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

- Maxflow Algorithms (eg Ford-Fulkerson, Edmonds-Karp, Push-Relabel)
- Parametricity (eg Theorems for Free!)
- Separation Logic
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
- Maxflow Algorithms (eg Ford-Fulkerson, Edmonds-Karp, Push-Relabel)
- Parametricity (eg Theorems for Free!)
- Separation Logic
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 (e.g. in Haskell)
- Hoare-Calculus
- Interactive Theorem Prover
  - Isabelle
  - Coq
  - Other
- Maxflow Algorithms (e.g. Ford-Fulkerson, Edmonds-Karp, Push-Relabel)
- Parametricity (e.g. 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
This lecture: Not an Isabelle introduction
• This lecture: Not an Isabelle introduction
• Trying to present ideas independent of Isabelle
• 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*

![Graph](image-url)
Finding Maximum Flow

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

```
• Start with empty flow
• Find augmenting path
• Increase flow
```
Finding Maximum Flow

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

\begin{itemize}
\item \(s\to a\to c\to t\)
\item \(s\to b\to d\to t\)
\end{itemize}
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 \textit{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 \( \# \) augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
Ford-Fulkerson Method

- Theorem: Flow is maximal iff \( \not\exists \) augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

```plaintext
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 \( \not\exists \) augmenting path
  - Corollary of Min-Cut/Max-Flow theorem
- Yields greedy algorithm for maximum flow

\[
\begin{align*}
\text{set flow to zero} \\
\text{while exists augmenting path} \\
\text{augment flow along path}
\end{align*}
\]

- 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 \(\text{capacity}[N][N]\)
  - Adjacency map by array \(\text{node list}[N]\)
Data Structures

• Residual Graph
  • Operations: successors of node, capacity of edge
  • Nodes by natural numbers from \{0..<N\}
  • Adjacency matrix by array (\textit{capacity}[N][N])
  • Adjacency map by array (\textit{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 \texttt{int[N]} and map \texttt{None} to $-1$
  • Current and next set: \texttt{node list}
Data Structures

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

- BFS algorithm
  - Predecessor map: Array ((node option)[N])
    - Or use \textit{int}[N] and map \textit{None} to \(-1\)
  - Current and next set: \textit{node list}

- Augmenting path: \textit{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
  \[\Rightarrow\] 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
• 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

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

  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
  \[\text{return} :: 'a \Rightarrow 'a M\]
  \[\text{bind} :: 'a M \Rightarrow ('a \Rightarrow 'b M) \Rightarrow 'b M\]
- Intuition:
  \[\text{return}\] a value,
  \[\text{bind}\] 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
  \[
  \text{return} :: \ 'a \Rightarrow \ 'a M \\
  \text{bind} :: \ 'a M \Rightarrow (\ 'a \Rightarrow \ 'b M) \Rightarrow \ 'b M
  \]
- Intuition:
  \[
  \text{return} \text{ a value} ,
  \text{bind} \text{ the result of } m_1 \text{ to variable } x , \text{ then execute } m_2 x
  \]
- Syntax sugar
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
  - `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} :: \( 'a \Rightarrow 'a M \)
  - \textbf{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} \{ \( x \leftarrow m_1; \ m_2 \ x \) \} = \text{bind } m_1 \ (\lambda x. \ m_2 \ x)\)
- Monad laws
  - \textbf{do} \{ \( x \leftarrow \text{return } v; \ f x \) \} = f \ v
  - \textbf{do} \{ \( x \leftarrow m; \ \text{return } x \) \} = m
  - \textbf{do} \{ \( y \leftarrow \text{do} \{ x \leftarrow m; \ f x \}; \ g y \) \} = \textbf{do} \{ \( x \leftarrow m; \ y \leftarrow f x; \ g y \) \}
Monad Syntax Sugar

- infix bind notation
Monad Syntax Sugar

- infix bind notation
  - \( m_1 \gg m_2 = \text{bind } m_1 \ m_2 \)
Monad Syntax Sugar

- **infix bind notation**
  
  - $m_1 >>= m_2$ \(= \text{bind } m_1 \ m_2\)
  
  - $m_1 >> m_2$ \(= \text{bind } m_1 (\lambda_. \ m_2)\)
Monad Syntax Sugar

• infix bind notation
  
  • \( m_1 \gg= m_2 \) = \textit{bind} \( m_1 \) \( m_2 \)
  
  • \( m_1 \gg m_2 \) = \textit{bind} \( m_1 \) (\( \lambda x. m_2 \))

• do-notation
Monad Syntax Sugar

- **infix bind notation**
  - \( m_1 >>= m_2 \) = \( \text{bind } m_1 \; m_2 \)
  - \( m_1 >>= m_2 \) = \( \text{bind } m_1 \; (\lambda \_ . \; m_2) \)

- **do-notation**
  - \( \textbf{do} \{ \; m_1 ; \; m_2 \; \} \) = \( \text{bind } m_1 \; (\lambda \_ . \; m_2) \)
Monad Syntax Sugar

- infix bind notation
  - \( m_1 \gg m_2 = bind m_1 m_2 \)
  - \( m_1 \gg m_2 = bind m_1 (\lambda \cdot m_2) \)

- do-notation
  - \( \text{do} \{ m_1; m_2 \} = bind m_1 (\lambda \cdot m_2) \)
  - \( \text{do} \{ x \leftarrow m_1; y \leftarrow m_2 \ x; z \leftarrow m_3 \ x \ y; \ldots \} \)
Monad Syntax Sugar

- infix bind notation
  - $m_1 >>= m_2 = bind m_1 m_2$
  - $m_1 >>= m_2 = bind m_1 (\lambda_. m_2)$

- do-notation
  - $\text{do } \{ m_1; m_2 \} = bind m_1 (\lambda_. m_2)$
  - $\text{do } \{ x\leftarrow m_1; y\leftarrow m_2 \; x; \; z \leftarrow m_3 \; x \; y; \; \ldots \; \}$
  - $\text{do } \{ x\leftarrow m_1; y\leftarrow m_2; m_3; \; z\leftarrow m_4 \; x; \; \ldots \; \}$
Monad Syntax Sugar

- **infix bind notation**
  
  - \( m_1 >>= m_2 \) \( \equiv \) \( \text{bind } m_1 \ m_2 \)
  
  - \( m_1 >>= m_2 \) \( \equiv \) \( \text{bind } m_1 \ (\lambda _. \ m_2) \)

- **do-notation**
  
  - \( \textbf{do} \ \{ \ m_1; \ m_2 \ \} \) \( \equiv \) \( \text{bind } m_1 \ (\lambda _. \ m_2) \)
  
  - \( \textbf{do} \ \{ \ x \leftarrow m_1; \ y \leftarrow m_2 \ x; \ z \leftarrow m_3 \ x \ y; \ldots \ \} \)
  
  - \( \textbf{do} \ \{ \ x \leftarrow m_1; \ y \leftarrow m_2; \ m_3; \ z \leftarrow m_4 \ x; \ldots \ \} \)
  
  - \( \textbf{do} \ \{ \ \ldots; \ (x_1,x_2,x_3) \leftarrow m; \ldots \ \} \)
Monad Syntax Sugar

• infix bind notation
  
  • \( m_1 \gg m_2 \) = \textit{bind} \( m_1 \) \( m_2 \)
  
  • \( m_1 \gg m_2 \) = \textit{bind} \( m_1 \) \((\lambda . \ m_2)\)

• do-notation
  
  • \texttt{do \{ \ m_1; m_2 \ \} = \textit{bind} \ m_1 \ (\lambda . \ m_2)\}
  
  • \texttt{do \{ \ x\leftarrow m_1; y\leftarrow m_2 \ x; z \leftarrow m_3 \ x \ y; \ldots \ \} \}
  
  • \texttt{do \{ \ x\leftarrow m_1; y\leftarrow m_2; m_3; z\leftarrow m_4 \ x; \ldots \ \} \}
  
  • \texttt{do \{ \ldots; (x_1,x_2,x_3)\leftarrow m; \ldots \ \} \}
  
  • \texttt{do \{ \ldots; \textbf{let} (x_1,x_2,x_3) = a; \ldots \ \} \}

• Any HOL function, and its syntax

  \texttt{do \{ \ x\leftarrow m_1; \textbf{if} \ x < 0 \textbf{then return} \ (\ -1 ) \textbf{else if} \ x = 0 \textbf{then return} \ 0 \textbf{else if} \ x > 0 \textbf{then return} \ 1 \ldots \ \} \}
Monad Syntax Sugar

- **infix bind notation**
  - $m_1 >>= m_2 = \text{bind } m_1 \ m_2$
  - $m_1 >>= m_2 = \text{bind } m_1 (\lambda \_ . \ m_2)$

- **do-notation**
  - do \{ \( m_1; m_2 \) \} = bind \( m_1 (\lambda \_ . \ m_2) \)
  - do \{ x\leftarrow m_1; y\leftarrow m_2 \ x; z \leftarrow m_3 \ x \ y; \ldots \ \} 
  - do \{ x\leftarrow m_1; y\leftarrow m_2; m_3; z\leftarrow m_4 \ x; \ldots \ \} 
  - do \{ \ldots ; (x_1,x_2,x_3)\leftarrow m; \ldots \ \}
  - do \{ \ldots ; \text {let} \ (x_1,x_2,x_3) = a; \ldots \ \}

- Any HOL function, and its syntax

