RS3 Code Generation Tutorial with Isabelle/HOL

The Refinement Framework

Peter Lammich

May 2015

About this Tutorial

- These are the slides of an tutorial on the Refinement Framework and Autoref, that I gave in May 2015
- You can download the accompanying theories and a snapshot of the refinement framework at
  `http://www21.in.tum.de/~lammich/refine_tutorial.html`
- To get started, run isabelle.sh to start the IDE. On first invocation, it will build the image, which will take some time.
  - It requires Isabelle 2014 to be on your path!

Table of Contents

Motivation

- Algorithmic ideas presented best on abstract level
- Can also be proved on abstract level
- Implementation is orthogonal issue

Direct Proofs

- Our experience shows
  - Direct proofs tend to get unmanageable
  - E.g. Dijkstra's algorithm

Separation of algorithmic idea and implementation

- Increased modularity
- Proofs are independent
- Changing implementation is simple

Reusable data structures

- Implementations often make use of standard data structures
  - Hash tables, red-black-trees, heaps, ...
- Abstractly, these correspond to standard HOL types
  - Set, map, ...
- Important to have library of reusable implementations
  - Here: Isabelle Collection Framework

Simple example: Set-Sum

- Specification: Σs
- Abstract algorithm

  r=0
  for $x \in s$ **do** $r = r + x$
  return *r*

- Concrete (lists)

  List.fold op + l 0

- Concrete (RBT)

  RBT.fold $(\lambda k \_\ r.\ r+k)$ *t 0*

Proving Set-sum correct

- Abstract idea: Invariant
  - *r is sum of elements already iterated over*
- For concrete algorithms, the proof depends on available lemmas
- Has to be repeated for each set implementation
  - Tedious, if proof gets complicated!

Demo

Setsum.thy

Table of Contents

Table of Contents

Nondeterminism

- Abstract specification may be nondeterministic
  - *Find an element with minimal priority*
  - *Compute a path from node u to v*
- Implementation is deterministic
  - But depends on details of used data structures

Monads

- If you do not know monads, skip to slide 18
- Structure $\alpha\ M$ with functions
  - $return\ :\ \alpha \to \alpha\ M$, $bind : \alpha\ M \to (\alpha \to \beta\ M) \to \beta\ M$
  - $bind\ m\ f$ also written $m \gg\!= f$
- That satisfy the monad laws

$$return\ x \gg\!= f = f \tag{id1}$$

$$m \gg\!= return = m \tag{id2}$$

$$(m \gg\!= f_1) \gg\!= f_2 = m \gg\!= (\lambda x.f_1\ x \gg\!= f_2) \tag{assoc}$$

- But now some more intuitive thing

Nondeterminism Monad

- α *set*
  - *return* $x = \{x\}$, $m \gg\!= f = \bigcup\{fx \mid x \in m\}$
  - *return* x yields the only value x
  - $m \gg\!= f$ nondeterministically chooses value of m, and applies f to it
    - Sequential composition

Adding errors

- Will turn out later: Nice to have errors
- α **nres** $=$ **res** α *set* $\mid$ **fail**
  - **return** $x =$ **res** $\{x\}$, **res** $X \gg\!= f = \bigsqcup\{fx \mid x \in X\}$, **fail** $\gg\!= f =$ **fail**
  - Intuition: Error propagates over bind.
  - Note: Complete lattice structure of α *set* lifted to α **nres**
    - **res** $X \sqsubseteq$ **res** $Y \iff X \subseteq Y$ and $\_ \sqsubseteq$ **fail**
  - Intuition: Possibility to choose error $\Longrightarrow$ error
- Assertions: **assert** $\Phi =$ **if** Φ **then return** $()$ **else fail**
  - Fail if Φ does not hold
- Specification: **spec** $x.\ \Phi\ x =$ **res** $\{x \mid \Phi\ x\}$
  - All values that satisfy Φ

