universität innsbruck

Interactive Theorem Proving using Isabelle/HOL

Session 11

René Thiemann

Department of Computer Science

Outline

- Code Generation using Target Language Types
- Code Generation with Subtypes
- Datatype Refinement

Previous Lecture

- turn function definitions into programs
- program refinement: change generated code by means of code equations
- 4 ways to handle conditional code equations

This Lecture: Code Generation for Types

- type-synonyms and datatype definitions: trivial
- usage of target language types
- subtypes and lift-definitions
- datatype refinement

Code Generation using Target Language Types

Code Generation using Target Language Types

- examples: map Isabelle lists, integers,... to Haskell lists, integers, ...
- advantages
  - resulting code is most likely more efficient
  - resulting code is more easily accessible; input to function might just be a Haskell type such as [**Integer**], instead of some Isabelle-created list type with elements of some Isabelle-created integer type, which has nothing to do with Haskell's built-in lists and integers
- challenge
  - operations on lists, integers, ... should(!) behave identical, regardless of whether execution is performed w.r.t. their Isabelle specification or whether the target language implementation is invoked

Integration of Target Language Types

- mapping types and constants to target language elements decreases level of trust
  - mapping to target language elements is often optional, e.g., activated only via explicit import of `"HOL-Library.Code_Target_Numeral"`
  - consequence: eases possibility of comparing verified code vs. target language primitives
- reliability is often ensured in form of code equations; these ensure that target-language functions are only invoked on well-defined inputs; example: modulo on integers
  - Isabelle: `x mod 0 = x` and `(-3) mod (-4) = -3`
  - target languages will throw division-by-zero error and might deviate for negative inputs
  - solution: code equation does preprocessing and captures corner cases

```
definition target_mod :: "integer ⇒ integer ⇒ integer" where
  "x > 0 ⟹ y > 0 ⟹ target_mod x y = x mod y"
(* there is some further setup which tells code generator to map
   target_mod to target-language modulo operation *)

(* verified code equation for mod *)
lemma [code]: "x mod y = (if y = 0 then x else
  if x > 0 ∧ y > 0 then target_mod x y else
  if x < 0 ∧ y < 0 then - target_mod (- x) (- y) else ...)" ⟨proof⟩
```

Code Generation with Subtypes

Recall Subtypes

- create a new (abstract) type by restricting a representative type via some predicate
- `Abs` and `Rep` convert between abstract and representative type
- `lift_definition` lifts functions on representative type to abstract type;
  proofs are required that predicate is satisfied whenever elements of abstract type are created

Example – Large Integers

```
typedef large_int = "{ n :: integer. n > 1000}" ⟨proof⟩
setup_lifting type_definition_large_int
lift_definition get_int :: "large_int ⇒ integer" is "λ x. x" .
lift_definition add_10 :: "large_int ⇒ large_int" is "λ x. x + 10" ⟨proof⟩
```

Translation into Code

- Abs and Rep convert between abstract and representative type
  - create datatype for abstract type where Abs is viewed as constructor
  - Rep is selector of that constructor

```
typedef large_int = "{ n :: integer. n > 1000}" ⟨proof⟩
setup_lifting type_definition_large_int
lift_definition get_int :: "large_int ⇒ integer" is "λ x. x" .
lift_definition add_10 :: "large_int ⇒ large_int" is "λ x. x + 10" ⟨proof⟩
```

```
data Large_int = Abs_large_int Integer  {- predicate > 1000 omitted -}

rep_large_int :: Large_int -> Integer   {- rep is just selector -}
rep_large_int (Abs_large_int x) = x     {- predicate missing in equality -}

get_int :: Large_int -> Integer         {- defining equations are easy -}
get_int x = rep_large_int x

add_10 :: Large_int -> Large_int
add_10 x = Abs_large_int (rep_large_int x + 10)
```

Validity of Translated Code

- logic: `x > 1000 ⟹ Rep_large_int (Abs_large_int x) = x`
- code: rep_large_int (Abs_large_int x) = x
-
```
lemma "1000 < (5 :: integer)" proof -
  have "1000 < get_int (Abs_large_int 5)" ⟨proof⟩
  also have "... = Rep_large_int (Abs_large_int 5)" ⟨proof⟩
  also have "... = 5" by eval
  finally show "1000 < 5" .
qed
```
- above Isabelle "proof" is not accepted: abstraction violation in `eval`-method
  - code generator takes care that abstraction functions are only invoked at places where a proof exists that predicate is satisfied (e.g., via `lift_definition`)
  - in particular, code generation will raise abstraction violation error for both `definition "foo x = Abs_large_int x"` and `definition "bar x = Abs_large_int (x * x + 5000)"`
  - warning: after generation of Haskell code, it is no problem to define `foo` manually in Haskell or just write an expression like Abs_large_int 5

Datatype Refinement

Datatype Refinement

- aim: pick any type-constructor and provide implementation of that type and operations
- running example: implement `'a set` and operations like `{}`, `insert`, `(∪)`, `(∈)`, ...
- advantage of datatype refinement
  - state and reason about algorithms abstractly (e.g., using sets)
  - independently verify an executable implementation (e.g., working on lists or trees)
- example from previous lecture

```
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}"
(* upcoming: how does value work in this case? *)
```

Datatype Refinement – First Step: Identify Required Operations

- code equation and invocation provide operations

```
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}"
```

- required operations
  - `Set.is_empty :: 'a set ⇒ bool` — code unfold on `A = {}`
  - `{} :: 'a set`
  - `(∈) :: 'a ⇒ 'a set ⇒ bool`
  - `(∪) :: 'a set ⇒ 'a set ⇒ 'a set`
  - `(-) :: 'a set ⇒ 'a set ⇒ 'a set`
  - ``(`) :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set``
  - `Set.filter :: ('a ⇒ bool) ⇒ 'a set ⇒ 'a set`
  - `insert :: 'a ⇒ 'a set ⇒ 'a set` — from `value` command

Datatype Refinement – Second Step: Implement Required Operations

- example: (extended) implementation of set-operations via ordered trees

```
lift_definition set_o :: "'a :: linorder otree => 'a set" is ...

lift_definition insert_o
  :: "'a :: linorder => 'a otree => 'a otree" is ...
definition union_o
  :: "'a :: linorder otree => 'a otree => 'a otree" where ...
...

(* soundness properties *)
lemma "set_o (insert_o x t) = insert x (set_o t)" ⟨proof⟩
lemma "set_o (union_o t1 t2) = set_o t1 ∪ set_o t2" ⟨proof⟩
...
```

- remark 1: it doesn't matter how the implementation is defined (via `fun`, `definition`, `lift_definition`,...), only the soundness properties are important
- remark 2: one could have used lists, hashmaps, ... instead of trees to represent sets

Datatype Refinement – Third Step: Activate Implementation

- `set_o :: 'a otree ⇒ 'a set` ignoring linorder
  - view `set_o` as constructor of type `'a set`
    - activation in Isabelle: `code_datatype set_o`
    - now code generator interprets type `'a set` as if there would have been a declaration `datatype 'a set = set_o "'a otree"`
    - generated code in Haskell:

```
data Set a = Set_o (Otree a)
-- or equivalent definition via "newtype" instead of "data"
```

- symmetric versions of soundness properties can be used as code equations

```
lemma [code]: "insert x (set_o t) = set_o (insert_o x t)" ⟨proof⟩
lemma [code]: "set_o t1 ∪ set_o t2 = set_o (union_o t1 t2)" ⟨proof⟩
...
```

```
union :: (Eq a, Linorder a) => Set a -> Set a -> Set a
union (Set_o t1) (Set_o t2) = Set_o (union_o t1 t2)
...
```

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.
- Florian Haftmann and Lukas Bulwahn.
  Code generation from Isabelle/HOL theories.
  isabelle doc codegen, 2021.
- Brian Huffman and Ondřej Kunčar.
  Lifting and transfer: A modular design for quotients in Isabelle/HOL.
  In *CPP*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
  doi:10.1007/978-3-319-03545-1_9.