```haskell
do { 
    x\leftarrow m;
    if x<0 then return (\neg 1 )
    else if x=0 then return 0
    else if x>0 then return 1
}
```
Monads for Programming

- Monad models sequential execution
  - \texttt{do \{ x\leftarrow m; f x \}} First execute \( m \), then \( f \)
Monads for Programming

- Monad models sequential execution
  - `do { x ← 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←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 = case m_1 of None ⇒ None | Some x ⇒ m_2 x
    
    fail = None
Monads for Programming

- Monad models sequential execution
  - `do { x←m; f x }` First execute \( m \), then \( f \)
- More functionality can be added by structure of type \('a M\)
  - Computations that can fail: \( M=\text{option} \)
    - `return x = Some x`
    - `bind m_1 m_2 = case m_1 of None ⇒ None | Some x ⇒ m_2 x`
    - `fail = None`
  - Example: `do { if x=0 then fail else return (1 / x) }`
- Many more: exceptions, state, output, probability, ...
Monads for Programming

- Monad models sequential execution
  - `do { x←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 = case m_1 of None ⇒ None | Some x ⇒ m_2 x`
    - `fail = None`
  - Example: `do { if x=0 then fail else return (1 / x) }`

- Nondeterministic computations `M=set`
  - `return x = {x}`
  - `bind m_1 m_2 = ∪ { m_2 x | x. x∈m_1 }
  - `choose x. Φ x = { x. Φ x }

- Many more: exceptions, state, output, probability, ...
Monads for Programming

• Monad models sequential execution
  - \texttt{do \{ x←m; f x \}} First execute \( m \), then \( f \)

• More functionality can be added by structure of type \( 'a M \)
  - Computations that can fail: \( M=\text{option} \)
    - \texttt{return } x = \text{Some } x
    - \texttt{bind } m_1 \ m_2 = \texttt{case } m_1 \text{ of None ⇒ None | Some } x \Rightarrow m_2 \ x
    - \texttt{fail } = \text{None}
  
    • Example: \texttt{do \{ if } x=0 \text{ then fail else return } (1 / x) \}

• Nondeterministic computations \( M=\text{set} \)
  - \texttt{return } x = \{ x \}
  - \texttt{bind } m_1 \ m_2 = \bigcup \{ m_2 \ x \mid x. \ x \in m_1 \}
  - \texttt{choose } x. \ \Phi \ x = \{ x. \ \Phi \ x \}

    • Example: \texttt{do \{ e ← 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: \( M=\text{option} \)
    - \texttt{return} \( x = \text{Some} \ x \)
    - \texttt{bind} \( m_1 \ m_2 = \text{case} \ m_1 \text{ of} \ None \Rightarrow \text{None} \mid \text{Some} \ x \Rightarrow \ m_2 \ x \)
    - \texttt{fail} = \text{None}

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

• Nondeterministic computations \( M=\text{set} \)
  - \texttt{return} \( x = \{x\} \)
  - \texttt{bind} \( m_1 \ m_2 = \bigcup \{ m_2 \ x \mid x. \ x \in m_1 \} \)
  - \texttt{choose} \( x. \ \Phi \ x = \{ x. \ \Phi \ x \} \)

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

• Many more: exceptions, state, output, probability, ...
The \textit{nres} Monad

- Combines failure and nondeterminism monad. \( M = \textit{nres} \) where

\begin{verbatim}
datatype 'a nres = FAIL | RES 'a set

return x = RES \{x\}
bind FAIL f = FAIL
bind (RES X) f = Sup \{f x | x \in X\}

where
Sup X = (if FAIL \in X then FAIL else RES (\bigcup \{Y. RES Y \in X\}))
\end{verbatim}
The \textit{nres} Monad

- Combines failure and nondeterminism monad. \( M = \textit{nres} \) where

  \texttt{datatype 'a nres = FAIL | RES 'a set}

  \texttt{return } x = \textit{RES} \{x\}

  \texttt{bind FAIL } f = \textit{FAIL}

  \texttt{bind (RES X) } f = \textit{Sup} \{f \cdot x | x. x \in X\}

  where

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

- Derived combinators

  \texttt{spec } x. \Phi x = \textit{RES} \{x. \Phi x\}

  \texttt{assert } \Phi = \texttt{if } \Phi \texttt{ then return () else FAIL}

  \texttt{select } p. \Phi p = \texttt{if } \exists x. \Phi x \texttt{ then RES } \{\texttt{Some} x | x. \Phi x\} \texttt{ else return None}
• Generalized fold combinator

\[\text{nfoldli} \; \lambda \; \text{c f s} = \text{return s}\]
\[\text{nfoldli} \; (x \neq \text{ls}) \; \text{c f s} =\]
\[(\text{if c s then } f \times s \Rightarrow (\lambda s. \text{nfoldli} \; \text{ls c f s}) \; \text{else return s})\]
Structural Recursion

- Generalized fold combinator
  \[\text{nfoldli}\ ]\ c\ f\ s\ =\ \text{return}\ s\]
  \[\text{nfoldli}\ (x\ \#\ ls)\ c\ f\ s\ =\]
  \[\text{(if}\ c\ s\ \text{then}\ f\ x\ s\ \gg\ =\ (λs. nfoldli\ ls\ c\ f\ s)\ \text{else}\ \text{return}\ s)\]

- Iteration over set
  \[\text{foreach}\ S\ f\ σ\ =\ \text{do}\ {\}
    \text{assert}\ (\text{finite}\ X);\]
  \[l←\text{spec}\ l.\ \text{distinct}\ l\ ∧\ S\ =\ \text{set}\ l;\]
  \[\text{nfoldli}\ l\ (λ_.\ \text{True})\ f\ σ\]
  \}
Structural Recursion

- Generalized fold combinator

\[
\text{nfoldli } [] \ c \ f \ s = \text{return } s \\
\text{nfoldli } (x \neq \text{ls}) \ c \ f \ s = \\
(\text{if } c \ s \text{ then } f \times s \geq (\lambda s. \text{nfoldli ls c f s}) \text{ else return } s)
\]

- Iteration over set

\text{foreach } S \ f \ \sigma = \text{do } \{
\text{assert } (\text{finite } X); \\
\text{l} \leftarrow \text{spec } l. \text{ distinct } l \land S = \text{ set } l; \\
\text{nfoldli l } (\lambda_\_. \text{ True}) \ f \ \sigma
\}

- Warning: Actual implementation of \text{foreach} and friends suffers from legacy problems. But works nicely at the surface!
Arbitrary Recursion

- Recursion via fixed-point construction
  
  \[ \text{trimono } B \iff (\text{rec}_T D. B D) = B (\text{rec}_T D. B D) \]
Arbitrary Recursion

- Recursion via fixed-point construction
  
  \( \text{trimono } B \longmapsto (\text{rec}_T \ D. \ B \ D) = B (\text{rec}_T \ D. \ B \ D) \)

- Yields **FAIL** if there is a nonterminating execution (total correctness)
Arbitrary Recursion

• Recursion via fixed-point construction

\[ \text{trimono } B \rightarrow (\text{rec}_T D. B D) = B (\text{rec}_T D. B D) \]

• Yields \textit{FAIL} if there is a nonterminating execution (total correctness)
  • \text{rec } D. B D ignores nonterminating executions (partial correctness)
Arbitrary Recursion

- Recursion via fixed-point construction

\[ \text{trimono } B \implies (\text{rec}_T \ D. \ B \ D) = B (\text{rec}_T \ D. \ B \ D) \]

- Yields \textit{FAIL} if there is a nonterminating execution (total correctness)
  - \text{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

  \textit{trimono} \ b \ \mapsto \ (\text{rec}_T \ D. \ B \ D) = \ b \ (\text{rec}_T \ D. \ B \ D)

  • Yields \textit{FAIL} if there is a nonterminating execution (total correctness)
    • \textit{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

\[ \text{trimono } B \iff (\text{rec}_T \ D. \ B \ D) = B (\text{rec}_T \ D. \ B \ D) \]

  • Yields \textit{FAIL} if there is a nonterminating execution (total correctness)
    • \text{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

\[ \text{while}_T \ b \ f \equiv \text{rec}_T \ W. (\lambda s. \text{if } b \ s \text{ then } f \ s \gg W \text{ else return } s) \]
Arbitrary Recursion

• Recursion via fixed-point construction

\[ \text{trimono } B \implies (\text{rec}_T D. B D) = B (\text{rec}_T D. B D) \]

• Yields \textit{FAIL} if there is a nonterminating execution (total correctness)

  • \text{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

\[ \text{while}_T b f \equiv \text{rec}_T W. (\lambda s. \text{if } b \ s \ \text{then} \ f \ s \ \text{else return } s) \]

• Program is ordinary term in HOL (shallow embedding)
Examples

Select item from (non-empty) set $S$:
Examples

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

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

$\text{sum\_up } S = \text{do }$
$(S,a) \leftarrow \text{while}_T (\lambda(S,a). \ S \neq \{\}) (\lambda(S,a). \text{do }$
$x \leftarrow \text{spec } x. \ x \in S$
$\text{return } (S \conc \{x\}, a + x)$
$\}) (S, 0);$
$\text{return } a$
$\}$

\textit{Only works for finite sets.}
Examples

Select item from (non-empty) set $S$:

$$\text{spec } x. \ x \in S$$

Iterate until set is empty, sum up elements

$$\text{sum\_up } S = \text{do } \{$$

$$(S,a) \leftarrow \text{while}_T (\lambda(S,a). \ S \neq \{\}) \ (\lambda(S,a). \text{do } \{$$

$$x \leftarrow \text{spec } x. \ x \in S;$$

$$\text{return } (S - \{x\}, a + x)$$

$$\}) \ (S, 0);$$

$$\text{return } a$$

$$\}$$

Only works for finite sets.
Check if $42 \in S$, $S$ finite. Iterate over $S$. 

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

\[ \text{is}_{42} \text{in } S = \text{do} \{ \]
\[ (S,f) \leftarrow \text{while}_T (\lambda(S,f). S\neq\emptyset) \land \neg f) (\lambda(S,a). \text{do} \{ \]
\[ x \leftarrow \text{spec } x. x \in S; \]
\[ \text{return } (S-\{x\}, x=42) \]
\[ \}) (S, \text{False}); \]
\[ \text{return } f \]
\[ \} \]
Examples

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

\[
\text{is}_{42} \text{in } S = \text{ do } \{
(S,f) \leftarrow \text{ while } T (\lambda(S,f) \cdot S \neq \{\} \land \neg f) (\lambda(S,a) \cdot \text{ do } \{
  x \leftarrow \text{spec } x \cdot x \in S;
  \text{ return } (S - \{x\}, x = 42)
\}) (S, \text{False});
\text{ return } f
\}
\]

We have emulated a \textit{break} by Boolean flag $f$. 
Edmonds-Karp Algorithm

\[
\begin{align*}
\text{let } f &= (\lambda_. 0); \\
(f,\_)& \leftarrow \text{while}_T \\
(\lambda(f,\text{brk}). \neg \text{brk}) \\
(\lambda(f,\_). \text{do} \{ \\
\quad p &\leftarrow \text{select \, p. \, Graph.isShortestPath (residualGraph c f) s p t}; \\
\quad \text{case } p \text{ of} \\
\quad \quad \text{None} &\Rightarrow \text{return } (f, \text{True}) \\
\quad \quad | \text{Some } p &\Rightarrow \text{do} \{ \\
\quad \quad \quad \text{let } f &= \text{NFlow.augment_with_path c f p}; \\
\quad \quad \quad \text{return } (f, \text{False}) \\
\quad \quad \}\} \\
(\_&\text{False}); \\
\text{return } f
\end{align*}
\]
• Locale \textit{Network} fixes flow network

\begin{verbatim}
locale Network
  fixes $c :: \text{nat} \times \text{nat} \Rightarrow \text{'capacity}$
  and $s :: \text{nat}$
  and $t :: \text{nat}$
  assumes $\text{Network } c s t$
\end{verbatim}
Comments

• Locale *Network* fixes flow network

```plaintext
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
• Locale \textit{Network} fixes flow network

\texttt{locale Network}
\begin{itemize}
  \item \texttt{fixes} \( c :: \text{nat} \times \text{nat} \Rightarrow 'capacity \)
  \item \texttt{and} \( s :: \text{nat} \)
  \item \texttt{and} \( t :: \text{nat} \)
\end{itemize}
\texttt{assumes Network c s t}

