



Interactive Theorem Proving using Isabelle/HOL

Session 10

René Thiemann

Department of Computer Science

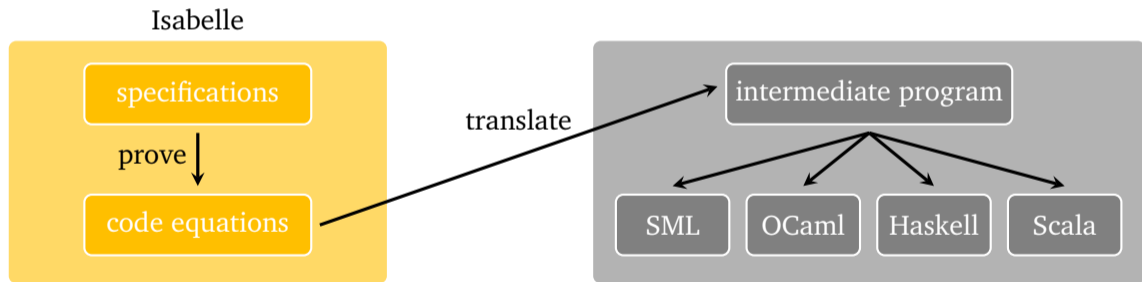
Outline

- Code Generation
- Code Equations Beyond Defining Equations
- Conditional Code Equations

Code Generation

Code Generator Architecture

- **code equations** – executable subset of Isabelle/HOL specifications of shape $f\ t_1\ \dots\ t_n = \dots$
- code equations are translated into **intermediate program** with datatypes and functions
- intermediate program is serialized into concrete programming language



Note

pen-and-paper proof that translation guarantees partial correctness [1]

Usage of the Code Generator

- `value` (code) "sort [7, 4, 8 :: nat]" – evaluate some expression
- `lemma` "sort [7, 4 :: nat] = [4, 7]" `by` `code_simp` – proof by evaluation
- `lemma` "sort [7, 4 :: nat] = [4, 7]" `by` `eval` – proof by evaluation
- `lemma` "sorted [x,y]" `quickcheck` – find counterexample by instantiation and evaluation
- `export_code` sort `in` *Language* – generate code for sort in *Language*

remark: `code_simp` and `eval` differ

- `code_simp` – code equations are applied via Isabelle kernel (trusted)
- `eval` – **reflection mechanism**: code equations are translated to SML, compiled on the fly, then SML evaluation is started, and SML result `true` is reflected as Isabelle result `True` (more efficient)

Exporting Haskell Code

- `code_thms` f – print code equations for f
- `export_code` f g `in` Haskell – generate Haskell code for functions f and g
- `export_code` f `in` Haskell `module_name` Name – generate code as module Name

Demo – Reverse

```
fun rev :: "'a list ⇒ 'a list" where
```

```
  "rev [] = []"
```

```
| "rev (x # xs) = rev xs @ [x]"
```

```
code_thms rev
```

```
export_code rev in Haskell module_name Rev1
```

- append equations are visible in `code_thms`
- however, Isabelle's append is mapped to Haskell's append function (++)
- similarly, Isabelle's list type is mapped to Haskell's list type
- mapping of Isabelle constants/types to target language const./types won't be discussed

Declaring Code Equations

- some commands, like `fun` and `definition`, implicitly declare code equations
- `declare fact [code del]` – remove code equation `fact`
- `declare [[code drop: f ...]]` – remove all code equations for functions `f ...`
- use attribute `[code]` to declare code equation

Demo – Efficient Code of Reverse Function (Program Refinement)

```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
  "itrev [] acc = acc"
| "itrev (x # xs) acc = itrev xs (x # acc)"
```

```
lemma itrev_rev [simp]: "itrev xs ys = rev xs @ ys" <proof>
declare [[code drop: rev]] (* drop old implementation of rev *)
lemma rev_code [code]: "rev xs = itrev xs []" <proof>
code_thms rev (* obtain improved (refined) code equations *)
```

Code Unfold – Automatic Rewriting of Code Equations

- some functions are not executable, in particular if defining equations contain quantifiers
`definition "test0 = (\forall x :: nat. even x)"`
- however, certain patterns with quantifiers look executable
`definition "test1 (xs :: nat list) = (Ball (set xs) even)"`
 reason: bounded quantification over set `xs` is identical to iteration over all list elements
- such an implementation for bounded quantification can be expressed via an equation
`lemma [code_unfold]: "Ball (set xs) p = list_all p xs" <proof>`
- effect of code unfold lemma
 - whenever rhs of code equation contains pattern `Ball (set xs) p` then it will be rewritten to `list_all p xs`
 - in example: code equation for `test1` gets rewritten to
`test1 xs = list_all even xs`