Do-notation

- More readable notation for monadic programs
- **let** $x = t$; f syntax for **let**$x = t$ **in** f
- $x \leftarrow m$; f syntax for $m \gg\!= (\lambda x.f)$
- This shortcuts are enclosed in **do**$\{\ldots\}$ - block

Theory-free intuition

- $x \leftarrow m$ — Execute m, assign result to x
- **return** x — Return result x
- **assert**Φ — Assert that Φ holds. Immediate termination with failure otherwise.
- $x \leftarrow$ **spec** $x.\ \Phi\ x$ — Assign some x that satisfies Φ (choose nondeterministically).
  - Note: If there are no such x, your program will have no possible results at all.

Examples

$$
\begin{array}{l}
\textbf{do}\ \{ \\
\quad \mathit{ASSERT}\ (l \neq []); \\
\quad \mathit{RETURN}\ (\mathit{hd}\ l) \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{do}\ \{ \\
\quad \mathit{ASSERT}\ (s \neq \{\}); \\
\quad \mathit{SPEC}\ (\lambda x.\ x \in s) \\
\}
\end{array}
$$

Table of Contents

Refinement Ordering

- Recall: Subset-ordering lifted to α **nres**
  - We will use $\leq$-symbol from now on

$$\begin{array}{l} \mathit{RES}\ X \leq \mathit{RES}\ Y \longleftrightarrow X \subseteq Y \\ \_ \quad \leq \mathit{FAIL} \longleftrightarrow \mathit{True} \\ \mathit{FAIL} \leq \mathit{RES}\ \_ \longleftrightarrow \mathit{False} \end{array}$$

- Intuition: $m \leq m'$
  - All results of m also possible in m' (or m' is error)
  - *m refines m′*
- Interesting cases
  - $m \leq$ **spec** Φ Possible results of m satisfy Φ
  - $m \leq$ **fail** Error refined by everything
  - **res** $\emptyset \leq m$ Empty result refines everything
    - We define **succeed** $=$ **res** $\emptyset$
    - Sometimes also called *magic*, as it magically satisfies any specification

Examples

$$sort\_spec\ l = SPEC\ (\lambda l'.\ multiset\_of\ l = multiset\_of\ l' \wedge sorted\ l')$$

$$sort\ l \leq sort\_spec\ l$$

$$distinct\ l \Longrightarrow sort\ l \leq sort\_spec\ l$$

$$pre\ a \Longrightarrow algo\ a \leq SPEC\ (\lambda r.\ post\ a\ r)$$
$$(* \ Compare:\ Hoare-triple\ \{pre\}\ algo\ \{post\}\ *)$$

Refinement

- Refinement ordering is transitive (it's a complete lattice)
  - In particular: *impl* $\leq$ *abs* and *abs* $\leq$ **spec** Φ implies *impl* $\leq$ **spec** Φ
  - Allows to split abstract correctness proof and implementation
- Bind (and other combinators) are monotone
  - $m' \leq m, f' \leq f \Longrightarrow m' \gg\!= f' \leq m \gg\!= f$
  - Only refining parts of program implies refinement

Examples

$$\mathit{min\_spec}\ l = \mathbf{do}\ \{\mathit{ASSERT}\ (l \neq []);\ \mathit{SPEC}\ x.\ x \in \mathit{set}\ l \wedge \mathit{prio}\ x = \mathit{Min}\ (\mathit{prio}\text{`}\mathit{set}\ l)\}$$

$$\mathit{min\_abs}\ l = \mathbf{do}\ \{\ \mathit{ASSERT}\ (l \neq []);\ l' \leftarrow \mathit{sort\_spec}\ l;\ \mathit{RETURN}\ (\mathit{hd}\ l')\ \}$$

$$\mathit{min\_impl}\ l = \mathbf{do}\ \{\ \mathit{ASSERT}\ (l \neq []);\ l' \leftarrow \mathit{sort}\ l;\ \mathit{RETURN}\ (\mathit{hd}\ l')\ \}$$

