ti}_3 > \mid Q \neq [],$
$t_1 = \mathit{Some}\ t_3$
return m
$\}$

Destructive update, m_i no longer valid

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      ti₂ ← return True;
      hm_update vi ti₂ mi
      < minheap Q Qi * int v vi * bool ti₂ t₂ * hashmap @r @ri > | Q≠[],
t₁=None
      }
  | Some _ ⇒
      < minheap Q Qi * hashmap m mi * int v vi * bool t₃ ti₃ > | Q≠[],
t₁=Some t₃
      return m
}
```

Synthesis of Program

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      ti₂ ← return True;
      hm_update vi ti₂ mi
      < minheap Q Qi * int v vi * bool ti₂ t₂ * hashmap @r @ri > | Q≠[],
t₁=None
      }
  | Some _ ⇒
      < minheap Q Qi * hashmap m mi * int v vi * bool t₃ ti₃ > | Q≠[],
t₁=Some t₃
      return m
}
```

Pass on m_i as result. No aliasing, so m_i no longer valid!

$\mathbf{do}\ \{$
  $(*\ \mathbf{assert}\ (Q{\neq}\{\});\ *)$
  $vi \leftarrow minheap\_get\_min\ Qi;$
  $ti_1 \leftarrow hm\_lookup\ vi\ mi;$
  $\mathbf{case}\ ti_1\ \mathbf{of}$
    $None \Rightarrow \mathbf{do}\ \{$
      $ti_2 \leftarrow \mathbf{return}\ True;$
      $hm\_update\ vi\ ti_2\ mi$
      $<minheap\ Q\ Qi * int\ v\ vi * bool\ ti_2\ t_2 * hashmap\ @r\ @ri> \mid Q{\neq}[],$
$t_1{=}None$
      $\}$
  $\mid Some\ \_ \Rightarrow$
      $\mathbf{return}\ mi$
      $<minheap\ Q\ Qi * int\ v\ vi * bool\ t_3\ ti_3 * hashmap\ @r\ @ri> \mid Q{\neq}[],$
$t_1{=}Some\ t_3$
$\}$

Merge.

$\textbf{do}$ $\{$
  $(*$ $\textbf{assert}$ $(Q \neq \{\}); *)$
  $vi \leftarrow minheap\_get\_min\ Qi;$
  $ti_1 \leftarrow hm\_lookup\ vi\ mi;$
  $\textbf{case}\ ti_1\ \textbf{of}$
    $None \Rightarrow \textbf{do}\ \{$
      $ti_2 \leftarrow \textbf{return}\ True;$
      $hm\_update\ vi\ ti_2\ mi$
      $< minheap\ Q\ Qi * int\ v\ vi * bool\ ti_2\ t_2 * hashmap\ @r\ @ri > \mid Q \neq [],$
$t_1 = None$
      $\}$
  $\mid Some\ \_ \Rightarrow$
      $\textbf{return}\ mi$
      $< minheap\ Q\ Qi * int\ v\ vi * bool\ t_3\ ti_3 * hashmap\ @r\ @ri > \mid Q \neq [],$
$t_1 = Some\ t_3$
$\}$

Merge. ti_2 goes out of scope.

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      ti₂ ← return True;
      hm_update vi ti₂ mi
      }
  | Some _ ⇒
      return mi

  < minheap Q Qi ∗ int v vi ∗ hashmap @r @ri ∗ ∗ (bool)option t₁ ti₁ > | Q≠[]
}
```

Synthesis of Program

```
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti₁ ← hm_lookup vi mi;
  case ti₁ of
    None ⇒ do {
      ti₂ ← return True;
      hm_update vi ti₂ mi
      }
  | Some _ ⇒
      return mi

  < minheap Q Qi * int v vi * hashmap @r @ri * * (bool)option t₁ ti₁ > | Q≠[]
}
```

Sepref_Demo.thy

Toy Example with Hashtable and Min-Heap

Workset_Demo_Impl.thy

Implementing the Workset Algorithm

`Edka_Impl_Demo.thy`

Implementation of Edmonds-Karp

Summary

- Select (imperative) data structures
- Synthesize Imperative/HOL program
- Generate ML/OCaml/Haskell/Scala program

Lecture Conclusions

- Prove algorithmic ideas on abstract level
  - No need to bother with implementation details
- Prove/reuse data-structures and sub-algorithms (independently)
  - Lot's of stuff has already been done: reuse, adapt, extend
- Use refinement to relate abstract with more concrete algorithms
  - Multiple steps. Step-size is a trade-off.
- Finally: Use Sepref to go to Imperative/HOL, generate code

Remarks on Learning Curve

- Should learn basic Isabelle first
  - E.g. Concrete Semantics book by Nipkow and Klein
- Then Refinement Framework (Look for tutorials in AFP)
  - Works quite smoothly
- Then Sepref Tool (Tutorial in AFP)
  - Can have quite subtle errors, needs some experience