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

• \texttt{select} \( p. \Phi p \) : Nondeterministically select value that satisfies \( \Phi \)
• Locale *Network* fixes flow network

```plaintext
def locale Network:
  fixes c :: nat × nat ⇒ '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. Φ 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 \( m' \)
  
  \[
  \_ \leq \text{FAIL} \\
  \text{RES } X \leq \text{RES } Y \text{ iff } X \subseteq Y
  \]
Refinement

- Program $m$ refines $m'$, if results of $m$ are also results of $m'$
  
  $- \leq \text{FAIL}$

  $\text{RES } X \leq \text{RES } Y$ iff $X \subseteq Y$

- Special case: correctness of program $m$ wrt. specification $\Phi$

\[
\begin{align*}
\text{Program } m & \text{ refines } m', \text{ if results of } m \text{ are also results of } m' \\
- & \leq \text{FAIL} \\
\text{RES } X & \leq \text{RES } Y \text{ iff } X \subseteq Y \\
\text{Special case: correctness of program } m & \text{ wrt. specification } \Phi
\end{align*}
\]
Refinement

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

• Program $m$ refines $m'$, if results of $m$ are also results of $m'$
  \[ \_ \leq \text{FAIL} \]
  \[ \text{RES } X \leq \text{RES } Y \text{ iff } X \subseteq Y \]

• Special case: correctness of program $m$ wrt. specification $\Phi$
  • All possible results of $m$ satisfy $\Phi$
  • $m \leq \text{RES } (\text{Collect } \Phi)$ (notation: $m \leq (\text{spec } x. \Phi x)$ )
Refinement

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

- $qsort :: int \text{ list} \Rightarrow list \text{ nres}$ correct
Examples

- \textit{qsort} :: \texttt{int list} \Rightarrow \texttt{list nres} \texttt{ correct}

  \[ \text{qsort } l \leq (\texttt{spec } l'. \texttt{mset } l' = \texttt{mset } l \land \texttt{sorted } l') \]
Examples

• *qsort :: int list ⇒ list nres* correct
  \[ qsort \quad l \leq (\texttt{spec} \quad l'. \quad mset \quad l' = mset \quad l \land \texttt{sorted} \quad l') \]

• *sum_up* correct:
Examples

- **qsort :: int list ⇒ list nres** correct
  
  \[
  \text{qsort } l \leq (\text{spec } l'. \text{ mset } l' = \text{mset } l \land \text{sorted } l')
  \]

- **sum_up** correct:
  
  \[
  \text{finite } S \implies \text{sum_up } S \leq (\text{spec } a. \ a = \sum S)
  \]
Examples

- `qsort :: int list \rightarrow\ list nres` correct
  
  \[ qsort l \leq (\textbf{spec } l'. mset l' = mset l \land \text{sorted } l') \]

- `sum_up` correct:

  \[ \text{finite } S \quad \Rightarrow \quad sum_{up} S \leq (\textbf{spec } a. a = \Sigma S) \]

- `get_min :: int set \rightarrow\ int nres` correct
Examples

- \textit{qsort} :: \texttt{int list} \Rightarrow \texttt{list nres} \text{ correct}
  \[ \text{qsort} \ l \leq (\texttt{spec} \ l'. \ \text{mset} \ l' = \text{mset} \ l \land \text{sorted} \ l') \]

- \textit{sum\_up} \text{ correct:}
  \[ \text{finite} \ S \implies \text{sum\_up} \ S \leq (\texttt{spec} \ a. \ a = \sum S) \]

- \textit{get\_min} :: \texttt{int set} \Rightarrow \texttt{int nres} \text{ correct}
  \[ S \neq \{\} \implies \text{get\_min} \ S \leq (\texttt{spec} \ x. \ x \in S \land (\forall y \in S. \ x \leq y)) \]
Correctness of Edmonds-Karp

- Prove \( \text{edmonds\_karp} \leq (\text{spec f. isMaxFlow f}) \)
  - In Network context (precondition!)
Correctness of Edmonds-Karp

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

- How to prove such lemmas?
Correctness of Edmonds-Karp

- Prove $\text{edmonds\_karp} \leq (\text{spec } f. \text{ isMaxFlow } f)$
  - 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 $Q$

• 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!
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!
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:**
  \[ wp \ m_1 (\lambda x. \ wp (m_2 x) \ Q) \implies wp \ (x \leftarrow m_1; \ m_2) \ Q \]
Standard Rules for \( wp \)

- **Sequential composition / bind:**
  \[
  wp \ m_1 \ (\lambda x. \ wp \ (m_2 \ x) \ Q) \implies wp \ (x\leftarrow m_1; \ m_2) \ Q
  \]

- **If-then-else**
  \[
  [ \ b\implies wp \ c_1 \ Q; \ \neg b\implies wp \ c_2 \ Q ] \implies wp \ (if \ b \ then \ c_1 \ else \ c_2) \ Q
  \]
Standard Rules for \( \text{wp} \)

- **Sequential composition / bind:**
  \[
  \text{wp } m_1 \ (\lambda x. \ \text{wp} \ (m_2 \ x) \ Q) \implies \text{wp} \ (x \leftarrow m_1; \ m_2) \ Q
  \]

- **If-then-else**
  \[
  \left[ \begin{array}{c}
  b \implies \text{wp} \ c_1 \ Q; \\
  \neg b \implies \text{wp} \ c_2 \ Q
  \end{array} \right] \implies \text{wp} \ (\textbf{if} \ b \ \textbf{then} \ c_1 \ \textbf{else} \ c_2) \ Q
  \]

- **While**
Standard Rules for $wp$

- **Sequential composition / bind:**
  $$wp \ m \_1 \ (\lambda x. \ wp \ (m \_2 \ x) \ Q) \implies wp \ (x \leftarrow m \_1; \ m \_2) \ Q$$

- **If-then-else**
  $$\left[ \begin{array}{l} b \Longrightarrow wp \ c \_1 \ Q; \ b \not\Longrightarrow wp \ c \_2 \ Q \end{array} \right] \implies wp \ (if \ b \ then \ c \_1 \ else \ c \_2) \ Q$$

- **While**
  
  - **Partial correctness:**
    $$\left[ \begin{array}{l} l \ s \_0; \ \land \ s. \left[ \begin{array}{l} b \ s; \ l \ s \end{array} \right] \ \Longrightarrow \ wp \ (c \ s) \ l \end{array} \right]$$
    $$\quad \implies wp \ (while \ b \ c \ s \_0) \ (\lambda s. \ l \ s \ \land \ b \ s)$$
Standard Rules for \( wp \)

- **Sequential composition / bind:**
  \[
  wp \ m_1 \ (\lambda x. \ wp \ (m_2 \ x) \ Q) \implies wp \ (x\leftarrow m_1; \ m_2) \ Q
  \]

- **If-then-else**
  \[
  \begin{array}{ll}
  & b \implies wp \ c_1 \ Q; \ 
  \neg b \implies wp \ c_2 \ Q \end{array} \implies wp \ (\textbf{if} \ b \ \textbf{then} \ c_1 \ \textbf{else} \ c_2) \ Q
  \]

- **While**
  
  - **Partial correctness:**
    \[
    \begin{array}{ll}
    & I \ s_0; \ \land \ s. \ [ \ b \ s; \ I \ s ] \implies wlp \ (c \ s) \ I \\
    \implies & wlp \ (\textbf{while} \ b \ c \ s_0) \ (\lambda s. \ I \ s \land \neg b \ s)
    \end{array}
    \]

  - **Total correctness:**
    \[
    \begin{array}{ll}
    & \text{wf} <; \ I \ s_0; \ \land \ s. \ [b \ s; \ I \ s] \implies wp \ (c \ s) \ (\lambda s'. \ I \ s' \land s'<s) \\
    \implies & wp \ (\textbf{while} \ b \ c \ s_0) \ (\lambda s. \ I \ s \land \neg b \ s)
    \end{array}
    \]
Standard Rules for $wp$

- Sequential composition / bind:
  
  \[ wp \ m_1 \ (\lambda x. \ wp \ (m_2 \ x) \ Q) \implies wp \ (x \leftarrow m_1; \ m_2) \ Q \]

- If-then-else
  
  \[ \begin{array}{c}
  b \implies wp \ c_1 \ Q; \\
  \neg b \implies wp \ c_2 \ Q
  \end{array} \implies wp \ (\text{if } b \text{ then } c_1 \text{ else } c_2) \ Q \]

- While

  - Partial correctness:
    
    \[ \begin{array}{c}
    l \ s_0; \ \land s. \ \begin{array}{c}
    b \ s; \ l \ s
    \end{array} \implies wlp \ (c \ s) \ l
    \end{array} \implies wlp \ (\text{while } b \ c \ s_0) \ (\lambda s. \ l \ s \land \neg b \ s) \]

  - Total correctness:
    
    \[ \begin{array}{c}
    wf <; \ l \ s_0; \ \land s. \ \begin{array}{c}
    b \ s; \ l \ s
    \end{array} \implies wp \ (c \ s) \ (\lambda s'. \ l \ s' \land s'<s)
    \end{array} \implies wp \ (\text{while } b \ c \ s_0) \ (\lambda s. \ l \ s \land \neg b \ s) \]

  - add consequence rule
    
    \[ \begin{array}{c}
    wf <; \ l \ s_0; \ \land s. \ \begin{array}{c}
    b \ s; \ l \ s
    \end{array} \implies wp \ (c \ s) \ (\lambda s'. \ l \ s' \land s'<s); \\
    \land s. \ l \ s \land \neg b \ s \implies Q \ s
    \end{array} \implies wp \ (\text{while } b \ c \ s_0) \ Q \]

31
Rules for Nres-Monad

\( \Phi x \implies \text{return } x \leq (\text{spec } x. \ \Phi x) \)
Rules for Nres-Monad

\[ \Phi \ x \implies \text{return } \ x \leq (\text{spec } x. \ \Phi \ x) \]

\[ m \leq (\text{spec } x. \ f \ x \leq (\text{spec } y. \ \Phi \ y)) \implies m \gg (\lambda x. \ f \ x) \leq (\text{spec } y. \ \Phi \ y) \]
Rules for Nres-Monad

Φ x \implies \textbf{return} x \leq (\textbf{spec} x. \Phi x)

m \leq (\textbf{spec} x. f x \leq (\textbf{spec} y. \Phi y)) \implies m \gg (\lambda x. f x) \leq (\textbf{spec} y. \Phi y)

\begin{align*}
[b \implies m_1 \leq (\textbf{spec} x. \Phi x); \neg b \implies m_2 \leq (\textbf{spec} x. \Phi x)] \\
\implies (\textbf{if} \ b \textbf{ then} \ m_1 \textbf{ else} \ m_2) \leq (\textbf{spec} x. \Phi x)
\end{align*}
Rules for Nres-Monad

\[ \Phi \ x \ \Rightarrow \ return \ x \leq (\text{spec } x. \ \Phi \ x) \]

\[ m \leq (\text{spec } x. \ f \ x \leq (\text{spec } y. \ \Phi \ y)) \Rightarrow m \Rightarrow (\lambda x. \ f \ x) \leq (\text{spec } y. \ \Phi \ y) \]

\[ [\ [ b \Rightarrow m_1 \leq (\text{spec } x. \ \Phi \ x); \ \neg \ b \Rightarrow m_2 \leq (\text{spec } x. \ \Phi \ x) \] ] \Rightarrow (\text{if } b \text{ then } m_1 \text{ else } m_2) \leq (\text{spec } x. \ \Phi \ x) \]

\[ [\ [ \text{wf } R; \ l \ s; \ \land s. \ [l \ s; \ b \ s] \Rightarrow f \ s \leq (\text{spec } s'. \ l \ s' \land (s', s) \in R); \ 
\land s. \ [l \ s; \ \neg b \ s] \Rightarrow \Phi \ s] ] \Rightarrow \text{while}_T \ b \ f \ s \leq (\text{spec } s. \ \Phi \ s) \]
Rules for Nres-Monad

\[ \Phi x \implies \text{return } x \leq (\text{spec } x. \Phi x) \]

\[ m \leq (\text{spec } x. f x \leq (\text{spec } y. \Phi y)) \implies m \gg (\lambda x. f x) \leq (\text{spec } y. \Phi y) \]

\[ [b \implies m_1 \leq (\text{spec } x. \Phi x); \neg b \implies m_2 \leq (\text{spec } x. \Phi x)] \implies (\text{if } b \text{ then } m_1 \text{ else } m_2) \leq (\text{spec } x. \Phi x) \]

\[ [\text{wf } R; l s; \land s. [l s; b s] \implies f s \leq (\text{spec } s'. l s' \land (s', s) \in R); \land s. [l s; \neg b s] \implies \Phi s] \implies \text{while}_T b f s \leq (\text{spec } s. \Phi s) \]

...
Verification Condition Generator

• Apply rules repeatedly.

\[ \text{spec}_x \cdot \Phi_x \land \forall x. \Phi_x \Rightarrow \Psi_x \Rightarrow m \leq (\text{spec}_x \cdot \Psi_x) \]

• Stop when no rule applies

• Subgoal is not of shape \( \leq \text{spec}_x \)

• Missing rule, e.g. for user-defined function

• Prove the generated VCs

• Using Isabelle's standard proof methods
Verification Condition Generator

- Apply rules repeatedly.
- Rule to be applied determined by topmost statement
  - Automatically!
  - May have to apply consequence rule before
  
\[ m \leq (\text{spec } x \cdot \Phi x) \land \forall x. \Phi x \Rightarrow \Psi x \implies m \leq (\text{spec } x \cdot \Psi x) \]
Verification Condition Generator

• Apply rules repeatedly.
• Rule to be applied determined by topmost statement
  • Automatically!
  • May have to apply consequence rule before
    \[ [m \leq (\text{spec } x \cdot \Phi x) ; \land x \cdot \Phi x \Rightarrow \Psi x] \Rightarrow m \leq (\text{spec } x \cdot \Psi x) \]
• Stop when no rule applies
  • Subgoal is not of shape _ \leq \text{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
    
    \[ \lfloor m \leq (\text{spec } x. \Phi x); \land x. \Phi x \implies \Psi x \rfloor \implies m \leq (\text{spec } x. \Psi x) \]

- Stop when no rule applies
  - Subgoal is not of shape \_ \leq \text{spec } \_ \_ (cf. RETURN_rule)
  - Missing rule, e.g. for user-defined function

- Prove the generated VCs
  - Using Isabelle’s standard proof methods
Finding good invariant is usually most creative task
Invariants

- Finding good invariant is usually most creative task
- Recall: Invariant for while-loop must
• Finding good invariant is usually most creative task
• Recall: Invariant for while-loop must
  • Hold initially ($I_{s_0}$)
Finding good invariant is usually most creative task

Recall: Invariant for while-loop must

- Hold initially ($I_s^0$)
- Be preserved by loop iteration ($[I_s; b] \implies f_s \leq \text{spec } l$)
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 ($[l; b] \implies f \leq \text{spec } l$)
  - Imply postcondition when loop terminates ($[l; \neg b] \implies \Phi$)
Invariant Examples

Note: informal syntax!

• \( a=0; \textbf{while } S \neq \{\} \ \textbf{do } \{ \ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{x\}; \ a = a + x \} \)

Specification: \( \textit{finite } S_0 \implies a = \sum S_0 \)

Invariant:
Invariant Examples

Note: informal syntax!

• $a=0; \textbf{while } S\neq \{\} \textbf{ do } \{ x\leftarrow \text{spec } x. \ x \in S; \ S=S\setminus \{x\}; \ a=a+x \}$
  
  Specification: \textit{finite } $S_0 \implies a = \sum S_0$
  
  Invariant: $a = \sum (S_0 - S)$
Invariant Examples

Note: informal syntax!

• \( a=0; \textbf{while } S\neq \{\} \textbf{ do } \{ \ x\leftarrow \text{spec } x. \ x\in S; \ S=S-\{x\}; \ a=a+x \} \)
  
  Specification: finite \( S_0 \implies a = \sum S_0 \)

  Invariant: \( a = \sum (S_0 - S) \) sufficient?
Note: informal syntax!

- \( a=0; \textbf{while} \; S \neq \{\} \; \textbf{do} \; \{ \ \text{\texttt{x}} \leftarrow \textbf{spec} \; \texttt{x}. \; x \in S; \; S = S \setminus \{x\}; \; a = a + x \} \)

  Specification: \( \textit{finite } S_0 \implies a = \sum S_0 \)

  Invariant: \( a = \sum (S_0 - S) \ \land \ S \subseteq S_0 \)
Invariant Examples

Note: informal syntax!

- $a=0; \textbf{while } S \neq \{\} \textbf{ do } \{ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{x\}; \ a = a + x \}$
  Specification: finite $S_0 \implies a = \sum S_0$
  Invariant: $a = \sum (S_0 - S) \land S \subseteq S_0$

- $f = False; \textbf{while } \neg f \textbf{ do } \{ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{x\}; \ f = (x == 42) \}$
  Specification: finite $S_0 \implies f = 42 \in S_0$
  Invariant:
Invariant Examples

Note: informal syntax!

• \( a=0; \textbf{while} \ S\neq\{} \textbf{do} \{ \ x\leftarrow \text{spec} \ x. \ x\in S; \ S=S\setminus\{x\}; \ a=a+x \} \)
  
  Specification: \( \text{finite } S_0 \implies a = \sum S_0 \)
  
  Invariant: \( a = \sum (S_0 - S) \wedge S \subseteq S_0 \)

• \( f=False; \textbf{while} \ \neg f \textbf{ do} \{ \ x\leftarrow \text{spec} \ x. \ x\in S; \ S=S\setminus\{x\}; \ f=(x==42) \} \)
  
  Specification: \( \text{finite } S_0 \implies f = 42 \in S_0 \)
  
  Invariant: \( f = x \in (S_0 - S) \wedge S \subseteq S_0 \)
Invariant Examples

Note: informal syntax!

- \( a=0; \textbf{while } S \neq \{\} \textbf{ do } \{ \ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{x\}; \ a = a + x \} \)
  Specification: \( \text{finite } S_0 \Rightarrow a = \sum S_0 \)
  Invariant: \( a = \sum (S_0 - S) \land S \subseteq S_0 \)

- \( f=\text{False}; \textbf{while } \neg f \textbf{ do } \{ \ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{x\}; \ f = (x == 42) \} \)
  Specification: \( \text{finite } S_0 \Rightarrow f = 42 \in S_0 \)
  Invariant: \( f = x \in (S_0 - S) \land S \subseteq S_0 \)

- \( \textbf{while } a \neq b \textbf{ do } \{ \textbf{if } a < b \textbf{ then } b = b - a \textbf{ else } a = a - b \} \)
  Specification: \( [a_0 > 0; \ b_0 > 0] \Rightarrow a = \text{gcd } a_0 \ b_0 \)
  Invariant:
Note: informal syntax!

- \(a=0; \textbf{while} \ S \neq \{\} \ \textbf{do} \ \{ \ x \leftarrow \text{spec} \ x. \ x \in S; \ S = S - \{x\}; \ a = a + x \} \)
  
  Specification: \(\text{finite } S_0 \iff a = \sum S_0\)
  
  Invariant: \(a = \sum (S_0 - S) \land S \subseteq S_0\)

- \(f = \text{False}; \textbf{while} \ \neg f \ \textbf{do} \ \{ \ x \leftarrow \text{spec} \ x. \ x \in S; \ S = S - \{x\}; \ f = (x = 42) \} \)
  
  Specification: \(\text{finite } S_0 \iff f = 42 \in S_0\)
  
  Invariant: \(f = x \in (S_0 - S) \land S \subseteq S_0\)

- \(\textbf{while} \ a \neq b \ \textbf{do} \ \{ \textbf{if} \ a < b \ \textbf{then} \ b = b - a \ \textbf{else} \ a = a - b \} \)
  
  Specification: \([a_0 > 0; b_0 > 0] \iff a = \gcd a_0 b_0\)
  
  Invariant: \(\gcd a b = \gcd a_0 b_0\)
Invariant Examples

Note: informal syntax!

- $a=0; \textbf{while } S \neq \emptyset \textbf{ do } \{ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{ x \}; \ a = a+x \}$
  
  Specification: $\text{finite } S_0 \implies a = \sum S_0$
  
  Invariant: $a = \sum (S_0 - S) \land S \subseteq S_0$

- $f=True; \textbf{while } \neg f \textbf{ do } \{ x \leftarrow \text{spec } x. \ x \in S; \ S = S - \{ x \}; \ f = (x==42) \}$

  Specification: $\text{finite } S_0 \implies f = 42 \in S_0$
  
  Invariant: $f = x \in (S_0 - S) \land S \subseteq S_0$

- $\textbf{while } a \neq b \textbf{ do } \{ \textbf{if } a < b \textbf{ then } b = b - a \textbf{ else } a = a - b \}$

  Specification: $[a_0 > 0; \ b_0 > 0] \implies a = \gcd a_0 \ b_0$
  
  Invariant: $\gcd a \ b = \gcd a_0 \ b_0 \land a > 0 \land b > 0$
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
Prove lemma: discharge subgoals

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

Rule $[P_1; \ldots; P_n] \rightarrow Q$
Backwards Verification in Isabelle

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

• Other proof methods to work on subgoals
  • \textit{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 $\square[P_1; \ldots; P_n] \Rightarrow Q$
  • Replace subgoal $Q$ with new subgoals $P_1, \ldots, P_n$ (unification!)

• Other proof methods to work on subgoals
  • \textit{auto}, \textit{simp}, \ldots \ Try to solve, present unsolvable parts

• Other useful tools
  • \textit{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 $[P_1; \ldots; P_n] \Rightarrow Q$
  • Replace subgoal $Q$ with new subgoals $P_1$, $\ldots$, $P_n$ (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 \([P_1; \ldots; P_n] \Rightarrow Q\)
  • Replace subgoal \(Q\) with new subgoals \(P_1, ..., P_n\) (unification!)

• Other proof methods to work on subgoals
  • \texttt{auto, simp, ...} Try to solve, present unsolvable parts

• Other useful tools
  • \texttt{sledgehammer} Run SMT solvers, replay proof in Isabelle kernel
  • \texttt{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
      $V = V \cup \{u\}$
      $W = W \cup \{ v. (u,v) \in E \}$
  end
  ```

  ```
  return $V$
  ```
A Complete Example: State-Space Search