Demo

Sort_Min.thy

Table of Contents

Fixed points

- Let $f : \alpha \to \alpha$ be a function
  - x with $f\ x = x$ is called fixed point
- Let $\leq$ be a complete lattice, and f be monotonic (i.e. $x \leq y \Longrightarrow f\ x \leq f\ y$)
  - A unique least fixed point lfp f exists
  - Dually, a unique greatest fixed point gfp f exists

Recursion

- Regard recursive function definition

  $$\textbf{let rec } f\ x = F\ f\ x$$

  - F is function body
  - E.g. $F\ f\ x = \textbf{if } x > 0 \textbf{ then } 2 * f\ (x - 1) \textbf{ else } 1$
- For f, we want the following equation
  - $f\ x = F\ f\ x$
  - I.e., f is a fixed point of F

Pointwise ordering, flat lattice

- Given an ordering $\leq \subseteq \alpha \times \alpha$, we extend it to functions $\beta \to \alpha$:
  - $g \leq f \iff \forall x.\ g\ x \leq f\ x$
- Given a set S, we define a complete lattice $\leq$ on $S \dot{\cup} \{\bot, \top\}$

$$\bot \leq \_$$

$$s \leq s \qquad \text{for } s \in S$$

$$\_ \leq \top$$

Recursion as least fixed point

let rec $f\ x = F\ f\ x$

- Now, we define $f = \text{lfp}\ F$
  - Wrt. flat lattice and pointwise ordering
- Intuitively: If $f\ x$ terminates: Only fixed-point is what we want
  - Otherwise: $f\ x = \bot$
- Dually, we could use gfp and get $\top$

Monotonicity, partial and total correctness

- Functions constructed using monad combinators, if-then-else, case, nested fixed-point combinators are monotonic by construction
- Can be automatically proved (Krauss' partial-function package)
- Moreover, for those functions, flat ordering matches refinement ordering
  - $\mathrm{lfp}_{flat} F = \mathrm{lfp}_{ref} F$, where $\bot =$ **succeed** and $\top =$ **fail**
- Thus, when defining a function with lfp
  - On nontermination, we get **succeed**
  - which satisfies any specification

  $\Longrightarrow$ partial correctness
- Dually, for gfp, we get **fail**

  $\Longrightarrow$ total correctness

REC and RECT combinators

- The refinement framework provides
  - $\mathbf{rec}, \mathbf{rec_T} :: ((\alpha \to \beta\ \mathbf{nres}) \to \alpha \to \beta\ \mathbf{nres}) \to \alpha \to \beta\ \mathbf{nres}$
  - $\mathbf{rec}\ F\ x = \mathbf{do}\{\mathbf{assert}(\mathit{mono}\ F); \mathrm{lfp}\ F\ x\}$
  - $\mathbf{rec_T}\ F\ x = \mathbf{do}\{\mathbf{assert}(\mathit{mono}\ F); \mathrm{gfp}\ F\ x\}$
- With proof rules

$$\frac{\mathit{pre}\ x; \forall f\ x.\ (\forall x.\ \mathit{pre}\ x \Longrightarrow f\ x \leq M\ x) \wedge \mathit{pre}\ x \Longrightarrow F\ f\ x \leq M\ x}{\mathbf{rec}\ F\ x \leq M\ x}$$

$$\frac{\mathit{pre}\ x; \forall f\ x.\ (\forall x'.\ \mathit{pre}\ x' \wedge x'\ V x \Longrightarrow f\ x' \leq M\ x') \wedge \mathit{pre}\ x \Longrightarrow F\ f\ x \leq M\ x}{\mathbf{rec_T}\ F\ x \leq M\ x}$$

for well-founded relation V

- And appropriate refinement rules (monotonicity)

WHILE-Loops

- Based on this, we also have while-loops
  - **while** *c f s* - iterate *f* on state *s* as long as *c* holds
  - And also $\textbf{while}_{\textbf{T}}$
- With the expected rules

