Code generation from Isabelle/HOL theories

Florian Haftmann

8 June 2008

Abstract

This tutorial gives a motivation-driven introduction to a generic code generator framework in Isabelle for generating executable code in functional programming languages from logical specifications.

Chapter 1

Code generation from Isabelle theories

1.1 Introduction

1.1.1 Motivation

Executing formal specifications as programs is a well-established topic in the theorem proving community. With increasing application of theorem proving systems in the area of software development and verification, its relevance manifests for running test cases and rapid prototyping. In logical calculi like constructive type theory, a notion of executability is implicit due to the nature of the calculus. In contrast, specifications in Isabelle can be highly non-executable. In order to bridge the gap between logic and executable specifications, an explicit non-trivial transformation has to be applied: code generation.

This tutorial introduces a generic code generator for the Isabelle system [6]. Generic in the sense that the *target language* for which code shall ultimately be generated is not fixed but may be an arbitrary state-of-the-art functional programming language (currently, the implementation supports SML [5], OCaml [4] and Haskell [7]). We aim to provide a versatile environment suitable for software development and verification, structuring the process of code generation into a small set of orthogonal principles while achieving a big coverage of application areas with maximum flexibility.

Conceptually the code generator framework is part of Isabelle's *Pure* meta logic; the object logic *HOL* which is an extension of *Pure* already comes with a reasonable framework setup and thus provides a good working horse for raising code-generation-driven applications. So, we assume some familiarity and experience with the ingredients of the *HOL Main* theory (see also [6]).

1.1.2 Overview

The code generator aims to be usable with no further ado in most cases while allowing for detailed customization. This manifests in the structure

of this tutorial: we start with a generic example §1.2 and introduce code generation concepts §1.3. Section §1.4 explains how to use the framework naively, presuming a reasonable default setup. Then, section §1.5 deals with advanced topics, introducing further aspects of the code generator framework in a motivation-driven manner. Last, section §1.6 introduces the framework's internal programming interfaces.

! Ultimately, the code generator which this tutorial deals with is supposed to replace the already established code generator by Stefan Berghofer [1]. So, for the moment, there are two distinct code generators in Isabelle. Also note that while the framework itself is object-logic independent, only *HOL* provides a reasonable framework setup.

1.2 An example: a simple theory of search trees

When writing executable specifications using *HOL*, it is convenient to use three existing packages: the datatype package for defining datatypes, the function package for (recursive) functions, and the class package for overloaded definitions.

We develope a small theory of search trees; trees are represented as a datatype with key type $'a$ and value type $'b$:

```
datatype ('a, 'b) searchtree = Leaf 'a::linorder 'b
  | Branch ('a, 'b) searchtree 'a ('a, 'b) searchtree
```

Note that we have constrained the type of keys to the class of total orders, *linorder*.

We define *find* and *update* functions:

```
primrec
  find :: ('a::linorder, 'b) searchtree ⇒ 'a ⇒ 'b option where
  find (Leaf key val) it = (if it = key then Some val else None)
  | find (Branch t1 key t2) it = (if it ≤ key then find t1 it else find t2 it)

fun
  update :: 'a::linorder × 'b ⇒ ('a, 'b) searchtree ⇒ ('a, 'b) searchtree where
  update (it, entry) (Leaf key val) = (
    if it = key then Leaf key entry
      else if it ≤ key
      then Branch (Leaf it entry) it (Leaf key val)
      else Branch (Leaf key val) it (Leaf it entry)
```

)
| *update* (*it*, *entry*) (*Branch* *t*1 *key* *t*2) = (
if *it* ≤ *key*
then (*Branch* (*update* (*it*, *entry*) *t*1) *key* *t*2)
else (*Branch* *t*1 *key* (*update* (*it*, *entry*) *t*2))
)

For testing purpose, we define a small example using natural numbers *nat* (which are a *linorder*) as keys and list of nats as values:

definition
example :: (*nat*, *nat list*) *searchtree*
where
example = *update* (*Suc* (*Suc* (*Suc* (*Suc* 0))), [*Suc* (*Suc* 0), *Suc* (*Suc* 0)]) (*update* (*Suc* (*Suc* (*Suc* 0)), [*Suc* (*Suc* (*Suc* 0))])
(*update* (*Suc* (*Suc* 0), [*Suc* (*Suc* 0)]) (*Leaf* (*Suc* 0) [])))