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

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

  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 $V$
  • $V \subseteq \text{reachable}$ 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 $V$
  • $V \subseteq \text{reachable}$ 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 $V \supseteq \text{reachable}$
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 $V$
  - $V \subseteq \text{reachable}$ 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 $V \supseteq \text{reachable}$
- Termination: Only if set of reachable nodes is finite
  - $V \subseteq \text{reachable}$ increases, or $V$ remains unchanged and $W \subseteq \text{reachable}$ 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

Example: \( \text{BFS } g \ s \ t \leq \text{select } p \). \text{Graph.} \text{isShortestPath } g \ s \ p \ t \)

Note: \((\text{select } p \cdot \Phi p) = (\text{spec } r \cdot \text{case } r \of \text{None} \Rightarrow \nexists p \cdot \Phi p \mid \text{Some } p \Rightarrow \Phi p)\)
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. \( \text{BFS} \ g \ s \ t \leq \text{select} \ p. \ \text{Graph.isShortestPath} \ g \ s \ p \ t \)
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. \( \text{BFS} \ g \ s \ t \leq \text{select} \ p. \ \text{Graph.isShortestPath} \ g \ s \ p \ t \)
  • Note: \((\text{select} \ p. \ \Phi \ p) = (\text{spec} \ r. \ \text{case} \ r \ of \ \text{None} \Rightarrow \nexists \ p. \ \Phi \ p \mid \text{Some} \ p \Rightarrow \Phi \ p)\)
• 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. $BFS \; g \; s \; t \leq \text{select } p. \; \text{Graph.isShortestPath} \; g \; s \; p \; t$
  • Note: $(\text{select } p. \; \Phi \; p) = (\text{spec } r. \; \text{case } r \; \text{of } \; \text{None } \Rightarrow \notin p. \; \Phi \; p \; | \; \text{Some } \; p \; \Rightarrow \; \Phi \; p)$
• 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. $BFS \ g \ s \ t \ \leq \ select \ p.\ \ Graph.isShortestPath \ g \ s \ p \ t$
  • Note: $(select\ p.\ \Phi\ p) = (spec\ r.\ case\ r\ of\ None \Rightarrow \not\exists\ p.\ \Phi\ p \mid Some\ p \Rightarrow \Phi\ p)$
• That’s easy! But what if data representation changes?
Using Residual Graph