Demo

Explore_Tree.thy

Table of Contents

Basic Idea

- Refinement not only implements specification by more concrete algorithm
- We also want to implement abstract data structures by more concrete ones
- For example, sets by lists or red-black trees, or hash-tables

Refinement Relation

- Relate concrete type $'c$ to abstract type $'a$
  - Relation $R :: ('c \times' a)\ set$
  - Usually *single-valued*, i.e. $(c, a) \in R \wedge (c, a') \in R \Longrightarrow a = a'$
    - (Right-Unique)
  - But not necessary total: There may be c with $\forall a.\ (c, a) \notin R$
    - Intuition: Concrete type has invariant, e.g., *distinct* list
  - Nor surjective, i.e., there are a with $\forall c.\ (c, a) \notin R$
    - Intuition: Concrete type cannot represent all abstract elements, e.g., only *finite* sets

Invariant and abstraction function

- Consider an invariant $I ::' c \to bool$ and an abstraction function $\alpha ::' c \to' a$
- We define $br\ \alpha\ I = \{(c, \alpha\ c) \mid c.\ I\ c\}$
  - Intuitively: Map concrete elements that satisfy the invariant to abstract elements.
- Exactly the single-valued relations can be represented like this

Concretization function

- Idea: Concrete program refines abstract one:
  - All outcomes in domain of refinement relation
  - All corresponding abstract values in abstract program
- For $R : ('c \times' a)\ set$, we define a concretization function $\Downarrow R :' a\ \mathbf{nres} \to' c\ \mathbf{nres}$

$$\Downarrow R\ (\mathbf{res}\ X) = \mathbf{res}\ (R^{-}\ ``\ X)$$
$$\Downarrow R\ \mathbf{fail} = \mathbf{fail}$$

  - Intuitively, this transforms the abstract program into the *biggest* refining concrete program
- Refinement now expressed by

$$concrete \leq \Downarrow R\ abstract$$

Remark: Galois connection

- For single-valued refinement relations, $\Downarrow R$ is the adjoint of a Galois connection
  - The other adjoint is $\Uparrow R$ defined by

$$\Uparrow R\ (\textbf{res}\ X) = \begin{cases} \textbf{res}\ (R\ ``\ X) & \text{if } X \subseteq \mathit{Domain}\ R \\ \textbf{fail} & \text{otherwise} \end{cases}$$

$$\Uparrow R\ \textbf{fail} = \textbf{fail}$$

- Galois connection means, that we have:

$$m' \leq \Downarrow R\ m \iff \Uparrow R\ m' \leq m$$

  - Intuitively, abstraction and concretization can be swapped
  - This gives us nice mathematical properties
  - But only for single-valued relations
  - Recently, we decided to drop single-valued restrictions where possible

Refinement conditions

- We can derive structure-preserving refinement rules
  - E.g. for return. bind, recursion (show in IDE)
- And build a verification condition generator on them
  - Additionally, there are rules that try to cope with non-exact matches
  - And a tool that helps finding appropriate refinement relations

Demo

Basic_Refinements.thy

Table of Contents

Deterministic Programs

- Executable specifications must be deterministic
- And must not contain **succeed**
- And all used functions should be executable
- Transfer to deterministic monad
  - α **dres** $=$ **succeed$_d$** $\mid$ **res$_d$** α $\mid$ **fail$_d$**
    - **return** $x =$ **res$_d$** x, **res$_d$** $x \gg\!= f = f\ x$, **fail$_d$** $\gg\!= f =$ **fail$_d$**
    - *nres_of* : α **dres** $\rightarrow$ α **nres**
- Transfer preserves structure
  - But has no rules for **res** (nor **spec**)
  - Assertions are dropped
- This can be automated
  - Yields *det* with *nres_of det* $\leq$ *impl*

Getting rid of dres-type

- Additionally, prove that program cannot yield **succeed**
  - Possible for total correct programs
- Then, extract result by selector

  $$\mathit{the\_res} : \alpha\ \mathbf{dres} \to \alpha$$

- And get **return** $(\mathit{the\_res}\ \mathit{det}) \leq \mathit{impl}$

Transfer to plain function

- If program is tail-recursive
  - I.e., only recursion combinator is while
- We can transfer to a plain HOL-definition
  - Without any deterministic monad involved

Recursion Combinators

- Code generator cannot handle recursion combinators (REC, RECT)
- They need to be converted to equations
  - For every instance, as a monotonicity proof is required
- Done automatically by command **prepare_code_thms**

Basic_Refinements.thy

continued

Hands-On session

- Now, it's your turn! Here are some ideas
- Extend graph-exploration/ worklist algorithm to remember visited nodes
  - And thus be total correct for arbitrary (finitely-branching) graphs
    - Hint: *find_theorems finite_psupset*
  - Implement the visited-nodes set by lists or red-black trees
    - You will need *thm rs.correct thm ls.correct*
- Extend the algorithm to return a path to the node

Table of Contents

Table of Contents

Motivation

- Refinement often just replaces abstract by concrete data types
  - E.g. α *set* to α *dlist*
- Tedious to write the algorithm down two times
- Could be automated

Generic Algorithms

- And, while we are automating this
- perhaps throw in some meta-programming
- automatically instantiate generic algorithms?
- E.g., setsum,
  - parameterized by iterator over set

Table of Contents

Recall:

- Refinement relation: Relates concrete and abstract type
- For example, $\langle R \rangle list\_set\_rel$
  - relates distinct lists to sets, members are related by R
- Basic relators
  - Function relator $(f, g) \in A \to B \iff \forall (x, y) \in A.\ (f\ x, f\ y) \in B$
  - Identity $(x, y) \in Id \iff x = y$
- Structure-preserving relators
  - $((a, b), (a', b')) \in A \times_r B \iff (a, a') \in A \wedge (b, b') \in B$
  - Also have $list\_rel, option\_rel, \ldots$

Relators for data refinement

- Consider operation „empty set”: $\{\}$ *::'a set*
  - We have ([],$\{\}$) $\in \langle A \rangle$ *list_set_rel*
  - For any relation A between the elements
- Consider operation „singleton set”: $\lambda x.\ \{x\}$ *::'a* $\Rightarrow$ *'a set*
  - We have ($\lambda x.\ [x]$, $\lambda y.\ \{y\}$) $\in A \to \langle A \rangle$ *list_set_rel*
  - In words: If x implements y then $[x]$ implements y

Synthesis with parametricity

- We have

$$\frac{\forall\, x\ y.\ (x, y) \in A \Longrightarrow (f\ x, g\ y) \in B}{(\lambda x.\ f\ x, \lambda y.\ g\ y) \in A \to B} \quad \text{(abs)}$$

$$\frac{(x, y) \in A; (f, g) \in A \to B}{(f\ x, g\ y) \in B} \quad \text{(app)}$$

- With these, and parametricity rules for the constants, we can synthesize an implementation from the abstract term
  - Compare with lifting and transfer
- BUT:
  - We must choose consistent implementations
  - All abstract operations expressed by single constant
  - Consider a couple of problems, see next slides
- This is exactly what Autoref does

Equality

- Equality is structural equality in HOL
- But structural equality on abstract type need not match structural equality on implementation
  - $[1, 2]$ and $[2, 1]$ both implement the same set

Set is non-free

- Beware of hidden equality
  - Try to implement $\in$ for lists
  - You'll need equality on the elements
  - $(eq,op=) \in A \to A \to bool\_rel$
  - Which may not be structural equality!
- We have $glist\_member :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
- Thus

$$(eq,op=) \in A \to A \to bool\_rel$$
$$\Longrightarrow (glist\_member\ eq,op \in) \in A \to \langle A \rangle list\_set\_rel \to bool\_rel$$

Other type classes

- The same applies for other type classes
- The concrete datatypes need not instantiate them consistently with their abstract counterparts
- Operations on them have to be made explicit
- For example, linear ordering on red-black trees