Then we generate code

export-code *example* **in** *SML* **file** *examples/tree.ML*

which looks like:

```
structure HOL =
struct

type 'a eq = {eq : 'a -> 'a -> bool};
fun eq (A_:'a eq) = #eq A_;

type 'a ord = {less_eq : 'a -> 'a -> bool, less : 'a -> 'a -> bool};
fun less_eq (A_:'a ord) = #less_eq A_;
fun less (A_:'a ord) = #less A_;

fun eqop A_ a = eq A_ a;

end; (*struct HOL*)

structure Orderings =
struct

type 'a order = {Orderings__ord_order : 'a HOL.ord};
fun ord_order (A_:'a order) = #Orderings__ord_order A_;

type 'a linorder = {Orderings__order_linorder : 'a order};
fun order_linorder (A_:'a linorder) = #Orderings__order_linorder A_;

end; (*struct Orderings*)

structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun eq_nat Zero_nat Zero_nat = true
  | eq_nat (Suc m) (Suc n) = eq_nat m n
  | eq_nat Zero_nat (Suc a) = false
  | eq_nat (Suc a) Zero_nat = false;
```

```
val eq_nata = {eq = eq_nat} : nat HOL.eq;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

val ord_nat = {less_eq = less_eq_nat, less = less_nat} : nat HOL.ord;

val order_nat = {Orderings__ord_order = ord_nat} : nat Orderings.order;

val linorder_nat = {Orderings__order_linorder = order_nat} :
  nat Orderings.linorder;

end; (*struct Nat*)

structure Codegen =
struct

datatype ('a, 'b) searchtree =
  Branch of ('a, 'b) searchtree * 'a * ('a, 'b) searchtree |
  Leaf of 'a * 'b;

fun update (A1_, A2_) (it, entry) (Branch (t1, key, t2)) =
  (if HOL.less_eq ((Orderings.ord_order o Orderings.order_linorder) A2_)
        it key
    then Branch (update (A1_, A2_) (it, entry) t1, key, t2)
    else Branch (t1, key, update (A1_, A2_) (it, entry) t2))
  | update (A1_, A2_) (it, entry) (Leaf (key, vala)) =
    (if HOL.eqop A1_ it key then Leaf (key, entry)
      else (if HOL.less_eq
                  ((Orderings.ord_order o Orderings.order_linorder) A2_) it
                  key
             then Branch (Leaf (it, entry), it, Leaf (key, vala))
             else Branch (Leaf (key, vala), it, Leaf (it, entry))));

val example : (Nat.nat, (Nat.nat list)) searchtree =
  update (Nat.eq_nata, Nat.linorder_nat)
    (Nat.Suc (Nat.Suc (Nat.Suc (Nat.Suc Nat.Zero_nat))),
      [Nat.Suc (Nat.Suc Nat.Zero_nat), Nat.Suc (Nat.Suc Nat.Zero_nat)])
    (update (Nat.eq_nata, Nat.linorder_nat)
      (Nat.Suc (Nat.Suc (Nat.Suc Nat.Zero_nat)),
        [Nat.Suc (Nat.Suc (Nat.Suc Nat.Zero_nat))])
      (update (Nat.eq_nata, Nat.linorder_nat)
        (Nat.Suc (Nat.Suc Nat.Zero_nat), [Nat.Suc (Nat.Suc Nat.Zero_nat)])
        (Leaf (Nat.Suc Nat.Zero_nat, []))));

end; (*struct Codegen*)
```

1.3 Code generation concepts and process

The code generator employs a notion of executability for three foundational executable ingredients known from functional programming: *defining equations*, *datatypes*, and *type classes*. A defining equation as a first approximation is a theorem of the form $f\ t_1\ t_2\ \ldots\ t_n \equiv t$ (an equation headed by a

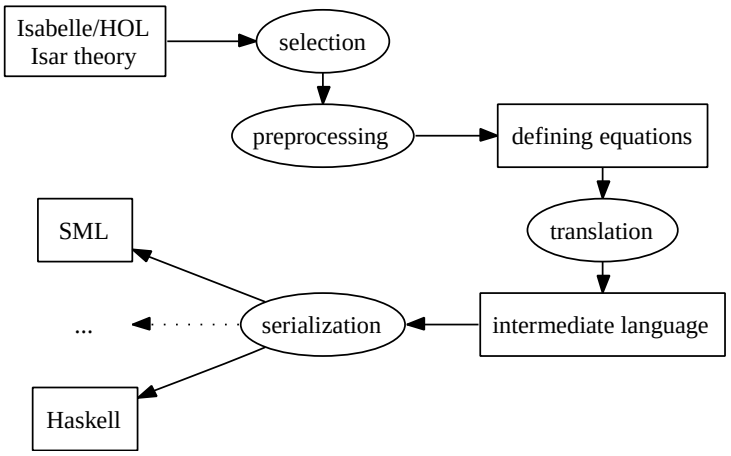

Figure 1.1: code generator – processing overview

constant f with arguments $t_1\ t_2\ \ldots\ t_n$ and right hand side t). Code generation aims to turn defining equations into a functional program by running through a process (see figure 1.1):

- Out of the vast collection of theorems proven in a *theory*, a reasonable subset modeling defining equations is *selected*.
- On those selected theorems, certain transformations are carried out (*preprocessing*). Their purpose is to turn theorems representing non- or badly executable specifications into equivalent but executable counterparts. The result is a structured collection of *code theorems*.
- These *code theorems* then are *translated* into an Haskell-like intermediate language.
- Finally, out of the intermediate language the final code in the desired *target language* is *serialized*.

From these steps, only the two last are carried out outside the logic; by keeping this layer as thin as possible, the amount of code to trust is kept to a minimum.

1.4 Basics

1.4.1 Invoking the code generator

Thanks to a reasonable setup of the *HOL* theories, in most cases code generation proceeds without further ado:

primrec
fac :: *nat* ⇒ *nat* **where**
fac 0 = 1
| *fac* (*Suc n*) = *Suc n* ∗ *fac n*

This executable specification is now turned to SML code:

export-code *fac* **in** *SML* **file** *examples/fac.ML*

The **export_code** command takes a space-separated list of constants together with *serialization directives* These start with a *target language* identifier, followed by a file specification where to write the generated code to.

