# The Isabelle Refinement Framework

Peter Lammich

March 18, 2021

# Outline


1 Introduction

2 Details

# High-Level Ideas

- Stepwise refinement
  - separates algorithmic ideas from impl details
  - modularizes programs and proofs
- Powerful (interactive) theorem prover
  - for ambitious background theory
  - trustworthy small kernel
  - Isabelle/HOL: mature prover+IDE, libraries+AFP, sledgehammer
- Automation
  - tools (e.g. VCG, automatic refinement)
  - do not extend TCB

# Example

**procedure** $\textsc{Kruskal}(E)$
  $\equiv \leftarrow =$ on nodes of $E$
  $F \leftarrow \emptyset$
  **while** $E \neq \emptyset$ **do**
    remove $(u, w, v)$ with minimal $w$ from $E$
    **if** $\neg u \equiv v$ **then**
      $F \leftarrow F \cup \{(u, w, v)\}$
      $\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\mathsf{stcl}}$

- First:
  - show this *textbook-level* algorithm correct
  - requires some theory on matroids + VCG
- Independently:
  - show that graphs can be implemented by weight-sorted edge lists
  - prove union-find data structure correct
- Finally:
  - assemble all proofs to get correct+efficient implementation

# Outline


1 Introduction

2 Details

# Nondeterminism/Error Monad

- Shallowly embedded in Isabelle/HOL

  $\alpha$ nres = `fail` | `spec` $\alpha$ `set`
  complete lattice: $\_ \leq$ `fail` | `spec` x $\leq$ `spec` y $\longleftrightarrow$ x $\subseteq$ y

- $a \leq b$ - $a$ refines $b$
  - $a$ has less possible results than $b$
  - if $b$ fails, $a$ can be anything
- Hoare-Triple: $P \implies c \leq (\lambda r.\ Q\ r)$
  $P\ args \implies c\ args \leq (\lambda r.\ Q\ args\ r)$

# Embedded Programming Language

- Monad: bind, return
- Flat ordering with fail: rec
- From HOL: if, let, pair, ...
- Derived: while, foreach, assert, ...
- Note: no (implicit) state at this level!
    - but state can be 'threaded through' explicitly

# VCG

- Hoare-Like rules enable syntax-based VCG

  $\mathsf{Q}\ \mathsf{x} \implies \texttt{return}\ \mathsf{x} \leq \mathsf{SPEC}\ \mathsf{Q}$
  $\mathsf{m} \leq \mathsf{SPEC}\ (\lambda \mathsf{x}.\ \mathsf{f}\ \mathsf{x} \leq \mathsf{SPEC}\ \mathsf{Q}) \implies \texttt{bind}\ \mathsf{m}\ \mathsf{f} \leq \mathsf{SPEC}\ \mathsf{Q}$
  $\ldots$

# Data Refinement

- Relation between concrete and abstract values, e.g.

  $R_{set}^{list}$ xs s $\equiv$ s = set xs $\wedge$ distinct xs

  - common form: R c a $\equiv$ a=$\alpha$ c $\wedge$ I c

- Lift to nres

  $\Downarrow$ R fail $\equiv$ fail
  $\Downarrow$ R (spec S) $\equiv$ spec c. $\exists$a$\in$S. R c a

# Example

$\mathrm{min\_set}\ s = \mathtt{assert}\ (s \neq \{\});\ \mathtt{spec}\ (x, s - \{x\}).\ x \in s$
$\mathrm{min\_list}\ (x \# xs) = \mathtt{return}\ (x, xs)$

$R_{set}^{list}\ xs\ s$
$\Longrightarrow\ \mathrm{min\_list}\ xs \leq \Downarrow(I \times R_{set}^{list})\ \mathrm{min\_set}\ xs$

shorter:

$\mathrm{min\_list},\ \mathrm{min\_set} : R_{set}^{list} \rightarrow I \times R_{set}^{list}$

# Monotonicity

- Combinators are monotonous, also wrt. data refinement

  $\mathsf{R\ x\ x'} \implies \texttt{return}\ \mathsf{x} \le\Downarrow\mathsf{R}\ \texttt{return}\ \mathsf{x'}$

  $\mathsf{m} \le \Downarrow\mathsf{R\ m'};\ \bigwedge \mathsf{x\ x'}.\ \mathsf{R\ x\ x'} \implies \mathsf{f\ x} \le\Downarrow\mathsf{S\ f'\ x'}$
  $\implies \texttt{bind}\ \mathsf{m\ f} \le\Downarrow\mathsf{S}\ \texttt{bind}\ \mathsf{m'\ f'}$

  ...