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

  \[ p \leftarrow \textbf{select } p. \text{Graph.isShortestPath (residualGraph c f) s p t} \]
Using Residual Graph

- Our current `edmonds_karp` computes residual graph in each iteration
  
  \[ p \leftarrow \text{select } p. \ Graph.\text{isShortestPath}(\text{residualGraph}\ c\ f)\ s\ p\ t \]

- Nice for correctness proof.
Using Residual Graph

- Our current \texttt{edmonds_karp} computes residual graph in each iteration
  \[ p \leftarrow \textbf{select} \ p. \ Graph.\texttt{isShortestPath}(\texttt{residualGraph} \ c \ f) \ s \ p \ t \]
- Nice for correctness proof. But very inefficient!
Using Residual Graph

- Our current `edmonds_karp` computes residual graph in each iteration
  
  \[ p \leftarrow \text{select } p. \text{Graph.isShortestPath}(\text{residualGraph } c \ f) \ s \ p \ t \]

- 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
  
  \[ p \leftarrow \text{select } p. \text{Graph.isShortestPath (residualGraph c f) s p t} \]

- 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:

```plaintext
let f = (λ_. 0);
(f,_) ← while
   (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:

```plaintext
let cf = c;
(cf,_) ← while_T (
    (λ(cf,brk). ¬brk)
) (λ(cf,_). do {
    p ← select p. Graph.isShortestPath cf s p t;
    case p of
    None ⇒ return (cf,True)
    Some p ⇒ do {
        let cf = Graph.augment_cf cf (set p) (resCap_cf cf p);
        return (cf, False)
    }
})
(cf,False);
return (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 \textit{edmonds\_karp2} to \textit{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

- \( \text{flow}_\text{of}_\text{cf} : (\text{nat} \times \text{nat} \Rightarrow '\text{capacity}) \Rightarrow \text{nat} \times \text{nat} \Rightarrow '\text{capacity} \)

  convert residual graph to flow
Relation between Residual Graph and Flow

- $\text{flow}_\text{of}_\text{cf} : (\text{nat} \times \text{nat} \Rightarrow '\text{capacity}) \Rightarrow \text{nat} \times \text{nat} \Rightarrow '\text{capacity}$
  convert residual graph to flow

- $\text{cf}_\text{i}_\text{rel} = \{(\text{cf}, \text{flow}_\text{of}_\text{cf} \ \text{cf}) \mid \text{cf. RGraph } c \ s \ t \ \text{cf}\}$
Relation between Residual Graph and Flow

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

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

- \textit{cfi\_rel} = \{ (\text{cf}, \text{flow\_of\_cf \ cf}) \mid \text{cf. RGraph c s t cf} \}
  
  Relation consists of 
  \begin{itemize}
    \item \textit{abstraction function} (\text{flow\_of\_cf})
    \item \textit{invariant} (\text{RGraph c s t})
  \end{itemize}

  This pattern occurs frequently. Shortcut: 
  \textit{cfi\_rel} \equiv \text{br flow\_of\_cf} (\text{RGraph c s t})
Relating Operations

• Show, for each operation: inputs related $\implies$ outputs related
Relating Operations

• Show, for each operation: inputs related $\implies$ outputs related
• Initial flow: Straightforward
  $(c, \lambda_. 0::'capacity) \in cfi_{rel}$
Relating Operations

• Show, for each operation: inputs related $\implies$ outputs related

• Initial flow: Straightforward
  
  \[(c, \lambda_0::'capacity) \in cfi_{rel}\]

• Augmentation of flow:
  
  \[\[(cf, f) \in cfi_{rel}; NPreflow.isAugmentingPath c s t f p]\]
  \[\implies (Graph.augment_cf cf (set p) (resCap_cf cf p),\]
  \[NFlow.augment_with_path c f p) \in cfi_{rel}\]
Relating Operations

- Show, for each operation: inputs related $\implies$ outputs related
- Initial flow: Straightforward
  \[(c, \lambda. 0::'capacity) \in cfi_{rel}\]
- Augmentation of flow:
  
  \[
  \begin{array}{c}
  \left[(cf, f) \in cfi_{rel}; \text{NPreflow.isAugmentingPath } c \ s \ t \ f \ p\right] \\
  \implies \left(\text{Graph.augment_cf } cf (\text{set } p) (\text{resCap_cf } cf \ p), \right. \\
  \text{NFlow.augment_with_path } c \ f \ p \\
  \left. \in cfi_{rel}\right)
  \end{array}
  \]
- Note the isAugmentingPath precondition!
Relating Operations

• Show, for each operation: inputs related $\implies$ outputs related

• Initial flow: Straightforward

$(c, \lambda_. \, 0::'capacity) \in cfi_{rel}$

• Augmentation of flow:

$[(cf, f) \in cfi_{rel}; \text{NPreflow.isAugmentingPath } c \, s \, t \, f \, p] \implies (\text{Graph.augment_cf } cf \, (\text{set } p) \, (\text{resCap_cf } cf \, p), \text{NFlow.augment_with_path } c \, f \, p) \in cfi_{rel}$

• Note the $\text{isAugmentingPath}$ 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 → 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 → residual graph

• We need to lift relation on result types to relation on \( nres \)
  • Each concrete result related to some abstract result
  • Given relation \( R :: (’c, ’a) \text{ set} \), 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 $R :: (\text{'c,}'a) \text{ set}$, and $m_1::\text{'c nres}$, $m_2::\text{'a nres}$
  • $m_1$ related to $m_2$, if
    • $m_2 = \text{FAIL}$, or
    • $m_1 = \text{RES } X$, $m_2 = \text{RES } Y$, and $\forall x \in X. \exists y \in Y. (x,y) \in R$

• Can be expressed as refinement
  $m_1 \leq \Downarrow R m_2$, where $\Downarrow R \text{ FAIL} = \text{FAIL}$
  $\Downarrow R (\text{spec } x. \exists y \in Y. (x,y) \in R)$
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
    - $m_2 = \text{FAIL}$, or
    - $m_1 = \text{RES } X$, $m_2 = \text{RES } Y$, and $\forall x \in X. \exists y \in Y. (x, y) \in R$
- Can be expressed as refinement
  - $m_1 \leq \downarrow R m_2$, where
    - $\downarrow R \text{FAIL} = \text{FAIL}$
    - $\downarrow R (\text{RES } Y) = (\text{spec } x. \exists y \in Y. (x, y) \in R)$
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
  \[(x_1, x_2) \in R \implies \text{return } x_1 \leq \downarrow R (\text{return } x_2)\]
Proving Relation Between Programs

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

- Some rules

\[(x_1, x_2) \in R \implies \text{return } x_1 \leq \downarrow R \left(\text{return } x_2\right)\]

\[
\left[ m_1 \leq \downarrow R \ m_2; \ \land x_1 \ x_2. \ (x_1, x_2) \in R \implies f_1 \ x_1 \leq \downarrow R' \ (f_2 \ x_2)\right] \implies m_1 \gg f_1 \leq \downarrow R' \ (m_2 \gg f_2)\]
Proving Relation Between Programs

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

- Some rules

\[(x_1, x_2) \in R \implies \text{return } x_1 \leq \downarrow R (\text{return } x_2)\]

\[[m_1 \leq \downarrow R m_2; \land x_1 x_2. (x_1, x_2) \in R \implies f_1 x_1 \leq \downarrow R' (f_2 x_2)]\]
\[\implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2)\]

\[[ (x, x') \in R; \land x x'. (x, x') \in R \implies b x = b' x';
\land x x'. [(x, x') \in R; b x; b' x'] \implies f x \leq \downarrow R (f' x')]\]
\[\implies \text{while } b f x \leq \downarrow R (\text{while } b' f' x')\]
• Prove $m_1 \leq \downarrow R m_2$, where $m_1$ and $m_2$ have same structure
• Prove $m_1 \leq \downarrow R m_2$, where $m_1$ and $m_2$ have same structure
• Repeatedly apply rules for combinators
• 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
• 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
• 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
• 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
• Recall bind-refine rule

\[
\begin{align*}
[m_1 \leq \downarrow R m_2; \wedge x_1 x_2. (x_1, x_2) \in R \implies f_1 x_1 \leq \downarrow R' (f_2 x_2)] \\
\implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2)
\end{align*}
\]
• Recall bind-refine rule

\[ [m_1 \leq \downarrow R m_2; \ \land x_1 \ x_2. (x_1, x_2) \in R \implies f_1 \ x_1 \leq \downarrow R' (f_2 \ x_2)] \]
\[ \implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2) \]

• What relation $R$ should we choose?
Unknown Relations

• Recall bind-refine rule

\[ \[ m_1 \leq \downarrow R m_2; \ \land x_1 \ x_2. \ (x_1, \ x_2) \in R \implies f_1 \ x_1 \leq \downarrow R' (f_2 \ x_2) \] \]
\[ \implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2) \]

• What relation \( R \) should we choose?
  • We don’t know!
• Recall bind-refine rule

\[ m_1 \leq \downarrow R m_2; \land x_1 x_2. (x_1, x_2) \in R \implies f_1 x_1 \leq \downarrow R' (f_2 x_2) \]
\[ \implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2) \]

• What relation \( R \) should we choose?
  • We don’t know! (yet)
• Recall bind-refine rule

\[ m_1 \leq \downarrow R m_2; \ \land x_1 \ x_2. (x_1, x_2) \in R \implies f_1 \ x_1 \leq \downarrow R' (f_2 \ x_2) \]

\[ \implies m_1 \gg f_1 \leq \downarrow R' (m_2 \gg f_2) \]

• What relation \( R \) should we choose?
  • We don’t know! (yet)

• In practice: Guess relation from type
Side Conditions

- Recall theorem for augmentation refinement

\[(cf, f) \in cfi_{rel}; \ NPreflow.isAugmentingPath c s t f p\]
\[\implies (Graph.augment_{cf} cf (set p) (resCap_{cf} cf p),\]
\[\quad NFlow.augment_{with\_path} c f p\]
\[\in cfi_{rel}\]
Side Conditions

• Recall theorem for augmentation refinement

\[
\begin{align*}
&[(cf, f) \in cfi_{rel}; \ NPreflow.isAugmentingPath c s t f p] \\
\implies (Graph.augment_cf cf (set p) (resCap_cf cf p), \\
&\quad NFlow.augment_with_path c f p) \\
&\in cfi_{rel}
\end{align*}
\]

• We do not refine paths at all
Side Conditions

- Recall theorem for augmentation refinement
  \[
  [(cf, f) \in \text{cf}_\text{rel}; \text{NPreflow.isAugmentingPath} c s t f p] \implies \text{Graph.augment}_\text{cf} cf (\text{set} p) \text{ (resCap}_\text{cf} cf p), \text{NFlow.augment}_\text{with_path} c f p) \in \text{cf}_\text{rel}
  \]

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

- Recall theorem for augmentation refinement
  \[
  \llbracket (cf, f) \in cfi_{rel}; NPreflow.isAugmentingPath c s t f p \rrbracket 
  \implies (Graph.augment_cf cf (set p) (resCap_cf cf p),
  NFlow.augment_with_path c f p) \in cfi_{rel}
  \]

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

- Solution: assertions and congruence rules
• assert $\Phi = (\text{if } \Phi \text{ then return } () \text{ else } FAIL)$
• assert $\Phi = (\text{if } \Phi \text{ then return } () \text{ else } FAIL)$
  • $[\Phi; \Phi \Rightarrow \Psi ()] \Rightarrow \text{assert } \Phi \leq \text{SPEC } \Psi$
• **assert** \( \Phi = (\text{if } \Phi \text{ then return } () \text{ else } \text{FAIL}) \)

• \([\Phi; \Phi \Rightarrow \Psi ()] \Rightarrow \text{assert } \Phi \leq \text{SPEC } \Psi\)

• Insert **assert** \((\text{isAugmentingPath } c s t f p)\) to abstract algorithm
• assert $Φ = (if Φ then return () else FAIL)$

• $[Φ; Φ \rightarrow Ψ ()] \rightarrow assert Φ \leq SPEC Ψ$

• Insert $assert (isAugmentingPath c s t f p)$ to abstract algorithm
  • Easy to prove there!
• assert $\Phi = (\text{if } \Phi \text{ then return } () \text{ else } \text{FAIL})$

- $[\Phi; \Phi \implies \Psi ()] \implies \text{assert } \Phi \leq \text{SPEC } \Psi$
- Insert assert ($\text{isAugmentingPath } c s t f p$) to abstract algorithm
  - Easy to prove there!