Internally, the defining equations for all selected constants are taken, including any transitively required constants, datatypes and classes, resulting in the following code:

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

val one_nat : nat = Suc Zero_nat;

fun plus_nat (Suc m) n = plus_nat m (Suc n)
  | plus_nat Zero_nat n = n;

fun times_nat (Suc m) n = plus_nat n (times_nat m n)
  | times_nat Zero_nat n = Zero_nat;

end; (*struct Nat*)

structure Codegen =
struct

fun fac (Nat.Suc n) = Nat.times_nat (Nat.Suc n) (fac n)
  | fac Nat.Zero_nat = Nat.one_nat;

end; (*struct Codegen*)
```

The code generator will complain when a required ingredient does not provide a executable counterpart, e.g. generating code for constants not yielding a defining equation (e.g. the Hilbert choice operation *SOME*):

definition
pick-some :: *'a list* ⇒ *'a* **where**
pick-some xs = (*SOME x*. *x* ∈ *set xs*)
export-code *pick-some* **in** *SML* **file** *examples/fail-const.ML*

will fail.

1.4.2 Theorem selection

The list of all defining equations in a theory may be inspected using the **print_codesetup** command:

print-codesetup

which displays a table of constant with corresponding defining equations (the additional stuff displayed shall not bother us for the moment).

The typical *HOL* tools are already set up in a way that function definitions introduced by **definition**, **primrec**, **fun**, **function**, **constdefs**, **recdef** are implicitly propagated to this defining equation table. Specific theorems may be selected using an attribute: *code func*. As example, a weight selector function:

primrec
pick :: $(nat \times 'a)$ *list* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ **where**
pick $(x\#xs)$ n = (*let* (k, v) = x *in*
if $n < k$ *then* v *else pick* xs $(n - k)$)

We want to eliminate the explicit destruction of x to (k, v):

lemma [*code func*]:
pick $((k, v)\#xs)$ n = (*if* $n < k$ *then* v *else pick* xs $(n - k)$)
by *simp*

export-code *pick* **in** *SML* **file** *examples/pick*1.*ML*

This theorem now is used for generating code:

```
structure HOL =
struct

fun leta s f = f s;

end; (*struct HOL*)

structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

fun minus_nat (Suc m) (Suc n) = minus_nat m n
  | minus_nat Zero_nat n = Zero_nat
  | minus_nat m Zero_nat = m;

end; (*struct Nat*)

structure Product_Type =
struct
```

```
fun split f (a, b) = f a b;

end; (*struct Product_Type*)

structure Codegen =
struct

fun pick ((k, v) :: xs) n =
  (if Nat.less_nat n k then v else pick xs (Nat.minus_nat n k))
  | pick (x :: xs) n =
    let
      val a = x;
      val (k, v) = a;
    in
      (if Nat.less_nat n k then v else pick xs (Nat.minus_nat n k))
    end;

end; (*struct Codegen*)
```

It might be convenient to remove the pointless original equation, using the *func del* attribute:

lemmas [*code func del*] = *pick.simps*

export-code *pick* **in** *SML* **file** *examples/pick2.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

fun minus_nat (Suc m) (Suc n) = minus_nat m n
  | minus_nat Zero_nat n = Zero_nat
  | minus_nat m Zero_nat = m;

end; (*struct Nat*)

structure Codegen =
struct

fun pick ((k, v) :: xs) n =
  (if Nat.less_nat n k then v else pick xs (Nat.minus_nat n k));

end; (*struct Codegen*)
```

Syntactic redundancies are implicitly dropped. For example, using a modified version of the *fac* function as defining equation, the then redundant (since syntactically subsumed) original defining equations are dropped, resulting in a warning:

lemma [*code func*]:

fac n = (*case n of* 0 ⇒ 1 | *Suc m* ⇒ *n* ∗ *fac m*)
by (*cases n*) *simp-all*

export-code *fac* **in** *SML* **file** *examples/fac-case.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

val one_nat : nat = Suc Zero_nat;

fun nat_case f1 f2 Zero_nat = f1
  | nat_case f1 f2 (Suc nat) = f2 nat;

fun plus_nat (Suc m) n = plus_nat m (Suc n)
  | plus_nat Zero_nat n = n;

fun times_nat (Suc m) n = plus_nat n (times_nat m n)
  | times_nat Zero_nat n = Zero_nat;

end; (*struct Nat*)

structure Codegen =
struct

fun fac n =
  (case n of Nat.Zero_nat => Nat.one_nat
     | Nat.Suc m => Nat.times_nat n (fac m));

end; (*struct Codegen*)
```

! The attributes *code* and *code del* associated with the existing code generator also apply to the new one: *code* implies *code func*, and *code del* implies *code func del*.

1.4.3 Type classes

Type classes enter the game via the Isar class package. For a short introduction how to use it, see [2]; here we just illustrate its impact on code generation.

In a target language, type classes may be represented natively (as in the case of Haskell). For languages like SML, they are implemented using *dictionaries*. Our following example specifies a class "null", assigning to each of its inhabitants a "null" value:

class *null* = *type* +
fixes *null* :: *'a*

primrec

head :: *'a*::*null list* ⇒ *'a* **where**
head [] = *null*
| *head* (*x*#*xs*) = *x*

We provide some instances for our *null*:

instantiation *option* **and** *list* :: (*type*) *null*
begin

definition
null = *None*

definition
null = []

instance ..

end

Constructing a dummy example:

definition
dummy = *head* [*Some* (*Suc* 0), *None*]

Type classes offer a suitable occasion to introduce the Haskell serializer. Its usage is almost the same as SML, but, in accordance with conventions some Haskell systems enforce, each module ends up in a single file. The module hierarchy is reflected in the file system, with root directory given as file specification.

export-code *dummy* **in** *Haskell* **file** *examples/*

```
module Codegen where {

import qualified Nat;

class Null a where {
  nulla :: a;
};

heada :: forall a. (Codegen.Null a) => [a] -> a;
heada (x : xs) = x;
heada [] = Codegen.nulla;

null_option :: forall a. Maybe a;
null_option = Nothing;

instance Codegen.Null (Maybe a) where {
  nulla = Codegen.null_option;
};

dummy :: Maybe Nat.Nat;
dummy = Codegen.heada [Just (Nat.Suc Nat.Zero_nat), Nothing];
```