Code Equations might Introduce Type-Class Constraints

- some functions are not executable in their original form
- **code equations can introduce additional type-class constraints**
- example

```
definition test2 :: "('a ⇒ bool) ⇒ bool" where
  "test2 p = (∃ x. p x)"
```

Isabelle generates code for `test2` with the additional restriction that `'a` must be a type in the `enum-class`, i.e., all elements of that type must be enumerable via a list

- consequences
 - `definition "test2_nat = test2 (λ x :: nat. x > 5)"` – code generation fails
 - `definition "test2_char = test2 (λ x. x > CHR 'a')"` – code generation succeeds

Code Equations Beyond Defining Equations

Code Equations – Limits and Opportunities

- limit: via code generation we will only get **partial** correctness
 - **if evaluation** of generated code on input **returns some result**, then this result is correct
- opportunity: code equations can be arbitrary equations that can be proven
- examples
 - program refinement (write more efficient code equations):
`lemma [code]: "rev xs = itrev xs []"`
 - implement any function in a trivial way: `lemma [code]: "f x y = f x y"`
- upcoming: examples illustrating the power of code equations

Code Equations – Partial Implementations

```
definition "complex_predicate (x :: nat) = (x > 894105890)"
(* assume we don't know the rhs, might be complex algorithm *)
```

```
definition "unknown_problem = ( $\exists$  x. complex_predicate x)"
(* unknown problem is not executable *)
```

```
lemma [code]: "unknown_problem = (
  if ( $\exists$  x  $\in$  set [0.. $<10000$ ]. complex_predicate x) then True
  else unknown_problem)" <proof>
(* unknown problem will be executable and loops *)
```

```
lemma [code]: "unknown_problem = (
  if ( $\exists$  x  $\in$  set [0.. $<10000$ ]. complex_predicate x) then True
  else Code.abort (STR 'giving up') ( $\lambda$  _. unknown_problem))" <proof>
(* unknown problem will be executable and fails *)
(* "Code.abort e (% _ . x) = x" in logic; throws error in evaluation *)
```

Code Equations – Phantom Arguments

we can implement Isabelle functions by functions that have auxiliary arguments that just exist in the implementation

```

definition approx_problem :: "nat ⇒ bool" where
  "approx_problem n = unknown_problem"
(* n is phantom argument *)

lemma [code]: "approx_problem n = (if complex_predicate n then True
  else approx_problem (n + 1))" <proof>
(* n controls the implementation *)

lemma [code]: "unknown_problem = approx_problem 0" <proof>

lemma unknown_problem by eval
(* evaluation succeeds because of unbounded iteration *)

```

Approximation Algorithm without Termination Proof

```

definition property :: "real  $\Rightarrow$  bool" ...
definition approx :: "nat  $\Rightarrow$  real  $\Rightarrow$  rat  $\times$  rat" ...
(* approximate real with precision n, e.g., via lower and upper bound *)
definition approx_alg :: "rat  $\times$  rat  $\Rightarrow$  bool option" ...
lemma "approx n r = lu  $\implies$  approx_alg lu = Some b  $\implies$  b = property r"
(* if approximation is successful, then property is determined *)
definition check_property :: "nat  $\Rightarrow$  real  $\Rightarrow$  bool" where
  "check_property n r = property r" (* impl. with phantom argument *)
lemma [code]: "check_property n r =
  (case approx_alg (approx n r) of
    Some b  $\Rightarrow$  b
  | None  $\Rightarrow$  check_property (n+2) r)" (* increase precision by 2 *)
lemma [code]: "property r = check_property 10 r"

```

Conditional Code Equations

Reachability in Graphs – Conditional Code Equations

```
context
```

```
  fixes G :: "'a rel"      (* fix local parameters (here: a graph) *)
  assumes fG: "finite G"  (* add assumptions (here: graph is finite) *)
```

```
begin (* context with G *)
```

```
fun reach_main :: "'a set ⇒ 'a set ⇒ 'a set" where
```

```
  "reach_main todo reached = (if todo = {} then reached
    else let successors = snd ` (Set.filter (λ (x,y). x ∈ todo) G);
          new = successors - reached
        in reach_main new (reached ∪ new))"
```

```
(* termination proof is not automatic, and requires finiteness of G! *)
```

```
definition "reach A = reach_main A A"
```

```
lemma "reach A = {y. ∃ x ∈ A. (x,y) ∈ G*}" <proof>
```

```
end (* of context *)
```

```
thm reach_main.simps (* outside context obtain conditional equation *)
```

```
(* finite G ==> reach_main G todo reached = (if todo = ... ) *)
```