• $(\Phi \implies m_1 \leq \Downarrow R m_2) \implies m_1 \leq \Downarrow R (\text{assert } \Phi \gg (\lambda_. m_2))$
• **assert** $\Phi = (\text{if } \Phi \text{ then return } () \text{ else } \text{FAIL})$
  
  • $[\Phi; \Phi \implies \Psi ()] \implies \text{assert } \Phi \leq \text{SPEC } \Psi$
  
  • Insert **assert** $(\text{isAugmentingPath } c s t f p)$ to abstract algorithm
    
    • Easy to prove there!
  
  • $(\Phi \implies m_1 \leq \downarrow R m_2) \implies m_1 \leq \downarrow R (\text{assert } \Phi \gg (\lambda_. m_2))$
    
    • During refinement, we can assume $\Phi$
• **assert** $\Phi = (\text{if } \Phi \text{ then return } () \text{ else } FAIL)$
  
  - $[\Phi; \Phi \implies \Psi ()] \implies \text{assert } \Phi \leq \text{SPEC } \Psi$
  
  • Insert **assert** $(\text{isAugmentingPath } c s t f p)$ to abstract algorithm
    
    - Easy to prove there!

- $(\Phi \implies m_1 \leq \downarrow R m_2) \implies m_1 \leq \downarrow R (\text{assert } \Phi \gg (\lambda. m_2))$
  
  - During refinement, we can **assume** $\Phi$

- Assertions transport knowledge down the refinement chain
• Consider $\textbf{if } b \textbf{ then } m_1 \textbf{ else } m_2$
• Consider \textbf{if} \( b \) \textbf{then} \( m_1 \) \textbf{else} \( m_2 \)
  
  • We can assume \( b / \neg b \) for refinement of \( m_1 / m_2 \)
Congruences

- Consider \textbf{if } \textit{b} \textbf{then } \textit{m}_1 \textbf{ else } \textit{m}_2
  - We can assume \textit{b }\neg \textit{b} for refinement of \textit{m}_1/\textit{m}_2
  - Similar for \textbf{while}_T, \textbf{foreach}_T, \textbf{case}, \textbf{bind}, \textbf{assert}, \ldots

\[ [\neg b \mathrel{\Rightarrow} \neg b] = \Rightarrow (\text{if } b \textbf{ then } S_1 \textbf{ else } S_2) \leq \lll R (\text{if } b' \textbf{ then } S_1' \textbf{ else } S_2') \]
• Consider \textbf{if} \( b \) \textbf{then} \( m_1 \) \textbf{else} \( m_2 \)
  
  • We can assume \( b \land \neg b \) for refinement of \( m_1/m_2 \)
  • Similar for \texttt{while}_T, \texttt{foreach}_T, \texttt{case}, \texttt{bind}, \texttt{assert}, \ldots

• Requires strengthened refinement rules for these combinators
• Consider **if** *b* **then** *m₁* **else** *m₂*
  
  • We can assume *b* \(\neg b\) for refinement of *m₁/m₂*
  
  • Similar for **while**₁, **foreach**₁, **case**, **bind**, **assert**, ...
  
  • Requires strengthened refinement rules for these combinators

\[
\begin{align*}
[b = b'; \ [b; b'] & \implies S₁ \leq \downarrow R S₁'; \ [\neg b; \neg b'] & \implies S₂ \leq \downarrow R S₂'] \\
\implies (if \ b \ then \ S₁ \ else \ S₂) & \leq \downarrow R (if \ b' \ then \ S₁' \ else \ S₂')
\end{align*}
\]
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 \textbf{return} \((a \cdot b, a \cdot b + 1)\) and \textit{let} \(x = a \cdot b\) in return \((x, x + 1)\)
  • Application of rules will get stuck

• Manually align programs (unfold let)
  • Isabelle has powerful \textit{subst} and \textit{rewrite}

• Set of \textbf{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
    \[ f \times \leq \Downarrow R M' \implies \text{Let} \times f \leq \Downarrow R M' \]
Slightly Different Program Structure

- Consider \textbf{return} \((a*b, a*b+1)\) and \textit{let} \(x=a*b\ \text{in} \ \text{return} \ (x, x+1)\)
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - Isabelle has powerful \texttt{subst} and \texttt{rewrite}
- Set of recovery rules built in VCG
  - eg
    \[
    f \times x \leq \Downarrow RM' \implies Let x f \leq \Downarrow RM'
    \]
- Convert any refinement goal to first-order formula and solve
Slightly Different Program Structure

- Consider `return (a\times b, a\times b + 1)` and `let x = a\times 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
    \[ f \times x \leq \downarrow R M' \implies Let \times f \leq \downarrow R M' \]
- 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 `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
    
    \[ f \times \leq \Downarrow R M' \implies Let \times f \leq \Downarrow R M' \]
- Convert any refinement goal to first-order formula and solve
  - Needs to be invoked manually
  - Works well for non-recursive programs
Slightly Different Program Structure

- Consider \textbf{return} \((a \times b, a \times b + 1)\) and \textit{let} \(x = a \times b\) \textit{in} \textbf{return} \((x, x + 1)\)
  - Application of rules will get stuck
- Manually align programs (unfold let)
  - \textit{Isabelle} has powerful \textit{subst} and \textit{rewrite}
- Set of \textit{recovery rules} built in \textit{VCG}
  - \textit{eg}
    \[
    f \times x \leq \Downarrow R M' \implies \text{Let } x \times f \leq \Downarrow R M'
    \]
- 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:
    $$R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2). (c_1, a_1) \in R_1 \land (c_2, a_2) \in R_2\}$$
Excursion: Natural Relators

• Given $R_1::(\langle'c_1 \times 'a_1\rangle$ set and $R_2::(\langle'c_2 \times 'a_2\rangle$ set

• How are pairs $\langle'c_1 \times 'c_2\rangle$ related to pairs $\langle'a_1 \times 'a_2\rangle$?

  • Product relation:
    
    $$R_1 \times_r R_2 = \{(\langle c_1, c_2 \rangle, a_1, a_2). (c_1, a_1) \in R_1 \land (c_2, a_2) \in R_2\}$$

• 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:
    \[
    R_1 \times_r R_2 = \{((c_1, c_2), a_1, a_2). (c_1, a_1) \in R_1 \land (c_2, a_2) \in R_2\}
    \]

• 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 \( R_1:('c_1 \times 'a_1) \text{ set} \) and \( R_2:('c_2 \times 'a_2) \text{ set} \)

• How are pairs \('c_1 \times 'c_2\) related to pairs \('a_1 \times 'a_2\)?

  • Product relation:
    \[
    R_1 \times_r R_2 = \{(c_1, c_2, a_1, a_2). (c_1, a_1) \in R_1 \land (c_2, a_2) \in R_2\}
    \]

• Many types have such a natural relator
  
  • Algebraic datatypes: Same structure, elements related
    
    • e.g. \( \langle R \rangle \text{list\_rel} \) for \('a \text{ list}'\), \( \langle R_1, R_2 \rangle \text{sum\_rel} \) for \('a + 'b\)
  
  • Functions: Argument related \(\implies\) 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 $R_1::(\text{'c}_1 \times \text{'a}_1) \text{ set}$ and $R_2::(\text{'c}_2 \times \text{'a}_2) \text{ set}$
- How are pairs $\text{'c}_1 \times \text{'c}_2$ related to pairs $\text{'a}_1 \times \text{'a}_2$ ?
  - 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. $\langle R \rangle \text{ list\_rel}$ for $\text{'a \ list}$, $\langle R_1, R_2 \rangle \text{ sum\_rel}$ for $\text{'a + 'b}$
  - Functions: Argument related $\implies$ 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 $\text{'a nres}$
    - $((m_1, m_2) \in \langle R \rangle \text{nres\_rel}) = (m_1 \leq \downarrow R m_2)$
Parametricity Examples

- First element of pair: \( \text{fst}: \text{'a} \times \text{'b} \Rightarrow \text{'a} \)
Parametricity Examples

• First element of pair: \( \text{fst} :: 'a \times 'b \Rightarrow 'a \)
  
  \((\text{fst}, \text{fst}) \in A \times_B B \rightarrow A\)
Parametricity Examples

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

- Append two lists: \( (@) : ('a \text{ list}) \Rightarrow ('a \text{ list}) \Rightarrow ('a \text{ list}) \)
Parametricity Examples

- First element of pair: $\text{fst} : \mathcal{P} \mathcal{A} \times \mathcal{P} \mathcal{B} \Rightarrow \mathcal{P} \mathcal{A}$
  $\langle \text{fst, fst} \rangle \in A \times_r B \rightarrow A$

- Append two lists: $\langle @ \rangle \mathcal{P} \mathcal{A} \mathcal{L} \Rightarrow \mathcal{P} \mathcal{A} \mathcal{L} \Rightarrow \mathcal{P} \mathcal{A} \mathcal{L}$
  $\langle \langle @ \rangle, \langle @ \rangle \rangle \in \langle A \rangle \mathcal{L} \mathcal{R} \rightarrow \langle A \rangle \mathcal{L} \mathcal{R} \rightarrow \langle A \rangle \mathcal{L} \mathcal{R}$
Parametricity Examples

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

- Append two lists: \( (@)::'a \text{ list } \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list } \)
  
  \( ((@), (@)) \in \langle A \rangle \text{ list}_\text{rel} \rightarrow \langle A \rangle \text{ list}_\text{rel} \rightarrow \langle A \rangle \text{ list}_\text{rel} \)

- Limitations in HOL
Parametricity Examples

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

- Append two lists: \(\text{@} : \texttt{'}a \text{ list} \Rightarrow \texttt{'}a \text{ list} \Rightarrow \texttt{'}a \text{ list} \)
  \((\text{@}, \text{@}) \in \langle A \rangle \text{list rel} \to \langle A \rangle \text{list rel} \to \langle A \rangle \text{list rel}\)

- Limitations in HOL
  - Partiality/underdefinedness: \(\text{hd} : \texttt{'}a \text{ list} \Rightarrow \texttt{'}a . (\text{hd}, \text{hd}) \in ?\)
Parametricity Examples

• First element of pair: \texttt{fst}::'a × 'b ⇒ 'a  
\((\texttt{fst}, \texttt{fst}) \in A × B → A\)

• Append two lists: \texttt{(}@::'a list ⇒ 'a list ⇒ 'a list  
\(((\@), (\@)) \in \langle A\rangle list\_rel → \langle A\rangle list\_rel → \langle A\rangle list\_rel\)

• Limitations in HOL
  • Partiality/underdefinedness: \texttt{hd}::'a list ⇒ 'a . (\texttt{hd},\texttt{hd}) \in ?  
  • equality/type-classes: \texttt{List.member}::'a list ⇒ 'a ⇒ bool .  
\((\texttt{List.member},\texttt{List.member}) \in ?\)
Parametricity Examples