```
}
```

(we have left out all other modules).

The whole code in SML with explicit dictionary passing:

export-code *dummy* **in** *SML* **file** *examples/class.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

end; (*struct Nat*)

structure Codegen =
struct

type 'a null = {null : 'a};
fun null (A_:'a null) = #null A_;

fun head A_ (x :: xs) = x
  | head A_ [] = null A_;

val null_option : 'a option = NONE;

fun null_optiona () = {null = null_option} : ('a option) null;

val dummy : Nat.nat option =
  head (null_optiona ()) [SOME (Nat.Suc Nat.Zero_nat), NONE];

end; (*struct Codegen*)
```

or in OCaml:

export-code *dummy* **in** *OCaml* **file** *examples/class.ocaml*

```
module Nat =
struct

type nat = Suc of nat | Zero_nat;;

end;; (*struct Nat*)

module Codegen =
struct

type 'a null = {null : 'a};;
let null _A = _A.null;;

let rec head _A = function x :: xs -> x
                  | [] -> null _A;;

let rec null_option = None;;

let null_optiona () = ({null = null_option} : ('a option) null);;

let rec dummy
```

```
  = head (null_optiona ()) [Some (Nat.Suc Nat.Zero_nat); None];;

end;; (*struct Codegen*)
```

The explicit association of constants to classes can be inspected using the **print_classes** command.

1.5 Recipes and advanced topics

In this tutorial, we do not attempt to give an exhaustive description of the code generator framework; instead, we cast a light on advanced topics by introducing them together with practically motivated examples. Concerning further reading, see

- the Isabelle/Isar Reference Manual [8] for exhaustive syntax diagrams.
- or [3] which deals with foundational issues of the code generator framework.

1.5.1 Library theories

The *HOL Main* theory already provides a code generator setup which should be suitable for most applications. Common extensions and modifications are available by certain theories of the *HOL* library; beside being useful in applications, they may serve as a tutorial for customizing the code generator setup.

Code-Integer represents *HOL* integers by big integer literals in target languages.

Code-Char represents *HOL* characters by character literals in target languages.

Code-Char-chr like *Code-Char*, but also offers treatment of character codes; includes *Code-Integer*.

Efficient-Nat implements natural numbers by integers, which in general will result in higher efficency; pattern matching with 0 / *Suc* is eliminated; includes *Code-Integer*.

Code-Index provides an additional datatype *index* which is mapped to target-language built-in integers. Useful for code setups which involve e.g. indexing of target-language arrays.

Code-Message provides an additional datatype *message-string* which is isomorphic to strings; *message-string*s are mapped to target-language strings. Useful for code setups which involve e.g. printing (error) messages.

! When importing any of these theories, they should form the last items in an import list. Since these theories adapt the code generator setup in a non-conservative fashion, strange effects may occur otherwise.

1.5.2 Preprocessing

Before selected function theorems are turned into abstract code, a chain of definitional transformation steps is carried out: *preprocessing*. There are three possibilities to customize preprocessing: *inline theorems*, *inline procedures* and *generic preprocessors*.

Inline theorems are rewriting rules applied to each defining equation. Due to the interpretation of theorems of defining equations, rewrites are applied to the right hand side and the arguments of the left hand side of an equation, but never to the constant heading the left hand side. Inline theorems may be declared an undeclared using the *code inline* or *code inline del* attribute respectively. Some common applications:

- replacing non-executable constructs by executable ones:

  lemma [*code inline*]:
  $x \in \mathit{set}\ \mathit{xs} \longleftrightarrow \mathit{x}\ \mathit{mem}\ \mathit{xs}$ **by** (*induct xs*) *simp-all*

- eliminating superfluous constants:

  lemma [*code inline*]:
  $1 = \mathit{Suc}\ 0$ **by** *simp*

- replacing executable but inconvenient constructs:

  lemma [*code inline*]:
  $\mathit{xs} = [] \longleftrightarrow \mathit{List.null}\ \mathit{xs}$ **by** (*induct xs*) *simp-all*

The current set of inline theorems may be inspected using the **print_codesetup** command.

Inline procedures are a generalized version of inline theorems written in ML – rewrite rules are generated dependent on the function theorems for a

certain function. One application is the implicit expanding of *nat* numerals to 0 / *Suc* representation. See further §1.6

Generic preprocessors provide a most general interface, transforming a list of function theorems to another list of function theorems, provided that neither the heading constant nor its type change. The 0 / *Suc* pattern elimination implemented in theory *EfficientNat* (§1.5.1) uses this interface.

! The order in which single preprocessing steps are carried out currently is not specified; in particular, preprocessing is *no* fix point process. Keep this in mind when setting up the preprocessor.

Further, the attribute *code unfold* associated with the existing code generator also applies to the new one: *code unfold* implies *code inline*.