Conditional Code Equations

- problem: conditional code equations $\text{cond } x \implies \text{lhs } x = \text{rhs } x$ are not accepted by code generator: **code equations must be unconditional!**
- possible solutions
 1. move condition into code equation
 $\text{lhs } x = (\text{if } \text{cond } x \text{ then } \text{rhs } x \text{ else } (\text{Code.abort}) (\text{lhs } x))$
 disadvantage: condition is checked in every iteration
 2. create dedicated type `typedef 'a ctyp = { x :: 'a. cond x }`,
 check condition initially once, but not in every iteration,
 work with lift-definitions to convert between types
 3. if the conditional code equations are tail-recursive, use `partial_function` to define equivalent unconditional code equations, avoids type-conversions
 4. just define desired property and from that prove a code equation without explicit function definition
- all solutions will be illustrated via the reachability example

Solution 1 – Move Condition into If-Then-Else

```
definition "err = STR 'reach invoked on infinite graph'"
```

```
lemma [code]:
```

```
"reach_main G todo reached = (if finite G (* check cond *) then
  if todo = {} then reached
  else let successors = snd ` (Set.filter (λ (x,y). x ∈ todo) G);
        new = successors - reached
        in reach_main G new (reached ∪ new)
  else Code.abort err (λ _. reach_main G todo reached))" <proof>
```

```
lemma [code]: "reach G A = (if finite G then reach_main G A A
  else Code.abort err (λ _. reach G A))" <proof>
```

```
value (code) "reach {(1,2 :: nat), (3,4), (2,4), (4,1)} {1}"
(* {4, 2, 1} *)
```

Solution 2 – Create Type for Condition

```
typedef 'a fset = "{ A :: 'a set. finite A}" by auto
setup_lifting type_definition_fset
```

```
lift_definition get_set :: "'a fset ⇒ 'a set" is "λ A. A" .
```

```
lemma "finite (get_set A)" <proof>
```

```
definition "reach_main_2 fG = reach_main (get_set fG)"
```

```
lemma [code]: "reach_main_2 fG todo reached = (if todo = {}
  then reached else let
    successors = snd ` (Set.filter (λ (x,y). x ∈ todo) (get_set fG));
    new = successors - reached
  in reach_main_2 fG new (reached ∪ new))" <proof>
```

Solution 2 – Continued

```
definition "reach_2 fG = reach (get_set fG)"
```

```
lemma [code]: "reach_2 fG A = reach_main_2 fG A A" <proof>
```

(* problems: create elements of fset; get connection to reach *)

```
lift_definition (code_dt) get_fset :: "'a set ⇒ 'a fset option" is
  "λ G. if finite G then Some G else None" <proof>
```

```
lemma [code]: "reach G A = (case get_fset G of
  Some fG ⇒ reach_2 fG A
| None ⇒ Code.abort err (λ _. reach G A))" <proof>
```

(* note: (code_dt) is required to obtain executable code,
since lifted type (fset) is wrapped within other type (option) *)

Solution 3 – `partial_function`

- `partial_function` (tailrec) allows user to specify unconditional defining equation, even if they are non-terminating, provided that the equation is tail-recursive
- syntactic constraints are similar to `definition`, except that recursion is allowed
- logically, non-termination is modeled via undefined

```

partial_function (tailrec) (* copy of reach_main *)
reach_main_3 :: "'a rel ⇒ 'a set ⇒ 'a set ⇒ 'a set" where
[code]: "reach_main_3 G todo reached = (if todo = {} then reached
      else let successors = snd ` (Set.filter (λ (x,y). x ∈ todo) G);
           new = successors - reached
           in reach_main_3 G new (reached ∪ new))"
definition "reach_3 G A = reach_main_3 G A A" (* copy of reach *)

lemma "finite G ⇒ reach_3 G A = reach G A" (* via reach_main.induct *)
lemma [code]: "reach G A = (if finite G then reach_3 G A
      else Code.abort err (λ _. reach G A))" ⟨proof⟩

```

Solution 4 – No Specification of Algorithm, Just Code Equation

```
definition reach' :: "'a rel ⇒ 'a set ⇒ 'a set" where
  "reach' G A = {y. ∃x∈A. (x, y) ∈ G*}"
```

```
lemma [code]: "reach' G A = (if A = {} then {} else
  let A_edges = Set.filter (λ (x,y). x ∈ A) G;
      successors = snd ` A_edges
  in A ∪ reach' (G - A_edges) successors)" <proof>
```

```
value (code) "reach' {(1,2 :: nat), (3,4), (2,4), (4,1)} {1}"
(* {2, 4, 1} *)
```

Further Reading



Florian Haftmann and Tobias Nipkow.

Code generation via higher-order rewrite systems.

In *FLOPS*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[doi:10.1007/978-3-642-12251-4_9](https://doi.org/10.1007/978-3-642-12251-4_9).



Florian Haftmann and Lukas Bulwahn.

Code generation from Isabelle/HOL theories.

isabelle doc codegen, 2021.