

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 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 2016 to be on your path
`http://isabelle.in.tum.de/index.html`!
 - with the AFP-Component installed
`https://www.isa-afp.org/using.shtml`!

Table of Contents

- 1 Motivation
- 2 Theory of Program Refinement
- 3 Automatic Refinement

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 + 1 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!

Setsum.thy

Table of Contents

- 1 Motivation
- 2 Theory of Program Refinement**
- 3 Automatic Refinement

Table of Contents

1 Motivation

2 Theory of Program Refinement

- The nondeterminism monad

- Refinement Ordering

- Recursion, total and partial correctness

- Data Refinement

- Translation to executable specification

3 Automatic Refinement

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 αM with functions
 - $return : \alpha \rightarrow \alpha M$, $bind : \alpha M \rightarrow (\alpha \rightarrow \beta M) \rightarrow \beta M$
 - $bind\ m\ f$ also written $m \gg= f$
- That satisfy the monad laws

$$return\ x \gg= f = f \quad (id1)$$

$$m \gg= return = m \quad (id2)$$

$$(m \gg= f_1) \gg= f_2 = m \gg= (\lambda x. f_1\ x \gg= f_2) \quad (assoc)$$

- 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* | **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 \implies 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 $m \gg= (\lambda x.f)$
- These shortcuts are enclosed in **do**{...} - 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

```
do {  
  ASSERT ( $l \neq []$ );  
  RETURN (hd  $l$ )  
}
```

```
do {  
  ASSERT ( $s \neq \{\}$ );  
  SPEC ( $\lambda x. x \in s$ )  
}
```

Table of Contents

1 Motivation

2 Theory of Program Refinement

- The nondeterminism monad

- Refinement Ordering

- Recursion, total and partial correctness

- Data Refinement

- Translation to executable specification

3 Automatic Refinement

Refinement Ordering

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

$$\begin{aligned} RES\ X \leq RES\ Y &\longleftrightarrow X \subseteq Y \parallel \\ _ \leq FAIL &\longleftrightarrow True \\ FAIL \leq RES\ _ &\longleftrightarrow False \end{aligned}$$

- 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

$sort_spec\ l = SPEC (\lambda l'. multiset_of\ l = multiset_of\ l' \wedge sorted\ l')$

$sort\ l \leq sort_spec\ l$

$distinct\ l \implies sort\ l \leq sort_spec\ l$

$pre\ a \implies algo\ a \leq SPEC (\lambda r. post\ a\ r)$

(* Compare: Hoare-triple {pre} algo {post} *)

Refinement

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

Examples

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

min_abs $l = \mathbf{do} \{ \mathbf{ASSERT} (l \neq []); l' \leftarrow \text{sort_spec } l; \mathbf{RETURN} (\text{hd } l') \}$

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

Sort_Min.thy

Table of Contents

1 Motivation

2 Theory of Program Refinement

- The nondeterminism monad

- Refinement Ordering

- Recursion, total and partial correctness

- Data Refinement

- Translation to executable specification

3 Automatic Refinement

Fixed points

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

Recursion

- Regard recursive function definition

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

- F is function body
- E.g. $F\ f\ x = \mathbf{if}\ x > 0\ \mathbf{then}\ 2 * f\ (x - 1)\ \mathbf{else}\ 1$
- 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 $\beta \rightarrow \alpha$:
 - $g \leq f \iff \forall x. g\ x \leq f\ x$
- Given a set S , we define a complete lattice \leq on $S \dot{\cup} \{\perp, \top\}$

$$\perp \leq _$$

$$s \leq s$$

$$_ \leq \top$$

for $s \in S$

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: $f\ x = \perp$
- 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
 - $\text{lfp}_{flat} F = \text{lfp}_{ref} F$, where $\perp = \mathbf{succed}$ and $\top = \mathbf{fail}$
- Thus, when defining a function with lfp
 - On nontermination, we get **succed**
 - which satisfies any specification \implies partial correctness
- Dually, for gfp , we get **fail**
 \implies total correctness

REC and RECT combinators

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

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

$$\frac{pre\ x; \forall f\ x. (\forall x'. pre\ x' \wedge x' Vx \implies f\ x' \leq M\ x') \wedge pre\ x \implies F\ f\ x \leq M\ x}{\mathbf{rec}_T\ F\ x \leq M\ x}$$

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 **while** τ
- With the expected rules

Explore_Tree.thy

Table of Contents

1 Motivation

2 Theory of Program Refinement

- The nondeterminism monad

- Refinement Ordering

- Recursion, total and partial correctness

- Data Refinement

- Translation to executable specification

3 Automatic Refinement

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. $(c, a) \in R \wedge (c, a') \in R \implies a = a'$
 - (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 $I ::' c \rightarrow bool$ and an abstraction function $\alpha ::' c \rightarrow' a$
- 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 $\Downarrow R : 'a \text{ nres} \rightarrow' c \text{ nres}$

$$\Downarrow R (\text{res } X) = \text{res } (R^{-1} \text{ `` } X)$$

$$\Downarrow R \text{ fail} = \text{fail}$$

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

$$\text{concrete} \leq \Downarrow R \text{ abstract}$$

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

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

- 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

Basic_Refinements.thy

Table of Contents

1 Motivation

2 Theory of Program Refinement

- The nondeterminism monad

- Refinement Ordering

- Recursion, total and partial correctness

- Data Refinement

- Translation to executable specification

3 Automatic Refinement

Deterministic Programs

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

Getting rid of dres-type

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

$$the_res : \alpha \mathbf{dres} \rightarrow \alpha$$

- And get **return** ($the_res \ det$) $\leq \ impl$

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

- 1 Motivation
- 2 Theory of Program Refinement
- 3 Automatic Refinement**

Table of Contents

1 Motivation

2 Theory of Program Refinement

3 Automatic Refinement

Motivation

Parametricity and Refinement Relations

Phases of Autoref

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

1 Motivation

2 Theory of Program Refinement

3 Automatic Refinement

Motivation

Parametricity and Refinement Relations

Phases of Autoref

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 $(f, g) \in A \rightarrow B \iff \forall (x, y) \in A. (f\ x, g\ y) \in B$
 - 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*, ...

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 $(\lambda x. [x], \lambda y. \{y\}) \in A \rightarrow \langle A \rangle list_set_rel$
 - In words: If x implements y then $[x]$ implements y

Synthesis with parametricity

- We have

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

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

- 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
 - $(eq,op=) \in A \rightarrow A \rightarrow bool_rel$
 - Which may not be structural equality!
- We have $glist_member :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
- Thus

$$(eq,op=) \in A \rightarrow A \rightarrow bool_rel \\ \implies (glist_member\ eq,op \in) \in A \rightarrow \langle A \rangle list_set_rel \rightarrow bool_rel$$

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

$$\{x\} = \text{insert } x \ \{\}$$

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

$$\begin{aligned} & \llbracket (\text{ins_impl}, \text{insert}) \in A \rightarrow \langle A \rangle Rs \rightarrow \langle A \rangle Rs; (\text{empty_impl}, \{\}) \in \langle A \rangle Rs \rrbracket \\ & \implies (\lambda x. \text{ins_impl } x \ \text{empty_impl}, \lambda x. \{x\}) \in A \rightarrow \langle A \rangle Rs \end{aligned}$$

- 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 $(\lambda x. [x], \lambda x. \{x\}) \in A \rightarrow \langle A \rangle list_set_rel$

Partially parametric functions

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

$$\llbracket l \neq []; (li, l) \in \langle A \rangle \text{list_rel} \rrbracket \implies (hd\ li, hd\ l) \in A$$

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** $s = \text{insert } x \ s; \dots$ **else** \dots
- In this case, insert can be implemented by *Cons*, in constant time

$$\begin{aligned} & \llbracket x \notin s; (xi, x) \in A; (l, s) \in \langle A \rangle \text{list_set_rel} \rrbracket \\ & \implies (xi \# l, \text{insert } x \ s) \in \langle A \rangle \text{list_set_rel} \end{aligned}$$

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 $(c,a) \in \langle R \rangle nres_rel \iff c \leq \Downarrow R a$
- Thus, automatic refinement just works for them
 - preserving the structure of the program
- Show: *param_RETURN*, *param_bind*, *param_RECT*

Table of Contents

1 Motivation

2 Theory of Program Refinement

3 Automatic Refinement

Motivation

Parametricity and Refinement Relations

Phases of Autoref

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

Autoref_Basic_Demo.thy