1.5.3 Concerning operational equality

Surely you have already noticed how equality is treated by the code generator:

primrec
collect-duplicates :: *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
collect-duplicates xs ys [] = *xs*
| *collect-duplicates xs ys* (*z*#*zs*) = (*if z* ∈ *set xs*
then if z ∈ *set ys*
then collect-duplicates xs ys zs
else collect-duplicates xs (*z*#*ys*) *zs*
else collect-duplicates (*z*#*xs*) (*z*#*ys*) *zs*)

The membership test during preprocessing is rewritten, resulting in *op mem*, which itself performs an explicit equality check.

export-code *collect-duplicates* **in** *SML* **file** *examples/collect-duplicates.ML*

```
structure HOL =
struct

type 'a eq = {eq : 'a -> 'a -> bool};
fun eq (A_:'a eq) = #eq A_;

fun eqop A_ a = eq A_ a;

end; (*struct HOL*)

structure List =
struct

fun member A_ x (y :: ys) =
  (if HOL.eqop A_ y x then true else member A_ x ys)
  | member A_ x [] = false;

end; (*struct List*)
```

```
structure Codegen =
struct

fun collect_duplicates A_ xs ys (z :: zs) =
  (if List.member A_ z xs
    then (if List.member A_ z ys then collect_duplicates A_ xs ys zs
           else collect_duplicates A_ xs (z :: ys) zs)
    else collect_duplicates A_ (z :: xs) (z :: ys) zs)
  | collect_duplicates A_ xs ys [] = xs;

end; (*struct Codegen*)
```

Obviously, polymorphic equality is implemented the Haskell way using a type class. How is this achieved? HOL introduces an explicit class *eq* with a corresponding operation *eq-class.eq* such that *eq-class.eq* $x\ y = (x = y)$. The preprocessing framework does the rest. For datatypes, instances of *eq* are implicitly derived when possible. For other types, you may instantiate *eq* manually like any other type class.

Though this *eq* class is designed to get rarely in the way, a subtlety enters the stage when definitions of overloaded constants are dependent on operational equality. For example, let us define a lexicographic ordering on tuples:

instantiation $*$ :: (*ord*, *ord*) *ord*
begin

definition
[*code func del*]: $p1 < p2 \longleftrightarrow$ (*let* $(x1, y1) = p1$; $(x2, y2) = p2$ *in*
$x1 < x2 \vee (x1 = x2 \wedge y1 < y2)$)

definition
[*code func del*]: $p1 \leq p2 \longleftrightarrow$ (*let* $(x1, y1) = p1$; $(x2, y2) = p2$ *in*
$x1 < x2 \vee (x1 = x2 \wedge y1 \leq y2)$)

instance ..

end

lemma *ord-prod* [*code func*]:
$(x1 :: 'a{::}ord, y1 :: 'b{::}ord) < (x2, y2) \longleftrightarrow x1 < x2 \vee (x1 = x2 \wedge y1 < y2)$
$(x1 :: 'a{::}ord, y1 :: 'b{::}ord) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee (x1 = x2 \wedge y1 \leq y2)$
unfolding *less-prod-def less-eq-prod-def* **by** *simp-all*

Then code generation will fail. Why? The definition of $op \leq$ depends on equality on both arguments, which are polymorphic and impose an additional *eq* class constraint, thus violating the type discipline for class operations.

The solution is to add *eq* explicitly to the first sort arguments in the code theorems:

lemma *ord-prod-code* [*code func*]:
$(x1 :: 'a{::}\{ord,\ eq\},\ y1 :: 'b{::}ord) < (x2,\ y2) \longleftrightarrow$
$x1 < x2 \vee (x1 = x2 \wedge y1 < y2)$
$(x1 :: 'a{::}\{ord,\ eq\},\ y1 :: 'b{::}ord) \leq (x2,\ y2) \longleftrightarrow$
$x1 < x2 \vee (x1 = x2 \wedge y1 \leq y2)$
unfolding *ord-prod* **by** *rule+*

Then code generation succeeds:

export-code $op \leq :: 'a{::}\{eq,\ ord\} \times 'b{::}ord \Rightarrow 'a \times 'b \Rightarrow bool$
in *SML* **file** *examples/lexicographic.ML*

```
structure HOL =
struct

type 'a eq = {eq : 'a -> 'a -> bool};
fun eq (A_:'a eq) = #eq A_;

type 'a ord = {less_eq : 'a -> 'a -> bool, less : 'a -> 'a -> bool};
fun less_eq (A_:'a ord) = #less_eq A_;
fun less (A_:'a ord) = #less A_;

end; (*struct HOL*)

structure Codegen =
struct

fun less_eq (A1_, A2_) B_ (x1, y1) (x2, y2) =
  HOL.less A2_ x1 x2 orelse HOL.eq A1_ x1 x2 andalso HOL.less_eq B_ y1 y2;

end; (*struct Codegen*)
```

In general, code theorems for overloaded constants may have more restrictive sort constraints than the underlying instance relation between class and type constructor as long as the whole system of constraints is coregular; code theorems violating coregularity are rejected immediately. Consequently, it might be necessary to delete disturbing theorems in the code theorem table, as we have done here with the original definitions *less-prod-def* and *less-eq-prod-def*.

In some cases, the automatically derived defining equations for equality on a particular type may not be appropriate. As example, watch the following datatype representing monomorphic parametric types (where type constructors are referred to by natural numbers):

datatype *monotype* = *Mono nat monotype list*

Then code generation for SML would fail with a message that the generated code conains illegal mutual dependencies: the theorem *Mono tyco*1

*typargs*1 = *Mono tyco*2 *typargs*2 ≡ *tyco*1 = *tyco*2 ∧ *typargs*1 = *typargs*2 already requires the instance *monotype* :: *eq*, which itself requires *Mono tyco*1 *typargs*1 = *Mono tyco*2 *typargs*2 ≡ *tyco*1 = *tyco*2 ∧ *typargs*1 = *typargs*2; Haskell has no problem with mutually recursive *instance* and *function* definitions, but the SML serializer does not support this.