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

- Append two lists: \( @(::\text{'}a \\text{list} \Rightarrow \text{'}a \\text{list} \Rightarrow \text{'}a \\text{list} \)
  \(((@), (@)) \in \langle A \rangle \text{list}\_\text{rel} \rightarrow \langle A \rangle \text{list}\_\text{rel} \rightarrow \langle A \rangle \text{list}\_\text{rel} \)

- Limitations in HOL
  - Partiality/underdefinedness: \( \text{hd}::\text{'}a \\text{list} \Rightarrow \text{'}a . (\text{hd}, \text{hd}) \in ? \)
  - equality/type-classes: \( \text{List}\_\text{member}::\text{'}a \\text{list} \Rightarrow \text{'}a \Rightarrow \text{bool} . (\text{List}\_\text{member}, \text{List}\_\text{member}) \in ? \)

- Solution: Preconditions, generalization
Parametricity Examples

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

- Append two lists: 
  \( (@)::'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \)
  
  \(((@), (@)) \in \langle A \rangle \text{list_rel} \rightarrow \langle A \rangle \text{list_rel} \rightarrow \langle A \rangle \text{list_rel}\)

- Limitations in HOL
  - Partiality/underdefinedness: 
    \( \text{hd}::'a \text{ list} \Rightarrow 'a . \ (\text{hd}, \text{hd}) \in ?\)
  - equality/type-classes: 
    \( \text{List.member}::'a \text{ list} \Rightarrow 'a \Rightarrow \text{bool} . \)
    \((\text{List.member}, \text{List.member}) \in ?\)

- Solution: Preconditions, generalization
  - \( [l \neq []]; (l', l) \in \langle A \rangle \text{list_rel} \Rightarrow (\text{hd} \ l', \text{hd} \ l) \in A\)
Parametricity Examples

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

• Append two lists: \( (\@) \colon 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \)
  \(((@), (@)) \in \langle A \rangle \text{list} \_\text{rel} \rightarrow \langle A \rangle \text{list} \_\text{rel} \rightarrow \langle A \rangle \text{list} \_\text{rel} \)

• Limitations in HOL
  • Partiality/underdefinedness: \( \text{hd} \colon 'a \text{ list} \Rightarrow 'a . (\text{hd}, \text{hd}) \in ? \)
  • equality/type-classes: \( \text{List} \_\text{member} \colon 'a \text{ list} \Rightarrow 'a \Rightarrow \text{bool} . (\text{List} \_\text{member}, \text{List} \_\text{member}) \in ? \)

• Solution: Preconditions, generalization
  • \([l \neq []; (l', l) \in \langle A \rangle \text{list} \_\text{rel}] \implies (\text{hd} \_l', \text{hd} \_l) \in A \)
  • \( \text{glist} \_\text{member} \colon ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool} . (\text{glist} \_\text{member}, \text{glist} \_\text{member}) \in (\langle R_a \rangle \text{list} \_\text{rel} \rightarrow \langle \text{bool} \_\text{rel} \rangle) \rightarrow \langle R_a \rangle \text{list} \_\text{rel} \rightarrow \langle \text{bool} \_\text{rel} \rangle \)
Summary

- Apply refinement rules for combiners
  - 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

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
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

\[
\text{datatype} \ (\texttt{'}a,\texttt{'}h) \ M = M (\text{run}: \texttt{'}h \Rightarrow (\texttt{'}a \times \texttt{'}h))
\]

\[
\text{return } x = M (\lambda s. (x,s))
\]

\[
\text{bind } m_1 \ m_2 = M (\lambda s. \text{let } (x,s') = \text{run } m_1 \ s \text{ in run } (m_2 x) \ s')
\]

\[
\text{get } = M (\lambda s. (s,s))
\]

\[
\text{put } s = M (\lambda_. ((),s))
\]
Model imperative program by state monad

```
datatype ('a,'h) M = M (run: 'h ⇒ ('a×'h))
```

```
return x = M (λs. (x,s))
bind m₁ m₂ = M (λs. let (x,s') = run m₁ s in run (m₂ x) s')
get = M (λs. (s,s))
put s = M (λ_. (()),s))
```

Program takes state and returns result and new state
Imperative/HOL

- Model imperative program by state monad

  \begin{align*}
  \text{datatype} \quad & (\forall a,h. M = M (run: (h) \Rightarrow (a \times h))) \\
  \text{return} \; x & = M (\lambda s. (x,s)) \\
  \text{bind} \; m_1 \; m_2 & = M (\lambda s. \text{let} \; (x,s') = \text{run} \; m_1 \; s \rightarrow \text{run} \; m_2 \; x \; s') \\
  \text{get} & = M (\lambda s. (s,s)) \\
  \text{put} \; s & = M (\lambda s. (((),s)))
  \end{align*}

- 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

  \[
  \text{datatype} \quad (\texttt{a}, \texttt{h}) \ M = M (\text{run}: \langle \texttt{h} \Rightarrow (\texttt{a} \times \texttt{h}) \rangle)
  \]

  \[
  \text{return} \ x = M (\lambda \ s. (x, s))
  \]

  \[
  \text{bind} \ m_1 \ m_2 = M (\lambda \ s. \text{let} \ (x, s') = \text{run} \ m_1 \ s \text{ in run} \ (m_2 \ x) \ s')
  \]

  \[
  \text{get} = M (\lambda \ s. (s, s))
  \]

  \[
  \text{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

  \[
  \text{datatype} \ (\ 'a, 'h) M = M \ (\text{run}: \langle 'h \Rightarrow ('a \times 'h) \rangle)
  \]

  \[
  \text{return} \ x = M \ (\lambda s. (x, s))
  \]

  \[
  \text{bind} \ m_1 \ m_2 = M \ (\lambda s. \text{let} \ ((x, s')) = \text{run} \ m_1 \ s \text{ in run} \ (m_2 \ x) \ s')
  \]

  \[
  \text{get} = M \ (\lambda s. (s, s))
  \]

  \[
  \text{put} \ s = M \ (\lambda s. ((()), s))
  \]

- 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

- Model imperative program by state monad

\[
\text{datatype} \ (a, h) \ M = M (\text{run}: \langle h \Rightarrow (a \times h) \rangle)
\]

\[
\text{return} \ x = M (\lambda s. (x, s))
\]

\[
\text{bind} \ m_1 \ m_2 = M (\lambda s. \text{let} \ (x, s') = \text{run} \ m_1 \ s \ \text{in} \ \text{run} \ (m_2 \ x) \ s')
\]

\[
\text{get} = M (\lambda s. (s, s))
\]

\[
\text{put} \ s = M (\lambda \_. ((()), s))
\]

- 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
  ```
  datatype ('a,'h) M = M (run: ('h ⇒ ('a×'h)))
  
  return x = M (λs. (x,s))
  bind m₁ m₂ = M (λs. let (x,s') = run m₁ s in run (m₂ x) s')
  get = M (λs. (s,s))
  put s = M (λ_. ((),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
Refinement

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

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

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

$hn\_refine \Gamma \ c \ \Gamma' \ R \ a$

- $\Gamma$ Heap content before execution
- $\Gamma'$ Heap content after execution
- $R$ Relation for result
Imperative Refinement Basics

- Establish relation between imperative-program $c$ and nres-program $a$
  
  $hn\_refine \Gamma \ c \ \Gamma' \ R \ a$

  - $\Gamma$ Heap content before execution
  - $\Gamma'$ Heap content after execution
  - $R$ Relation for result

- Formally
  
  $hn\_refine \Gamma \ c \ \Gamma' \ R \ m \equiv$

  $nofail \ m \rightarrow <\Gamma> \ c \ <\lambda r. \ \Gamma' \ast (\exists A x. \ R \ x \ r \ast \uparrow (\text{return} \ x \leq \ m))>_t$

  - where $<P> \ c \ <\lambda r. \ Q \ r>_t$ is Hoare-triple for Imperative/HOL programs
• Refinement relations now also cover heap content.
• Refinement relations now also cover heap content.

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

• Examples:
  
  • `array_assn int_assn::int list ⇒ int Heap.array ⇒ assn` — Implements list of integers by array of integers
Separation Logic

• Refinement relations now also cover heap content.

• Examples:
  
  • \texttt{array\_assn int\_assn::int list \Rightarrow int Heap.array \Rightarrow assn} — Implements list of integers by array of integers
  
  • \texttt{amtx\_assn M N int\_assn::(nat \times nat \Rightarrow int) \Rightarrow int Heap.array \Rightarrow assn} — Function \texttt{nat \times nat \Rightarrow int} by \texttt{M\times N} array
Separation Logic

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

- Refinement relations now also cover heap content.
- Examples:
  - `array_assn int_assn::int list ⇒ int Heap.array ⇒ assn` — Implements list of integers by array of integers
  - `amtx_assn M N int_assn:(nat × nat ⇒ int) ⇒ int Heap.array ⇒ assn` — Function `nat × nat ⇒ int` by `M×N` array
  - `int_assn = (λ i i'. ↑ (i' = i))` — Pure assertion, no heap content.
- Assertions separated by `∗`: They do not alias!
Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - `array_assn int_assn::int list ⇒ int Heap.array ⇒ assn` — Implements list of integers by array of integers
  - `amtx_assn M N int_assn::(nat × nat ⇒ int) ⇒ int Heap.array ⇒ assn` — Function `nat × nat ⇒ int` by `M×N` array
  - `int_assn = (λi i'. ↑(i' = i))` — Pure assertion, no heap content.
- Assertions separated by `*` : They do not alias!
  - `array_assn int_assn l_1 a_1 * array_assn int_assn l_2 a_2`
Separation Logic

- Refinement relations now also cover heap content.
- Examples:
  - `array_assn int_assn::int list ⇒ int Heap.array ⇒ assn` — Implements list of integers by array of integers
  - `amtx_assn M N int_assn::(nat × nat ⇒ int) ⇒ int Heap.array ⇒ assn` — Function `nat × nat ⇒ int` by `M×N` array
  - `int_assn = (λi i’. ↑ (i’ = i))` — Pure assertion, no heap content.
- Assertions separated by `*` : They do not alias!
  - `array_assn int_assn l_1 a_1 * array_assn int_assn l_2 a_2`
  - Arrays `a_1` and `a_2` do not overlap
Frame Rule

\[ <P> \ c <Q> \implies <P \ast F> \ c <\lambda r. Q \ r \ast F>_t \]
Frame Rule

\[
\langle P \rangle \ c \ \langle Q \rangle \quad \Longrightarrow \quad \langle P \ast F \rangle \ c \ \langle \lambda r. \ Q \ r \ast F \rangle_t
\]

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

\[
<P> ~ c ~ <Q> \implies <P * F> ~ c ~ <\lambda r. Q ~ r * F>_t
\]

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

Rule for lookup in hashtable:

\[
<\text{is\_hashmap} ~ m ~ ht> ~ hm\_lookup ~ k ~ ht ~ <\lambda r. \text{is\_hashmap} ~ m ~ ht ~ r ~ \uparrow (r = m ~ k)> 
\]
\[
< P \> \ c \ < Q > \implies < P \ast F \> \ c \ < \lambda r. \ Q \ r \ast F >_t
\]

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

Rule for lookup in hashtable:

\[
< \text{is\_hashmap} \ m \ ht > \ \text{hm\_lookup} \ k \ ht \ < \lambda r. \ \text{is\_hashmap} \ m \ ht \ast \uparrow (r = m \ k)> \\
\]

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

\[
< \text{is\_hashmap} \ m_1 \ ht_1 \ast \\
\text{is\_array\_list} \ l_2 \\
\text{al}_2> \ \text{hm\_lookup} \ k \\
ht_1 < \lambda r. \ \text{is\_hashmap} \ m_1 \ ht_1 \ast \text{is\_array\_list} \ l_2 \ \text{al}_2 \ast \uparrow (r = m_1 \ k)>_t
\]
Operation Refinement

<is_hashmap m ht> \text{hm\_lookup} k \text{ht} <\lambda r. \text{is\_hashmap} m \text{ht} \ast \uparrow (r = m k)>
Operation Refinement

<is_hashmap m ht> hm_lookup k ht <λr. is_hashmap m ht * ↑ (r = m k)>

The hashtable still exists after operation
Operation Refinement

\(<\text{is\_hashmap } m \ ht> \ \text{hm\_lookup } k \ ht <\lambda r. \ \text{is\_hashmap } m \ ht \ast \uparrow (r = m k)>\)

The hashtable still exists after operation

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

\(<\text{is}\_\text{hashtable} \ m \ \text{ht}> \ \text{hm\_lookup} \ k \ \text{ht} <\lambda r. \ \text{is}\_\text{hashtable} \ m \ \text{ht} \ \ast \ \uparrow (r = m \ k)>\)

The hashtable still exists after operation

\(<\text{is}\_\text{hashtable} \ m \ \text{ht}> \ \text{hm\_update} \ k \ v \ \text{ht} <\lambda r. \ \text{is}\_\text{hashtable} \ (m(k \mapsto v)) \ r>_t\)

Original hashtable is gone (destructive update)
Operation Refinement

\[
<\text{is\_hashtable } m \text{ HT}> \text{ hm\_lookup } k \text{ HT} <\lambda r. \text{ is\_hashtable } m \text{ HT} \uparrow (r = m \ k)>
\]

The hashtable still exists after operation

\[
<\text{is\_hashtable } m \text{ HT}> \text{ hm\_update } k \ v \text{ HT} <\lambda r. \text{ is\_hashtable } (m(k \mapsto v)) \ r>\]

Original hashtable is gone (destructive update)

Shortcut notations

\[
(hm\_lookup, \lambda k \ m. \ m \ k) \in id^k \ast is\_hashtable^k \rightarrow id
\]

\[
(hm\_update, \lambda k \ v \ m. \ m(k \mapsto v)) \in id^k \ast id^k \ast is\_hashtable^d \rightarrow is\_hashtable
\]
Operation Refinement

\[ <\text{is\_hashmap \( m \) \( h_t \)}> \text{hm\_lookup \( k \) \( h_t \)} \langle \lambda r. \text{is\_hashmap \( m \) \( h_t \) \uparrow (r = m \( k \))} \rangle \]

The hashtable still exists after operation

\[ <\text{is\_hashmap \( m \) \( h_t \)}> \text{hm\_update \( k \) \( v \) \( h_t \)} \langle \lambda r. \text{is\_hashmap} (m(k \mapsto v)) r \rangle_t \]

Original hashtable is gone (destructive update)

Shortcut notations

\[
(hm\_lookup, \lambda k m. m k) \in id^k \ast \text{is\_hashmap}^k \rightarrow id \\
(hm\_update, \lambda k v m. m(k \mapsto v)) \in id^k \ast id^k \ast \text{is\_hashmap}^d \rightarrow \text{is\_hashmap}
\]

k — keep

d — destroy
Synthesis of Program

Identify operations

\[
\begin{align*}
\text{do } & \{ \\
& \quad \text{assert } (Q \neq \{\}) \\
& \quad v \leftarrow \text{spec } x. \ x \in Q \land (\forall y \in Q. \ x \leq y) \\
& \quad \text{case } m \ v \ of \\
& \quad \quad \text{None } \Rightarrow \text{return } (m(v \mapsto \text{True})) \\
& \quad \quad \text{Some } - \Rightarrow \text{return } m \\
\} 
\end{align*}
\]
Identify operations

\[
\text{do } \{
\quad \text{assert } (Q\neq \{\}); \\
\quad v \leftarrow \text{pq_get_min } Q; \\
\quad \text{case map_lookup } v \ m \ \text{of} \\
\quad \quad \text{None } \Rightarrow \text{return } (\text{map_update } v \ True \ m) \\
\quad \quad \text{Some } _\_ \Rightarrow \text{return } m
\}
\]
do {
  assert (Q≠{});
  v ← pq_get_min Q;
  case map_lookup v m of
    None ⇒ return (map_update v True m)
  | Some _ ⇒ return m
}
Monadify (flatten expressions)