Generic algorithms

- Consider again the singleton set operation
- It can be (abstractly) expressed by empty set and insertion

$$\{x\} = \mathit{insert}\ x\ \{\}$$

- Thus, if we have implementations for insert and empty-set
- we also get one for singleton set

$$[\![(\mathit{ins\_impl},\mathit{insert}) \in A \to \langle A\rangle Rs \to \langle A\rangle Rs;\ (\mathit{empty\_impl},\{\}) \in \langle A\rangle Rs\ ]\!]$$
$$\Longrightarrow (\ \lambda x.\ \mathit{ins\_impl}\ x\ \mathit{empty\_impl},\ \lambda x.\ \{x\}\ ) \in A \to \langle A\rangle Rs$$

  - Note: This works for any relator *Rs*!

Specialization (Type)

- Apart from generic algorithm, we may still define specialized versions for certain data types
- E.g., we still have $(\lambda x.\ [x],\ \lambda x.\ \{x\}) \in A \to \langle A \rangle \mathit{list\_set\_rel}$

Partially parametric functions

- Consider the function *hd* : α *list* $\rightarrow \alpha$
- For refining the elements of a list, keeping the list structure, we would like to have (*hd*,*hd*) $\in \langle A \rangle$*list_rel* $\rightarrow A$
  - However, we cannot prove that!
  - As *hd* [] $=$ *undefined*, this would imply (*undefined*, *undefined*) $\in A$
  - Which we cannot prove!
- Solution: Restrict parametricity theorem to non-empty lists

$$[\![\, l \neq [];\ (li,l) \in \langle A \rangle list\_rel \,]\!] \Longrightarrow (hd\ li,\ hd\ l) \in A$$

Specialization (Precondition)

- Consider insertion of element into set, implemented on distinct lists
- We need to check whether element is already in
  - Linear time required
- But, sometimes, we know that the element is not in the set e.g.
  if $x \notin s$ **then let** *s=insert x s;* . . . **else** . . .
- In this case, insert can be implemented by *Cons*, in constant time

$$[\![\ x \notin s;\ (xi,x) \in A;\ (l,s) \in \langle A \rangle list\_set\_rel\ ]\!]$$
$$\Longrightarrow (xi\#l,\ insert\ x\ s) \in \langle A \rangle list\_set\_rel$$

Wrap-up

- Idea of automatic refinement via parametricity is very simple
- But lots of things to think of if implemented for the real stuff
  - Abstract operations are not single constants (*Map.empty*, $x \neq \{\}$, ...)
  - Consistent selection of implementations
  - Hidden operations and type-classes
    - we get generic algorithms as a bonus
  - Partial parametricity
    - we get precondition-based specialization as a bonus

Relation to Refinement Framework

- Combinators of nondeterminism monad are parametric
- With relator $(c,a) \in \langle R \rangle nres\_rel \longleftrightarrow c \leq \Downarrow R\ a$
- Thus, automatic refinement just works for them
  - preserving the structure of the program
- Show: *param_RETURN, param_bind, param_RECT*

Table of Contents

Identify

- Try to identify the abstract datatypes and operations
- Rewrite to have each operation represented by a single constant
  - Which uniquely identifies the abstract concept
- Uses a heuristics
  - Typing rules + rewriting
- Example, show some *autoref_itype* rules

Fix Relators

- Infer consistent relators
- Again, typing.
  - With conditional rules for generic algorithms.
- And many heuristics to get a „good” implementation
  - User annotations
  - Priority of implementations (e.g., prefer RBT over list)
  - Homogeneity: Implement types involved in operation the same way
    - $A \cup B$: Try to use the same impl for A, B, and the result
- Note: Does not consider side conditions!

Translate

- Do the translation with the fixed relators
- Try to discharge side-conditions
  - Try specialized rules before more general ones
- Infer operations required for generic algorithms

Demo

Autoref_Basic_Demo.thy