In such cases, you have to provide you own equality equations involving auxiliary constants. In our case, *list-all*2 can do the job:

lemma *monotype-eq-list-all*2 [*code func*]:
*Mono tyco*1 *typargs*1 = *Mono tyco*2 *typargs*2 ⟷
*tyco*1 = *tyco*2 ∧ *list-all*2 (*op* =) *typargs*1 *typargs*2
by (*simp add*: *list-all*2-*eq* [*symmetric*])

does not depend on instance *monotype* :: *eq*:

export-code *op* = :: *monotype* ⇒ *monotype* ⇒ *bool*
in *SML* **file** *examples/monotype.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun eq_nat Zero_nat Zero_nat = true
  | eq_nat (Suc m) (Suc n) = eq_nat m n
  | eq_nat Zero_nat (Suc a) = false
  | eq_nat (Suc a) Zero_nat = false;

end; (*struct Nat*)

structure List =
struct

fun null (x :: xs) = false
  | null [] = true;

fun list_all2 p (x :: xs) (y :: ys) = p x y andalso list_all2 p xs ys
  | list_all2 p xs [] = null xs
  | list_all2 p [] ys = null ys;

end; (*struct List*)

structure Codegen =
struct

datatype monotype = Mono of Nat.nat * monotype list;

fun eq_monotype (Mono (tyco1, typargs1)) (Mono (tyco2, typargs2)) =
  Nat.eq_nat tyco1 tyco2 andalso
    List.list_all2 eq_monotype typargs1 typargs2;

end; (*struct Codegen*)
```

1.5.4 Programs as sets of theorems

As told in §1.3, code generation is based on a structured collection of code theorems. For explorative purpose, this collection may be inspected using the **code_thms** command:

code-thms *op mod* :: *nat* ⇒ *nat* ⇒ *nat*

prints a table with *all* defining equations for *op mod*, including *all* defining equations those equations depend on recursivly. **code_thms** provides a convenient mechanism to inspect the impact of a preprocessor setup on defining equations.

Similarly, the **code_deps** command shows a graph visualizing dependencies between defining equations.

1.5.5 Constructor sets for datatypes

Conceptually, any datatype is spanned by a set of *constructors* of type $\tau = \ldots \Rightarrow \kappa\ \alpha_1\ \ldots\ \alpha_n$ where $\{\alpha_1, \ldots, \alpha_n\}$ is excactly the set of *all* type variables in τ. The HOL datatype package by default registers any new datatype in the table of datatypes, which may be inspected using the **print_codesetup** command.

In some cases, it may be convenient to alter or extend this table; as an example, we will develope an alternative representation of natural numbers as binary digits, whose size does increase logarithmically with its value, not linear [1]. First, the digit representation:

definition *Dig*0 :: *nat* ⇒ *nat* **where**
  *Dig*0 *n* = 2 ∗ *n*

definition *Dig*1 :: *nat* ⇒ *nat* **where**
  *Dig*1 *n* = *Suc* (2 ∗ *n*)

We will use these two "¿digits"¡ to represent natural numbers in binary digits, e.g.:

lemma 42: 42 = *Dig*0 (*Dig*1 (*Dig*0 (*Dig*1 (*Dig*0 1))))
  by (*simp add*: *Dig*0-*def Dig*1-*def*)

Of course we also have to provide proper code equations for the operations, e.g. *op* +:

lemma *plus-Dig* [*code func*]:
  0 + *n* = *n*

[1] Indeed, the *Efficient-Nat* theory 1.5.1 does something similar

$m + 0 = m$
$1 + Dig0\ n = Dig1\ n$
$Dig0\ m + 1 = Dig1\ m$
$1 + Dig1\ n = Dig0\ (n + 1)$
$Dig1\ m + 1 = Dig0\ (m + 1)$
$Dig0\ m + Dig0\ n = Dig0\ (m + n)$
$Dig0\ m + Dig1\ n = Dig1\ (m + n)$
$Dig1\ m + Dig0\ n = Dig1\ (m + n)$
$Dig1\ m + Dig1\ n = Dig0\ (m + n + 1)$
by (*simp-all add*: *Dig*0-*def Dig*1-*def*)

We then instruct the code generator to view 0, 1, *Dig*0 and *Dig*1 as datatype constructors:

code-datatype 0::*nat* 1::*nat Dig*0 *Dig*1

For the former constructor *Suc*, we provide a code equation and remove some parts of the default code generator setup which are an obstacle here:

lemma *Suc-Dig* [*code func*]:
$Suc\ n = n + 1$
by *simp*

declare *One-nat-def* [*code inline del*]
declare *add-Suc-shift* [*code func del*]

This yields the following code:

export-code *op* + :: *nat* ⇒ *nat* ⇒ *nat* **in** *SML* **file** *examples/nat-binary.ML*

```
structure Nat =
struct

datatype nat = Dig1 of nat | Dig0 of nat | One_nat | Zero_nat;

fun plus_nat (Dig1 m) (Dig1 n) = Dig0 (plus_nat (plus_nat m n) One_nat)
  | plus_nat (Dig1 m) (Dig0 n) = Dig1 (plus_nat m n)
  | plus_nat (Dig0 m) (Dig1 n) = Dig1 (plus_nat m n)
  | plus_nat (Dig0 m) (Dig0 n) = Dig0 (plus_nat m n)
  | plus_nat (Dig1 m) One_nat = Dig0 (plus_nat m One_nat)
  | plus_nat One_nat (Dig1 n) = Dig0 (plus_nat n One_nat)
  | plus_nat (Dig0 m) One_nat = Dig1 m
  | plus_nat One_nat (Dig0 n) = Dig1 n
  | plus_nat m Zero_nat = m
  | plus_nat Zero_nat n = n;