- Enables syntax-based VCG for refinement
  - some heuristics required, e.g., to introduce $R$ in bind-rule
- Common refinements:
  - specification refinement: $\mathsf{c} \le \texttt{spec}\ \mathsf{Q}$
  - structural refinement: $\mathsf{combinator\ c} \le\Downarrow \mathsf{R\ combinator\ c'}$
  - operator refinement: $\mathsf{op\ x} \le\Downarrow \mathsf{R\ op\ x'}$
  - solved by (combined) VCG

# Synthesis

- Given abstract program and concrete data-structures
  - synthesize concrete program
  - using implementations for `op`s and `spec`s from data-structures

# Example

```
procedure KRUSKAL(E)
    (≡) ← = on nodes of E
    F ← ∅
    while E ≠ ∅ do
        remove (u, w, v) with minimal w from E
        if ¬u ≡ v then
            F ← F ∪ {(u, w, v)}
            ≡ ← (≡ ∪ {(u, v)})^stcl
```

# Example

**procedure** $\textsc{Kruskal}(E)$
  $R_{graph}^{list}\ xs\ E$
  $(\equiv) \leftarrow =$ on nodes of $E$
  $F \leftarrow \emptyset$
  **while** $E \neq \emptyset$ **do**
    remove $(u, w, v)$ with minimal $w$ from $E$
    **if** $\neg u \equiv v$ **then**
      $F \leftarrow F \cup \{(u, w, v)\}$
      $\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\mathsf{stcl}}$

# Example

**procedure** $\textsc{Kruskal}(E)$

$uf \leftarrow$ init-uf $xs$

$R^{list}_{graph}\ xs\ E \quad R^{uf}_{per}\ uf\ (\equiv)$

$F \leftarrow \emptyset$

**while** $E \neq \emptyset$ **do**

remove $(u, w, v)$ with minimal $w$ from $E$

**if** $\neg u \equiv v$ **then**

$F \leftarrow F \cup \{(u, w, v)\}$

$\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\text{stcl}}$

# Example

**procedure** $\textsc{Kruskal}(E)$

  $uf \leftarrow$ init-uf $xs$

  $ys \leftarrow []$

  $R^{list}_{graph}\ xs\ E \quad R^{uf}_{per}\ uf\ (\equiv) \quad R^{list}_{set}\ ys\ F$

  **while** $E \neq \emptyset$ **do**

    remove $(u, w, v)$ with minimal $w$ from $E$

    **if** $\neg u \equiv v$ **then**

      $F \leftarrow F \cup \{(u, w, v)\}$

      $\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\mathsf{stcl}}$

# Example

**procedure** $\textsc{Kruskal}(E)$
  $uf \leftarrow$ init-uf $xs$
  $ys \leftarrow []$
  **while** $xs \neq []$ **do**
    $R^{list}_{graph}\ xs\ E \quad R^{uf}_{per}\ uf\ (\equiv) \quad R^{list}_{set}\ ys\ F$
    remove $(u, w, v)$ with minimal $w$ from $E$
    **if** $\neg u \equiv v$ **then**
      $F \leftarrow F \cup \{(u, w, v)\}$
      $\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\mathsf{stcl}}$

# Example

**procedure** $\textsc{Kruskal}(E)$

$uf \leftarrow \text{init-uf } xs$

$ys \leftarrow []$

**while** $xs \neq []$ **do**

$((u, w, v), xs) \leftarrow (hd\ xs, tl\ xs)$

$R_{graph}^{list}\ xs\ E \quad R_{per}^{uf}\ uf\ (\equiv) \quad R_{set}^{list}\ ys\ F$

**if** $\neg u \equiv v$ **then**

$F \leftarrow F \cup \{(u, w, v)\}$

$\equiv \leftarrow (\equiv \cup \{(u, v)\})^{\text{stcl}}$

# Backends

- Purely functional
  - refine until no SPECs left
  - then transfer to option monad, and use code generator
- Imperative
  - synthesis steps use separation logic
  - and target a heap monad
- Targets
  - Imperative/HOL: heap monad for Isabelle's code generator
  - Isabelle LLVM: shallow embedding of LLVM into Isabelle/HOL

# Conclusions

- These techniques allow efficient implementations of complex algorithms
  - SAT: sat-solver, drat-checker, ...
  - Graph: Edmonds Karp, push-relabel, Dijkstra, Kruskal, Prim, Floyd-Warshall, ...
  - Automata: LTL-modelchecker, Timed-Automata model checker, ...
  - Sorting: Introsort, Pdqsort, ...
- Recently: Isabelle-LLVM with Time
  - Verify correctness+asymptotic complexity
- But no concurrency yet