```haskell
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
}
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
    }

Initialize heap content
Synthesis of Program

Initialize heap content

```plaintext
do {
  < minheap Qi Q * hashmap mi m >
  assert (Q\neq\{\});
  v ← pq\_get\_min Q;
  t_1 ← return (map\_lookup v m)
  case t_1 of
    None ⇒ do {t_2 ← return True; return (map\_update v t_2 m)}
    | Some _ ⇒ return m
}```
Synthesis of Program

Symbolic forward execution

do {
    < minheap Qi Q * hashmap mi m >
    assert (Q≠{});
    v ← pq_get_min Q;
    t_1 ← return (map_lookup v m)
    case t_1 of
        None ⇒ do {t_2 ← return True; return (map_update v t_2 m)}
        Some _ ⇒ return m
}
Synthesis of Program

Assert becomes no-op

do {
    (* assert (Q≠{}); *)
    < minheap Qi Q * hashmap mi m > | 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

\[(minheap\_get\_min, \ pq\_get\_min) \in minheap^k \rightarrow int\]

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

Result now also bound.

do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  < minheap Q Qi * hashmap m mi * int v vi > | 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

\((hm\_lookup, map\_lookup) \in int^k \times hashmap^k \rightarrow (bool)option\)

do {
  (* assert \((Q\neq\{\})\); *)
  \(vi \leftarrow \text{minheap}\_\text{get\_min } Qi;\)
  \(<\text{minheap } Q Qi \times hashmap m mi \times int v vi > | Q\neq\[]\)
  \(t_1 \leftarrow \text{return } (map\_lookup v m)\)
  case \(t_1\) of
    None \(\Rightarrow\) do \(\{\) \(t_2 \leftarrow \text{return } True; \) \text{return } (map\_update v t_2 m)\}\)
    | Some \_ \(\Rightarrow\) return \(m\) \}
do {
  (* assert (Q≠{}); *)
  vi ← minheap_get_min Qi;
  ti_1 ← hm_lookup vi mi;
  < minheap Q Qi * hashmap m mi * int v vi * (bool)option t_1 ti_1 > | Q≠[]
  case t_1 of
    None ⇒ do {t_2 ← return True; return (map_update v t_2 m)}
    Some _ ⇒ return m
}
Synthesis of Program

Split: Translate both branches separately

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

\[(\text{return } True, \text{return } True) \in - \rightarrow \text{bool}\]

do {
(* assert (Q\neq\{\}); *)

vi ← minheap\_get\_min Qi;

ti_1 ← hm\_lookup vi mi;

case ti_1 of

None ⇒ do {

< minheap Q Qi * hashmap m mi * int v vi > | Q\neq\[], t_1=\text{None}

t_2 ← \text{return } True;

return (map\_update v t_2 m)
}

| Some _ ⇒

< minheap Q Qi * hashmap m mi * int v vi * bool t_3 ti_3 > | Q\neq\[],
t_1=\text{Some } t_3

return m

}
do {
    (\* assert (Q\(\not=\)\{\})); *)
    vi ← minheap_get_min Q_i;
    ti_1 ← hm_lookup vi mi;
    case \(t_i_1\) of
        None ⇒ do {
            ti_2 ← return True;
            < minheap Q Q_i \(\not=\) [] \_ \_ bool ti_2 t_2 > \| Q\(\not=\)\[], t_1=\text{None}
            return (map_update v t_2 m)
        }
        Some _ ⇒
            < minheap Q Q_i \(\not=\) [] \_ \_ bool t_3 ti_3 > \| Q\(\not=\)\[],
            t_1=Some t_3
            return m
    }
Synthesis of Program

\((hm\_update, map\_update) \in int^k \times bool^k \times hashmap^d \rightarrow hashmap\)

do {
(* assert \((Q \neq \emptyset)\); *)
\(vi \leftarrow minheap\_get\_min Qi;\)
\(ti_1 \leftarrow hm\_lookup vi mi;\)
case \(ti_1\) of
  None \Rightarrow do {
    \(ti_2 \leftarrow return True;\)
    \(<\ minheap Q Qi \times hashmap m mi \times int v vi \times bool ti_2 t_2 > | Q \neq [], t_1=\text{None}\)
    return (map\_update v t_2 m)
  }
  Some \_ \Rightarrow
    \(<\ minheap Q Qi \times hashmap m mi \times int v vi \times bool t_3 ti_3 > | Q \neq [], t_1=\text{Some} t_3\)
    return m
}
Synthesis of Program

Destructive update, $m_i$ no longer valid

\[
\text{do } \{ \\
\text{\hspace{2em}}(* \text{ assert } (Q \neq \{\}) ; *) \\
\text{\hspace{2em}}vi \leftarrow \text{minheap\_get\_min } Qi; \\
\text{\hspace{2em}}ti_1 \leftarrow \text{hm\_lookup } vi \ mi; \\
\text{\hspace{2em}}\text{case } ti_1 \text{ of} \\
\text{\hspace{3em}}\text{None } \Rightarrow \text{ do } \{ \\
\text{\hspace{4em}}ti_2 \leftarrow \text{return } \text{True}; \\
\text{\hspace{4em}}\text{hm\_update } vi \ ti_2 \ mi \\
\text{\hspace{4em}}< \text{minheap } Q \ Qi \ast \text{int } v \ vi \ast \text{bool } ti_2 \ t_2 \ast \text{hashmap } @r \ @ri > \ | \ Q \neq [], \\
\text{\hspace{4em}}t_1 = \text{None} \\
\text{\hspace{3em}}\} \\
\text{\hspace{3em}}| \ \text{Some } _ \Rightarrow \\
\text{\hspace{4em}}< \text{minheap } Q \ Qi \ast \text{hashmap } m \ mi \ast \text{int } v \ vi \ast \text{bool } t_3 \ ti_3 \ > \ | \ Q \neq [], \\
\text{\hspace{4em}}t_1 = \text{Some } t_3 \\
\text{\hspace{3em}}\text{return } m \\
\text{\hspace{2em}}\} 
\]
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

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

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

Merge.

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 _ ⇒
      return mi
      < minheap Q Qi * int v vi * bool t₃ ti₃ * hashmap @r @ri > | Q≠[],
      t₁=Some t₃
  }
}
Synthesis of Program

Merge. \( t_i_2 \) goes out of scope.

\[
\text{do } \{
\begin{align*}
&\quad \text{(* assert \( Q \neq \{\} \); *)} \\
&\quad vi \leftarrow \text{minheap\_get\_min} \ Qi; \\
&\quad ti_1 \leftarrow \text{hm\_lookup} \ vi \ mi; \\
&\quad \text{case } ti_1 \text{ of} \\
&\quad \qquad \text{None } \Rightarrow \text{do } \{
\begin{align*}
&\quad \quad ti_2 \leftarrow \text{return} \ True; \\
&\quad \quad \text{hm\_update} \ vi \ ti_2 \ mi
\end{align*}
\}
\quad \mid \text{Some } \_ \Rightarrow \\
&\quad \quad \text{return} \ mi
\end{align*}
\}
\]

\[
< \text{minheap} \ Q \ Qi \ast \text{int} \ v \ vi \ast \text{hashmap} \ @r \ @ri \ast \ast \text{(bool)} \text{option} \ t_1 \ ti_1 > | \ Q \neq []
\]
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