end; (*struct Nat*)
```

From this example, it can be easily glimpsed that using own constructor sets is a little delicate since it changes the set of valid patterns for values of that type. Without going into much detail, here some practical hints:

- When changing the constuctor set for datatypes, take care to provide an alternative for the *case* combinator (e.g. by replacing it using the preprocessor).
- Values in the target language need not to be normalized – different values in the target language may represent the same value in the logic (e.g. $Dig1\ 0 = 1$).
- Usually, a good methodology to deal with the subleties of pattern matching is to see the type as an abstract type: provide a set of operations which operate on the concrete representation of the type, and derive further operations by combinations of these primitive ones, without relying on a particular representation.

1.5.6 Customizing serialization

Basics

Consider the following function and its corresponding SML code:

primrec
in-interval :: $nat \times nat \Rightarrow nat \Rightarrow bool$ **where**
in-interval $(k, l)\ n \longleftrightarrow k \leq n \wedge n \leq l$**export-code** *in-interval* **in** *SML* **file** *examples/bool-literal.ML*

```
structure HOL =
struct

datatype boola = False | True;

fun anda x True = x
  | anda x False = False
  | anda True x = x
  | anda False x = False;

end; (*struct HOL*)

structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = HOL.False
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = HOL.True;

end; (*struct Nat*)

structure Codegen =
struct
```

```
fun in_interval (k, l) n =
  HOL.anda (Nat.less_eq_nat k n) (Nat.less_eq_nat n l);

end; (*struct Codegen*)
```

Though this is correct code, it is a little bit unsatisfactory: boolean values and operators are materialized as distinguished entities with have nothing to do with the SML-builtin notion of "bool". This results in less readable code; additionally, eager evaluation may cause programs to loop or break which would perfectly terminate when the existing SML "bool" would be used. To map the HOL "bool" on SML "bool", we may use *custom serializations*:

```
code_type bool
  (SML "bool")
code_const True and False and "op ∧"
  (SML "true" and "false" and "_ andalso _")
```

The **code_type** commad takes a type constructor as arguments together with a list of custom serializations. Each custom serialization starts with a target language identifier followed by an expression, which during code serialization is inserted whenever the type constructor would occur. For constants, **code_const** implements the corresponding mechanism. Each "_" in a serialization expression is treated as a placeholder for the type constructor's (the constant's) arguments.

export-code *in-interval* **in** *SML* **file** *examples/bool-mlbool.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

end; (*struct Nat*)

structure Codegen =
struct

fun in_interval (k, l) n =
  (Nat.less_eq_nat k n) andalso (Nat.less_eq_nat n l);

end; (*struct Codegen*)
```

This still is not perfect: the parentheses around the "andalso" expression are superfluous. Though the serializer by no means attempts to imitate the rich Isabelle syntax framework, it provides some common idioms, notably associative infixes with precedences which may be used here:

```
code_const "op ∧"
  (SML infixl 1 "andalso")
```

export-code *in-interval* **in** *SML* **file** *examples/bool-infix.ML*

```
structure Nat =
struct

datatype nat = Suc of nat | Zero_nat;

fun less_nat m (Suc n) = less_eq_nat m n
  | less_nat n Zero_nat = false
and less_eq_nat (Suc m) n = less_nat m n
  | less_eq_nat Zero_nat n = true;

end; (*struct Nat*)

structure Codegen =
struct

fun in_interval (k, l) n =
  Nat.less_eq_nat k n andalso Nat.less_eq_nat n l;

end; (*struct Codegen*)
```

Next, we try to map HOL pairs to SML pairs, using the infix "`*`" type constructor and parentheses:

```
code_type *
  (SML infix 2 "*")
code_const Pair
  (SML "!((_),/ (_))")
```

The initial bang "`!`" tells the serializer to never put parentheses around the whole expression (they are already present), while the parentheses around argument place holders tell not to put parentheses around the arguments. The slash "`/`" (followed by arbitrary white space) inserts a space which may be used as a break if necessary during pretty printing.

These examples give a glimpse what mechanisms custom serializations provide; however their usage requires careful thinking in order not to introduce inconsistencies – or, in other words: custom serializations are completely axiomatic.

A further noteworthy details is that any special character in a custom serialization may be quoted using "`'`"; thus, in "`fn '_ => _`" the first "`_`" is a proper underscore while the second "`_`" is a placeholder.

The HOL theories provide further examples for custom serializations.

Haskell serialization

For convenience, the default HOL setup for Haskell maps the *eq* class to its counterpart in Haskell, giving custom serializations for the class (**code_class**) and its operation:

```
code_class eq
  (Haskell "Eq" where "op =" ≡ "(==)")

code_const "op ="
  (Haskell infixl 4 "==")
```

A problem now occurs whenever a type which is an instance of *eq* in HOL is mapped on a Haskell-builtin type which is also an instance of Haskell *Eq*:

typedecl *bar*

instantiation *bar* :: *eq*
begin

definition *eq-class.eq* (*x*::*bar*) *y* ⟷ *x* = *y*

instance by *default* (*simp add*: *eq-bar-def*)

end

```
code_type bar
  (Haskell "Integer")
```

The code generator would produce an additional instance, which of course is rejected. To suppress this additional instance, use **code_instance**:

```
code_instance bar :: eq
  (Haskell -)
```

Pretty printing

The serializer provides ML interfaces to set up pretty serializations for expressions like lists, numerals and characters; these are monolithic stubs and should only be used with the theories introduces in §1.5.1.

1.5.7 Cyclic module dependencies

Sometimes the awkward situation occurs that dependencies between definitions introduce cyclic dependencies between modules, which in the Haskell

world leaves you to the mercy of the Haskell implementation you are using, while for SML code generation is not possible.

A solution is to declare module names explicitly. Let use assume the three cyclically dependent modules are named *A*, *B* and *C*. Then, by stating

code-modulename *SML*
A ABC
B ABC
C ABC

we explicitly map all those modules on *ABC*, resulting in an ad-hoc merge of this three modules at serialization time.

1.5.8 Incremental code generation

Code generation is *incremental*: theorems and abstract intermediate code are cached and extended on demand. The cache may be partially or fully dropped if the underlying executable content of the theory changes. Implementation of caching is supposed to transparently hid away the details from the user. Anyway, caching reaches the surface by using a slightly more general form of the **code_thms**, **code_deps** and **export_code** commands: the list of constants may be omitted. Then, all constants with code theorems in the current cache are referred to.

1.6 ML interfaces

Since the code generator framework not only aims to provide a nice Isar interface but also to form a base for code-generation-based applications, here a short description of the most important ML interfaces.

1.6.1 Executable theory content: *Code*

This Pure module implements the core notions of executable content of a theory.

Managing executable content

ML Reference

```
Code.add_func: thm -> theory -> theory
Code.del_func: thm -> theory -> theory
Code.add_funcl: string * thm list Susp.T -> theory -> theory
Code.add_inline: thm -> theory -> theory
Code.del_inline: thm -> theory -> theory
Code.add_inline_proc: string * (theory -> cterm list -> thm list)
    -> theory -> theory
Code.del_inline_proc: string -> theory -> theory
Code.add_preproc: string * (theory -> thm list -> thm list)
    -> theory -> theory
Code.del_preproc: string -> theory -> theory
Code.add_datatype: (string * typ) list -> theory -> theory
Code.get_datatype: theory -> string
    -> (string * sort) list * (string * typ list) list
Code.get_datatype_of_constr: theory -> string -> string option
```

`Code.add_func` *thm thy* adds function theorem *thm* to executable content.

`Code.del_func` *thm thy* removes function theorem *thm* from executable content, if present.

`Code.add_funcl` (*const*, *lthms*) *thy* adds suspended defining equations *lthms* for constant *const* to executable content.

`Code.add_inline` *thm thy* adds inlining theorem *thm* to executable content.

`Code.del_inline` *thm thy* remove inlining theorem *thm* from executable content, if present.

`Code.add_inline_proc` (*name*, *f*) *thy* adds inline procedure *f* (named *name*) to executable content; *f* is a computation of rewrite rules dependent on the current theory context and the list of all arguments and right hand sides of the defining equations belonging to a certain function definition.

`Code.del_inline_proc` *name thy* removes inline procedure named *name* from executable content.

`Code.add_preproc` (*name*, *f*) *thy* adds generic preprocessor *f* (named *name*) to executable content; *f* is a transformation of the defining equations belonging to a certain function definition, depending on the current theory context.

`Code.del_preproc` *name thy* removes generic preprcoessor named *name* from executable content.

`Code.add_datatype` *cs thy* adds a datatype to executable content, with generation set *cs*.

`Code.get_datatype_of_constr` *thy const* returns type constructor corresponding to constructor *const*; returns *NONE* if *const* is no constructor.

1.6.2 Auxiliary

ML **Reference**

```
CodeUnit.read_const: theory -> string -> string
CodeUnit.head_func: thm -> string * ((string * sort) list * typ)
CodeUnit.rewrite_func: thm list -> thm -> thm
```

`CodeUnit.read_const` *thy s* reads a constant as a concrete term expression *s*.

`CodeUnit.head_func` *thm* extracts the constant and its type from a defining equation *thm*.

`CodeUnit.rewrite_func` *rews thm* rewrites a defining equation *thm* with a set of rewrite rules *rews*; only arguments and right hand side are rewritten, not the head of the defining equation.

1.6.3 Implementing code generator applications

Implementing code generator applications on top of the framework set out so far usually not only involves using those primitive interfaces but also storing code-dependent data and various other things.

! Some interfaces discussed here have not reached a final state yet. Changes likely to occur in future.

Data depending on the theory's executable content

Due to incrementality of code generation, changes in the theory's executable content have to be propagated in a certain fashion. Additionally, such changes may occur not only during theory extension but also during theory merge, which is a little bit nasty from an implementation point of view. The framework provides a solution to this technical challenge by providing a functorial data slot `CodeDataFun`; on instantiation of this functor, the following types and operations are required:

type T
val empty: T
val merge: *Pretty.pp* $\rightarrow$ $T * T \rightarrow T$
val purge: *theory option* $\rightarrow$ *CodeUnit.const list option* $\rightarrow$ $T \rightarrow T$

T the type of data to store.

empty initial (empty) data.

merge merging two data slots.

purge thy consts propagates changes in executable content; if possible, the current theory context is handed over as argument *thy* (if there is no current theory context (e.g. during theory merge, `NONE`); *consts* indicates the kind of change: `NONE` stands for a fundamental change which invalidates any existing code, *SOME consts* hints that executable content for constants *consts* has changed.

An instance of `CodeDataFun` provides the following interface:

get: $\mathit{theory} \to T$
change: $\mathit{theory} \to (T \to T) \to T$
change-yield: $\mathit{theory} \to (T \to 'a * T) \to 'a * T$

get retrieval of the current data.

change update of current data (cached!) by giving a continuation.

change-yield update with side result.

Happy proving, happy hacking!

Bibliography

[1] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[2] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar*. http://isabelle.in.tum.de/doc/classes.pdf.

[3] Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 08 2007.

[4] Xavier Leroy et al. *The Objective Caml system – Documentation and user's manual*. http://caml.inria.fr/pub/docs/manual-ocaml/.

[5] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[6] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[7] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[8] Markus Wenzel. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf.