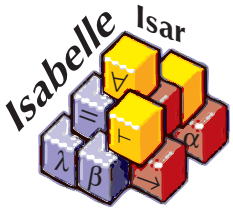

Code generation from Isabelle/HOL theories

Florian Haftmann with contributions from Lukas Bulwahn

3 December 2009

Abstract

This tutorial gives an introduction to a generic code generator framework in Isabelle for generating executable code in functional programming languages from logical specifications in Isabelle/HOL.

1 Introduction and Overview

This tutorial introduces a generic code generator for the *Isabelle* system. 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]).

Conceptually the code generator framework is part of Isabelle's *Pure* meta logic framework; the 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* distribution theories. (see also [6]).

The code generator aims to be usable with no further ado in most cases while allowing for detailed customisation. This manifests in the structure of this tutorial: after a short conceptual introduction with an example (§1.1), we discuss the generic customisation facilities (§2). A further section (§3) is dedicated to the matter of *adaptation* to specific target language environments. After some further issues (§4) we conclude with an overview of some ML programming interfaces (§5).

! Ultimately, the code generator which this tutorial deals with is supposed to replace the existing code generator by Stefan Berghofer [2]. So, for the moment, there are two distinct code generators in Isabelle. In case of ambiguity, we will refer to the framework described here as *generic code generator*, to the other as *SML code generator*. Also note that while the framework itself is object-logic independent, only *HOL* provides a reasonable framework setup.

1.1 Code generation via shallow embedding

The key concept for understanding *Isabelle*'s code generation is *shallow embedding*, i.e. logical entities like constants, types and classes are identified with corresponding concepts in the target language.

Inside *HOL*, the **datatype** and **definition**/**primrec**/**fun** declarations form the core of a functional programming language. The default code generator setup allows to turn those into functional programs immediately. This means that "naive" code generation can proceed without further ado. For example, here a simple "implementation" of amortised queues:

datatype *'a queue* = *AQueue 'a list 'a list*

definition *empty* :: *'a queue* **where**
empty = *AQueue* [] []

primrec *enqueue* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **where**
enqueue x (*AQueue xs ys*) = *AQueue* (*x* # *xs*) *ys*

fun *dequeue* :: *'a queue* ⇒ *'a option* × *'a queue* **where**
dequeue (*AQueue* [] []) = (*None*, *AQueue* [] [])
| *dequeue* (*AQueue xs* (*y* # *ys*)) = (*Some y*, *AQueue xs ys*)
| *dequeue* (*AQueue xs* []) =
(*case rev xs of y* # *ys* ⇒ (*Some y*, *AQueue* [] *ys*))

Then we can generate code e.g. for *SML* as follows:

export-code *empty dequeue enqueue* **in** *SML*
module-name *Example* **file** *examples/example.ML*

resulting in the following code:

```
structure Example =
struct

fun foldl f a [] = a
  | foldl f a (x :: xs) = foldl f (f a x) xs;

fun rev xs = foldl (fn xsa => fn x => x :: xsa) [] xs;

fun list_case f1 f2 (a :: lista) = f2 a lista
  | list_case f1 f2 [] = f1;

datatype 'a queue = AQueue of 'a list * 'a list;

val empty : 'a queue = AQueue ([], []);

fun dequeue (AQueue ([], [])) = (NONE, AQueue ([], []))
  | dequeue (AQueue (xs, y :: ys)) = (SOME y, AQueue (xs, ys))
  | dequeue (AQueue (v :: va, [])) =
    let
      val y :: ys = rev (v :: va);
    in
      (SOME y, AQueue ([], ys))
    end;

fun enqueue x (AQueue (xs, ys)) = AQueue (x :: xs, ys);

end; (*struct Example*)
```

The **export-code** command takes a space-separated list of constants for which code shall be generated; anything else needed for those is added implicitly. Then follows a target language identifier (*SML*, *OCaml* or *Haskell*) and a freely chosen module name. A file name denotes the destination to store the generated code. Note that the semantics of the destination depends

on the target language: for *SML* and *OCaml* it denotes a *file*, for *Haskell* it denotes a *directory* where a file named as the module name (with extension *.hs*) is written:

export-code *empty dequeue enqueue* **in** *Haskell*
  module-name *Example* **file** *examples/*

This is how the corresponding code in *Haskell* looks like:

```
module Example where {


foldla :: forall a b. (a -> b -> a) -> a -> [b] -> a;
foldla f a [] = a;
foldla f a (x : xs) = foldla f (f a x) xs;

rev :: forall a. [a] -> [a];
rev xs = foldla (\ xsa x -> x : xsa) [] xs;

list_case :: forall a b. a -> (b -> [b] -> a) -> [b] -> a;
list_case f1 f2 (a : list) = f2 a list;
list_case f1 f2 [] = f1;

data Queue a = AQueue [a] [a];

empty :: forall a. Queue a;
empty = AQueue [] [];

dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (AQueue [] []) = (Nothing, AQueue [] []);
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys);
dequeue (AQueue (v : va) []) =
  let {
    (y : ys) = rev (v : va);
  } in (Just y, AQueue [] ys);

enqueue :: forall a. a -> Queue a -> Queue a;
enqueue x (AQueue xs ys) = AQueue (x : xs) ys;

}
```

This demonstrates the basic usage of the **export-code** command; for more details see §4.

1.2 Code generator architecture

What you have seen so far should be already enough in a lot of cases. If you are content with this, you can quit reading here. Anyway, in order to customise and adapt the code generator, it is inevitable to gain some understanding how it works.

The code generator employs a notion of executability for three foundational executable ingredients known from functional programming: *code equations*,

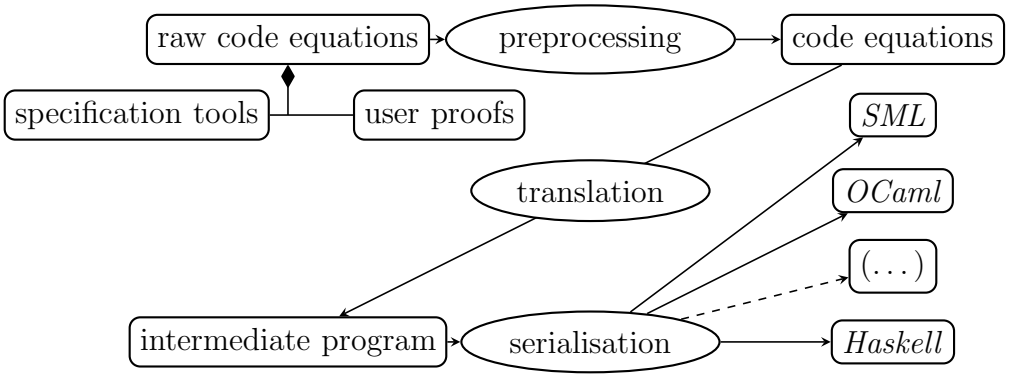

Figure 1: Code generator architecture

datatypes, and *type classes*. A code 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 constant f with arguments $t_1\ t_2\ \ldots\ t_n$ and right hand side t). Code generation aims to turn code equations into a functional program. This is achieved by three major components which operate sequentially, i.e. the result of one is the input of the next in the chain, see figure 1:

- Starting point is a collection of raw code equations in a theory; due to proof irrelevance it is not relevant where they stem from but typically they are either descendant of specification tools or explicit proofs by the user.

- Before these raw code equations are continued with, they can be subjected to theorem transformations. This *preprocessor* is an interface which allows to apply the full expressiveness of ML-based theorem transformations to code generation. The result of the preprocessing step is a structured collection of code equations.

- These code equations are *translated* to a program in an abstract intermediate language. Think of it as a kind of "Mini-Haskell" with four *statements*: *data* (for datatypes), *fun* (stemming from code equations), also *class* and *inst* (for type classes).

- Finally, the abstract program is *serialised* into concrete source code of a target language. This step only produces concrete syntax but does not change the program in essence; all conceptual transformations occur in the translation step.

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.

2 Turning Theories into Programs

2.1 The *Isabelle/HOL* default setup

We have already seen how by default equations stemming from **definition/primrec/fun** statements are used for code generation. This default behaviour can be changed, e.g. by providing different code equations. All kinds of customisation shown in this section is *safe* in the sense that the user does not have to worry about correctness – all programs generatable that way are partially correct.

2.2 Selecting code equations

Coming back to our introductory example, we could provide an alternative code equations for *dequeue* explicitly:

```
lemma [code]:
  dequeue (AQueue xs []) =
    (if xs = [] then (None, AQueue [] [])
      else dequeue (AQueue [] (rev xs)))
  dequeue (AQueue xs (y # ys)) =
    (Some y, AQueue xs ys)
  by (cases xs, simp-all) (cases rev xs, simp-all)
```

The annotation [*code*] is an *Isar attribute* which states that the given theorems should be considered as code equations for a *fun* statement – the corresponding constant is determined syntactically. The resulting code:

```
dequeue :: forall a. Queue a -> (Maybe a, Queue a);
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys);
dequeue (AQueue xs []) =
  (if nulla xs then (Nothing, AQueue [] [])
    else dequeue (AQueue [] (rev xs)));
```

You may note that the equality test $xs = []$ has been replaced by the predicate *null xs*. This is due to the default setup in the *preprocessor* to be discussed further below (§2.4).

Changing the default constructor set of datatypes is also possible. See §2.5 for an example.

As told in §1.2, 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 dequeue
```

prints a table with *all* code equations for *dequeue*, including *all* code equations those equations depend on recursively.

Similarly, the **code-deps** command shows a graph visualising dependencies between code equations.

2.3 *class* and *instantiation*

Concerning type classes and code generation, let us examine an example from abstract algebra:

```
class semigroup =
  fixes mult :: 'a ⇒ 'a ⇒ 'a (infixl ⊗ 70)
  assumes assoc: (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)

class monoid = semigroup +
  fixes neutral :: 'a (1)
  assumes neutl: 1 ⊗ x = x
    and neutr: x ⊗ 1 = x

instantiation nat :: monoid
begin

primrec mult-nat where
    0 ⊗ n = (0::nat)
  | Suc m ⊗ n = n + m ⊗ n

definition neutral-nat where
  1 = Suc 0

lemma add-mult-distrib:
  fixes n m q :: nat
  shows (n + m) ⊗ q = n ⊗ q + m ⊗ q
  by (induct n) simp-all
```

instance proof
  fix m n q :: nat
  show $m \otimes n \otimes q = m \otimes (n \otimes q)$
    by (*induct m*) (*simp-all add*: *add-mult-distrib*)
  show $\mathbf{1} \otimes n = n$
    by (*simp add*: *neutral-nat-def*)
  show $m \otimes \mathbf{1} = m$
    by (*induct m*) (*simp-all add*: *neutral-nat-def*)
qed

end

We define the natural operation of the natural numbers on monoids:

primrec (**in** *monoid*) *pow* :: $nat \Rightarrow {'a} \Rightarrow {'a}$ **where**
    pow 0 $a = \mathbf{1}$
  | *pow* (*Suc n*) $a = a \otimes$ *pow n a*

This we use to define the discrete exponentiation function:

definition *bexp* :: $nat \Rightarrow nat$ **where**
  bexp n = *pow n* (*Suc* (*Suc* 0))

The corresponding code:

```
module Example where {


data Nat = Zero_nat | Suc Nat;

class Semigroup a where {
  mult :: a -> a -> a;
};

class (Semigroup a) => Monoid a where {
  neutral :: a;
};

pow :: forall a. (Monoid a) => Nat -> a -> a;
pow Zero_nat a = neutral;
pow (Suc n) a = mult a (pow n a);

plus_nat :: Nat -> Nat -> Nat;
plus_nat (Suc m) n = plus_nat m (Suc n);
plus_nat Zero_nat n = n;

neutral_nat :: Nat;
neutral_nat = Suc Zero_nat;

mult_nat :: Nat -> Nat -> Nat;
mult_nat Zero_nat n = Zero_nat;
mult_nat (Suc m) n = plus_nat n (mult_nat m n);
```

```
instance Semigroup Nat where {
  mult = mult_nat;
};

instance Monoid Nat where {
  neutral = neutral_nat;
};

bexp :: Nat -> Nat;
bexp n = pow n (Suc (Suc Zero_nat));

}
```

This is a convenient place to show how explicit dictionary construction manifests in generated code (here, the same example in *SML*):

```
structure Example =
struct

datatype nat = Zero_nat | Suc of nat;

type 'a semigroup = {mult : 'a -> 'a -> 'a};
fun mult (A_:'a semigroup) = #mult A_;

type 'a monoid = {semigroup_monoid : 'a semigroup, neutral : 'a};
fun semigroup_monoid (A_:'a monoid) = #semigroup_monoid A_;
fun neutral (A_:'a monoid) = #neutral A_;

fun pow A_ Zero_nat a = neutral A_
  | pow A_ (Suc n) a = mult (semigroup_monoid A_) a (pow A_ n a);

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

val neutral_nat : nat = Suc Zero_nat;

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

val semigroup_nat = {mult = mult_nat} : nat semigroup;

val monoid_nat = {semigroup_monoid = semigroup_nat, neutral = neutral_nat}
  : nat monoid;

fun bexp n = pow monoid_nat n (Suc (Suc Zero_nat));

end; (*struct Example*)
```

Note the parameters with trailing underscore (`A_`) which are the dictionary parameters.

2.4 The preprocessor

Before selected function theorems are turned into abstract code, a chain of definitional transformation steps is carried out: *preprocessing*. In essence,

the preprocessor consists of two components: a *simpset* and *function transformers*.

The *simpset* allows to employ the full generality of the Isabelle simplifier. Due to the interpretation of theorems as code 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. An important special case are *unfold theorems* which may be declared and undeclared using the *code-unfold* or *code-unfold del* attribute respectively.

Some common applications:

- replacing non-executable constructs by executable ones:

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

- eliminating superfluous constants:

  lemma [*code-inline*]:
  $1 = Suc\ 0$ **by** *simp*

- replacing executable but inconvenient constructs:

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

Function transformers provide a very 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 *Efficient-Nat* (see §3.3) uses this interface.
The current setup of the preprocessor may be inspected using the **print-codeproc** command. **code-thms** provides a convenient mechanism to inspect the impact of a preprocessor setup on code equations.

! Attribute *code-unfold* also applies to the preprocessor of the ancient *SML code generator*; in case this is not what you intend, use *code-inline* instead.

2.5 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 exactly 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 is appropriate to alter or extend this table. As an example, we will develop an alternative representation of the queue example given in §1.1. The amortised representation is convenient for generating code but exposes its "implementation" details, which may be cumbersome when proving theorems about it. Therefore, here a simple, straightforward representation of queues:

```
datatype 'a queue = Queue 'a list

definition empty :: 'a queue where
  empty = Queue []

primrec enqueue :: 'a ⇒ 'a queue ⇒ 'a queue where
  enqueue x (Queue xs) = Queue (xs @ [x])

fun dequeue :: 'a queue ⇒ 'a option × 'a queue where
    dequeue (Queue []) = (None, Queue [])
  | dequeue (Queue (x # xs)) = (Some x, Queue xs)
```

This we can use directly for proving; for executing, we provide an alternative characterisation:

```
definition AQueue :: 'a list ⇒ 'a list ⇒ 'a queue where
  AQueue xs ys = Queue (ys @ rev xs)

code-datatype AQueue
```

Here we define a "constructor" *AQueue* which is defined in terms of *Queue* and interprets its arguments according to what the *content* of an amortised queue is supposed to be. Equipped with this, we are able to prove the following equations for our primitive queue operations which "implement" the simple queues in an amortised fashion:

```
lemma empty-AQueue [code]:
  empty = AQueue [] []
```

unfolding *AQueue-def empty-def* **by** *simp*

lemma *enqueue-AQueue* [*code*]:
enqueue x (*AQueue xs ys*) = *AQueue* (*x* # *xs*) *ys*
unfolding *AQueue-def* **by** *simp*

lemma *dequeue-AQueue* [*code*]:
dequeue (*AQueue xs* []) =
(*if xs* = [] *then* (*None*, *AQueue* [] []))
else dequeue (*AQueue* [] (*rev xs*)))
dequeue (*AQueue xs* (*y* # *ys*)) = (*Some y*, *AQueue xs ys*)
unfolding *AQueue-def* **by** *simp-all*

For completeness, we provide a substitute for the *case* combinator on queues:

lemma *queue-case-AQueue* [*code*]:
queue-case f (*AQueue xs ys*) = *f* (*ys* @ *rev xs*)
unfolding *AQueue-def* **by** *simp*

The resulting code looks as expected:

```
structure Example =
struct

fun foldl f a [] = a
  | foldl f a (x :: xs) = foldl f (f a x) xs;

fun rev xs = foldl (fn xsa => fn x => x :: xsa) [] xs;

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

datatype 'a queue = AQueue of 'a list * 'a list;

val empty : 'a queue = AQueue ([], []);

fun dequeue (AQueue (xs, y :: ys)) = (SOME y, AQueue (xs, ys))
  | dequeue (AQueue (xs, [])) =
    (if null xs then (NONE, AQueue ([], []))
      else dequeue (AQueue ([], rev xs)));

fun enqueue x (AQueue (xs, ys)) = AQueue (x :: xs, ys);

end; (*struct Example*)
```

From this example, it can be 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 constructor set for datatypes, take care to provide alternative equations for the *case* combinator.

- Values in the target language need not to be normalised – different values in the target language may represent the same value in the logic.
- Usually, a good methodology to deal with the subtleties 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.

2.6 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.

```
structure Example =
struct

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

fun eqa A_ a b = eq A_ a b;

fun member A_ x [] = false
  | member A_ x (y :: ys) = eqa A_ x y orelse member A_ x ys;

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

end; (*struct Example*)
```

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* = *op* =. The

preprocessing framework does the rest by propagating the *eq* constraints through all dependent code equations. For datatypes, instances of *eq* are implicitly derived when possible. For other types, you may instantiate *eq* manually like any other type class.

2.7 Explicit partiality

Partiality usually enters the game by partial patterns, as in the following example, again for amortised queues:

definition *strict-dequeue* :: *'a queue* ⇒ *'a* × *'a queue* **where**
strict-dequeue q = (*case dequeue q*
of (*Some x*, *q'*) ⇒ (*x*, *q'*))

lemma *strict-dequeue-AQueue* [*code*]:
strict-dequeue (*AQueue xs* (*y* # *ys*)) = (*y*, *AQueue xs ys*)
strict-dequeue (*AQueue xs* []) =
(*case rev xs of y* # *ys* ⇒ (*y*, *AQueue* [] *ys*))
by (*simp-all add*: *strict-dequeue-def dequeue-AQueue split*: *list.splits*)

In the corresponding code, there is no equation for the pattern *AQueue* [] []:

```
strict_dequeue :: forall a. Queue a -> (a, Queue a);
strict_dequeue (AQueue xs []) =
  let {
    (y : ys) = rev xs;
  } in (y, AQueue [] ys);
strict_dequeue (AQueue xs (y : ys)) = (y, AQueue xs ys);
```

In some cases it is desirable to have this pseudo-"partiality" more explicitly, e.g. as follows:

axiomatization *empty-queue* :: *'a*

definition *strict-dequeue'* :: *'a queue* ⇒ *'a* × *'a queue* **where**
strict-dequeue' q = (*case dequeue q of* (*Some x*, *q'*) ⇒ (*x*, *q'*) | - ⇒ *empty-queue*)

lemma *strict-dequeue'-AQueue* [*code*]:
strict-dequeue' (*AQueue xs* []) = (*if xs* = [] *then empty-queue*
else strict-dequeue' (*AQueue* [] (*rev xs*)))
strict-dequeue' (*AQueue xs* (*y* # *ys*)) =
(*y*, *AQueue xs ys*)
by (*simp-all add*: *strict-dequeue'-def dequeue-AQueue split*: *list.splits*)

Observe that on the right hand side of the definition of *strict-dequeue'* the constant *empty-queue* occurs which is unspecified.

Normally, if constants without any code equations occur in a program, the code generator complains (since in most cases this is not what the user expects). But such constants can also be thought of as function definitions with no equations which always fail, since there is never a successful pattern match on the left hand side. In order to categorise a constant into that category explicitly, use **code-abort**:

code-abort *empty-queue*

Then the code generator will just insert an error or exception at the appropriate position:

```
empty_queue :: forall a. a;
empty_queue = error "empty_queue";

strict_dequeue' :: forall a. Queue a -> (a, Queue a);
strict_dequeue' (AQueue xs (y : ys)) = (y, AQueue xs ys);
strict_dequeue' (AQueue xs []) =
  (if nulla xs then empty_queue
    else strict_dequeue' (AQueue [] (rev xs)));
```

This feature however is rarely needed in practice. Note also that the *HOL* default setup already declares *undefined* as **code-abort**, which is most likely to be used in such situations.

2.8 Inductive Predicates

To execute inductive predicates, a special preprocessor, the predicate compiler, generates code equations from the introduction rules of the predicates. The mechanisms of this compiler are described in [1]. Consider the simple predicate *append* given by these two introduction rules:

$$append\ []\ ys\ ys$$
$$append\ xs\ ys\ zs \Longrightarrow append\ (x\ \#\ xs)\ ys\ (x\ \#\ zs)$$

To invoke the compiler, simply use **code-pred**:

code-pred *append* **.**

The **code-pred** command takes the name of the inductive predicate and then you put a period to discharge a trivial correctness proof. The compiler infers possible modes for the predicate and produces the derived code equations. Modes annotate which (parts of the) arguments are to be taken as input, and which output. Modes are similar to types, but use the notation i for input and o for output.

For *append*, the compiler can infer the following modes:

- $i \Rightarrow i \Rightarrow i \Rightarrow bool$
- $i \Rightarrow i \Rightarrow o \Rightarrow bool$
- $o \Rightarrow o \Rightarrow i \Rightarrow bool$

You can compute sets of predicates using **values**:

values {*zs*. *append* [(1::*nat*),2,3] [4,5] *zs*}

outputs {[1, 2, 3, 4, 5]}, and

values {(*xs*, *ys*). *append xs ys* [(2::*nat*),3]}

outputs {([], [2, 3]), ([2], [3]), ([2, 3], [])}.

If you are only interested in the first elements of the set comprehension (with respect to a depth-first search on the introduction rules), you can pass an argument to **values** to specify the number of elements you want:

values 1 {(*xs*, *ys*). *append xs ys* [(1::*nat*),2,3,4]}
values 3 {(*xs*, *ys*). *append xs ys* [(1::*nat*),2,3,4]}

The **values** command can only compute set comprehensions for which a mode has been inferred.

The code equations for a predicate are made available as theorems with the suffix *equation*, and can be inspected with:

thm *append.equation*

More advanced options are described in the following subsections.

Alternative names for functions

By default, the functions generated from a predicate are named after the predicate with the mode mangled into the name (e.g., *append-i-i-o*). You can specify your own names as follows:

code-pred (*modes*: *i* => *i* => *o* => *bool as concat*,
o => *o* => *i* => *bool as split*,
i => *o* => *i* => *bool as suffix*) *append* .

Alternative introduction rules

Sometimes the introduction rules of an predicate are not executable because they contain non-executable constants or specific modes could not be inferred. It is also possible that the introduction rules yield a function that loops forever due to the execution in a depth-first search manner. Therefore, you can declare alternative introduction rules for predicates with the attribute *code-pred-intro*. For example, the transitive closure is defined by:

$$r\ a\ b \Longrightarrow r^{++}\ a\ b$$
$$[\![r^{++}\ a\ b;\ r\ b\ c]\!] \Longrightarrow r^{++}\ a\ c$$

These rules do not suit well for executing the transitive closure with the mode $(i \Rightarrow o \Rightarrow bool) \Rightarrow i \Rightarrow o \Rightarrow bool$, as the second rule will cause an infinite loop in the recursive call. This can be avoided using the following alternative rules which are declared to the predicate compiler by the attribute *code-pred-intro*:

lemma [*code-pred-intro*]:
$r\ a\ b \Longrightarrow r^{++}\ a\ b$
$r\ a\ b \Longrightarrow r^{++}\ b\ c \Longrightarrow r^{++}\ a\ c$
by *auto*

After declaring all alternative rules for the transitive closure, you invoke **code-pred** as usual. As you have declared alternative rules for the predicate, you are urged to prove that these introduction rules are complete, i.e., that you can derive an elimination rule for the alternative rules:

code-pred *tranclp*
proof −
case *tranclp*
from *this converse-tranclpE*[*OF this*(1)] **show** *thesis* **by** *metis*
qed

Alternative rules can also be used for constants that have not been defined inductively. For example, the lexicographic order which is defined as:

$$
\begin{array}{l}
\mathit{lexord}\ r \equiv \\
\{(x,\ y).\ \\
\ \exists\, a\ v.\ y = x\ @\ a\ \#\ v \vee \\
\qquad (\exists\, u\ a\ b\ v\ w.\ (a,\ b) \in r \wedge x = u\ @\ a\ \#\ v \wedge y = u\ @\ b\ \#\ w)\}
\end{array}
$$

To make it executable, you can derive the following two rules and prove the elimination rule:

lemma [*code-pred-intro*]:
$\mathit{append}\ \mathit{xs}\ (a\ \#\ v)\ \mathit{ys} \Longrightarrow \mathit{lexord}\ r\ (\mathit{xs},\ \mathit{ys})$
lemma [*code-pred-intro*]:
$\mathit{append}\ u\ (a\ \#\ v)\ \mathit{xs} \Longrightarrow \mathit{append}\ u\ (b\ \#\ w)\ \mathit{ys} \Longrightarrow r\ (a,\ b)$
$\Longrightarrow \mathit{lexord}\ r\ (\mathit{xs},\ \mathit{ys})$
code-pred *lexord*

Options for values

In the presence of higher-order predicates, multiple modes for some predicate could be inferred that are not disambiguated by the pattern of the set comprehension. To disambiguate the modes for the arguments of a predicate, you can state the modes explicitly in the **values** command. Consider the simple predicate *succ*:

inductive $\mathit{succ} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
where
$\mathit{succ}\ 0\ (\mathit{Suc}\ 0)$
$|\ \mathit{succ}\ x\ y \Longrightarrow \mathit{succ}\ (\mathit{Suc}\ x)\ (\mathit{Suc}\ y)$

code-pred *succ* **.**

For this, the predicate compiler can infer modes $o \Rightarrow o \Rightarrow \mathit{bool}$, $i \Rightarrow o \Rightarrow \mathit{bool}$, $o \Rightarrow i \Rightarrow \mathit{bool}$ and $i \Rightarrow i \Rightarrow \mathit{bool}$. The invocation of **values** $\{n.\ \mathit{tranclp}\ \mathit{succ}\ 10\ n\}$ loops, as multiple modes for the predicate *succ* are possible and here the first mode $o \Rightarrow o \Rightarrow \mathit{bool}$ is chosen. To choose another mode for the argument, you can declare the mode for the argument between the **values** and the number of elements.

values [*mode*: i => o => *bool*] 20 $\{n.\ \mathit{tranclp}\ \mathit{succ}\ 10\ n\}$
values [*mode*: o => i => *bool*] 10 $\{n.\ \mathit{tranclp}\ \mathit{succ}\ n\ 10\}$

Embedding into functional code within Isabelle/HOL

To embed the computation of an inductive predicate into functions that are defined in Isabelle/HOL, you have a number of options:

- You want to use the first-order predicate with the mode where all arguments are input. Then you can use the predicate directly, e.g.

  valid-suffix ys zs =
  (*if append* [*Suc* 0, 2] *ys zs then Some ys else None*)

- If you know that the execution returns only one value (it is deterministic), then you can use the combinator *Predicate.the*, e.g., a functional concatenation of lists is defined with

  functional-concat xs ys = *Predicate.the* (*append-i-i-o xs ys*)

  Note that if the evaluation does not return a unique value, it raises a run-time error *not-unique*.

Further Examples

Further examples for compiling inductive predicates can be found in the *HOL/ex/Predicate-Compile-ex* theory file. There are also some examples in the Archive of Formal Proofs, notably in the *POPLmark−deBruijn* and the *FeatherweightJava* sessions.

3 Adaptation to target languages

3.1 Adapting code generation

The aspects of code generation introduced so far have two aspects in common:

- They act uniformly, without reference to a specific target language.
- They are *safe* in the sense that as long as you trust the code generator meta theory and implementation, you cannot produce programs that yield results which are not derivable in the logic.

In this section we will introduce means to *adapt* the serialiser to a specific target language, i.e. to print program fragments in a way which accommodates "already existing" ingredients of a target language environment, for three reasons:

- improving readability and aesthetics of generated code
- gaining efficiency
- interface with language parts which have no direct counterpart in *HOL* (say, imperative data structures)

Generally, you should avoid using those features yourself *at any cost*:

- The safe configuration methods act uniformly on every target language, whereas for adaptation you have to treat each target language separate.
- Application is extremely tedious since there is no abstraction which would allow for a static check, making it easy to produce garbage.
- More or less subtle errors can be introduced unconsciously.

However, even if you ought refrain from setting up adaptation yourself, already the *HOL* comes with some reasonable default adaptations (say, using target language list syntax). There also some common adaptation cases which you can setup by importing particular library theories. In order to understand these, we provide some clues here; these however are not supposed to replace a careful study of the sources.

3.2 The adaptation principle

Figure 2 illustrates what "adaptation" is conceptually supposed to be:
In the tame view, code generation acts as broker between *logic*, *intermediate language* and *target language* by means of *translation* and *serialisation*; for the latter, the serialiser has to observe the structure of the *language* itself plus some *reserved* keywords which have to be avoided for generated code. However, if you consider *adaptation* mechanisms, the code generated by the serializer is just the tip of the iceberg:

- *serialisation* can be *parametrised* such that logical entities are mapped to target-specific ones (e.g. target-specific list syntax, see also §3.4)

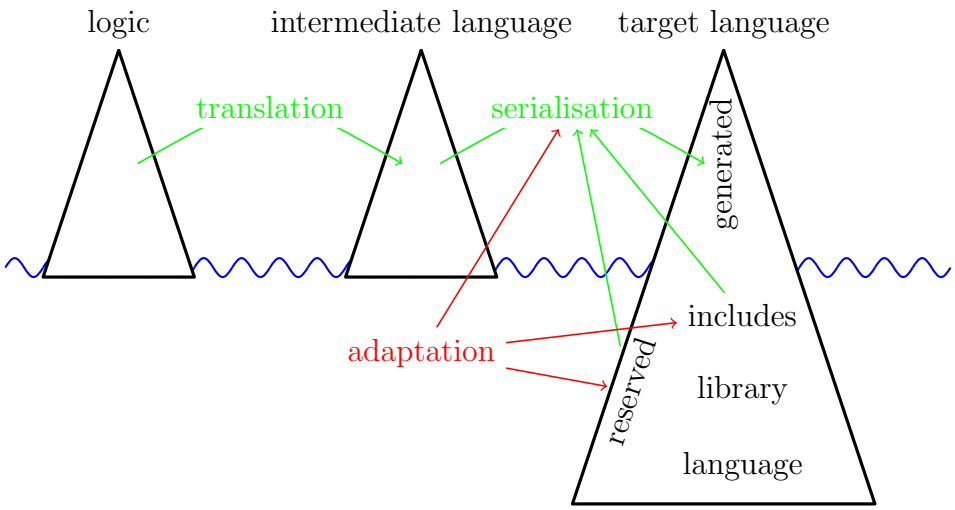

Figure 2: The adaptation principle

- Such parametrisations can involve references to a target-specific standard *library* (e.g. using the *Haskell* `Maybe` type instead of the *HOL* *option* type); if such are used, the corresponding identifiers (in our example, `Maybe`, `Nothing` and `Just`) also have to be considered *reserved*.
- Even more, the user can enrich the library of the target-language by providing code snippets ("*includes*") which are prepended to any generated code (see §3.6); this typically also involves further *reserved* identifiers.

As figure 2 illustrates, all these adaptation mechanisms have to act consistently; it is at the discretion of the user to take care for this.

3.3 Common adaptation patterns

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 customising the code generator setup (see below §3.4).

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-Char*.

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

Code-Numeral 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.

String provides an additional datatype *String.literal* which is isomorphic to strings; *String.literal*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.

3.4 Parametrising serialisation

Consider the following function and its corresponding SML code:

primrec *in-interval* :: $nat \times nat \Rightarrow nat \Rightarrow bool$ **where**
in-interval $(k, l)\ n \longleftrightarrow k \leq n \wedge n \leq l$

```
structure Example =
struct

datatype nat = Zero_nat | Suc of nat;

datatype boola = True | False;

fun anda p True = p
  | anda p False = False
  | anda True p = p
  | anda False p = False;

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 in_interval (k, l) n = anda (less_eq_nat k n) (less_eq_nat n l);

end; (*struct Example*)
```

Though this is correct code, it is a little bit unsatisfactory: boolean values and operators are materialised as distinguished entities with have nothing to do with the SML-built-in 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 serialisations*:

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

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

```
structure Example =
struct

datatype nat = Zero_nat | Suc of 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 in_interval (k, l) n = (less_eq_nat k n) andalso (less_eq_nat n l);

end; (*struct Example*)
```

This still is not perfect: the parentheses around the "andalso" expression are superfluous. Though the serialiser 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")

structure Example =
struct

datatype nat = Zero_nat | Suc of 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 in_interval (k, l) n = less_eq_nat k n andalso less_eq_nat n l;

end; (*struct Example*)
```

The attentive reader may ask how we assert that no generated code will accidentally overwrite. For this reason the serialiser has an internal table of identifiers which have to be avoided to be used for new declarations. Initially, this table typically contains the keywords of the target language. It can be extended manually, thus avoiding accidental overwrites, using the **code-reserved** command:

code-reserved *SML bool true false andalso*

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 serialiser never to 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 serialisations provide; however their usage requires careful thinking in order not to introduce inconsistencies – or, in other words: custom serialisations are completely axiomatic.

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

3.5 *Haskell* serialisation

For convenience, the default *HOL* setup for *Haskell* maps the *eq* class to its counterpart in *Haskell*, giving custom serialisations for the class *eq* (by command **code-class**) and its operation *eq-class.eq*

```
code_class eq
  (Haskell "Eq")

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*-built-in 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 by the *Haskell* compiler. To suppress this additional instance, use *code-instance*:

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

3.6 Enhancing the target language context

In rare cases it is necessary to *enrich* the context of a target language; this is accomplished using the **code-include** command:

```
code_include Haskell "Errno"
{*errno i = error ("Error number: " ++ show i)*}

code_reserved Haskell Errno
```

Such named *include*s are then prepended to every generated code. Inspect such code in order to find out how **code-include** behaves with respect to a particular target language.

4 Further issues

4.1 Further reading

Do dive deeper into the issue of code generation, you should visit the Isabelle/Isar Reference Manual [8] which contains exhaustive syntax diagrams.

4.2 Modules

When invoking the **export-code** command it is possible to leave out the **module-name** part; then code is distributed over different modules, where the module name space roughly is induced by the *Isabelle* theory name space.

Then 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*/*OCaml* 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 serialisation time.

4.3 Evaluation oracle

Code generation may also be used to *evaluate* expressions (using *SML* as target language of course). For instance, the **value** allows to reduce an expression to a normal form with respect to the underlying code equations:

value 42 / (12 :: *rat*)

will display 7 / 2.

The *eval* method tries to reduce a goal by code generation to *True* and solves it in that case, but fails otherwise:

lemma 42 / (12 :: *rat*) = 7 / 2
  by *eval*

The soundness of the *eval* method depends crucially on the correctness of the code generator; this is one of the reasons why you should not use adaptation (see §3) frivolously.

4.4 Code antiquotation

In scenarios involving techniques like reflection it is quite common that code generated from a theory forms the basis for implementing a proof procedure in *SML*. To facilitate interfacing of generated code with system code, the code generator provides a *code* antiquotation:

datatype *form* = *T* | *F* | *And form form* | *Or form form*

```
ML {*
  fun eval_form @{code T} = true
    | eval_form @{code F} = false
    | eval_form (@{code And} (p, q)) =
        eval_form p andalso eval_form q
    | eval_form (@{code Or} (p, q)) =
        eval_form p orelse eval_form q;
*}
```

code takes as argument the name of a constant; after the whole *SML* is read, the necessary code is generated transparently and the corresponding constant names are inserted. This technique also allows to use pattern matching on constructors stemming from compiled *datatypes*.

For a less simplistic example, theory *Ferrack* is a good reference.

4.5 Imperative data structures

If you consider imperative data structures as inevitable for a specific application, you should consider *Imperative Functional Programming with Isabelle/HOL* ([3]); the framework described there is available in theory *Imperative-HOL*.

5 ML system 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.

5.1 Executable theory content: *Code*

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

Managing executable content

ML **Reference**

```
Code.add_eqn: thm -> theory -> theory
Code.del_eqn: thm -> theory -> theory
Code_Preproc.map_pre: (simpset -> simpset) -> theory -> theory
Code_Preproc.map_post: (simpset -> simpset) -> theory -> theory
Code_Preproc.add_functrans: string * (theory -> (thm * bool) list -> (thm * bool) list option
    -> theory -> theory
Code_Preproc.del_functrans: 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_eqn` *thm thy* adds function theorem *thm* to executable content.

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

`Code_Preproc.map_pre` *f thy* changes the preprocessor simpset.

`Code_Preproc.add_functrans` (*name*, *f*) *thy* adds function transformer *f* (named *name*) to executable content; *f* is a transformer of the code equations belonging to a certain function definition, depending on the current theory context. Returning *NONE* indicates that no transformation took place; otherwise, the whole process will be iterated with the new code equations.

`Code_Preproc.del_functrans` *name thy* removes function transformer 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.

5.2 Auxiliary

ML

Reference

```
Code.read_const: theory -> string -> string
```

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

5.3 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.

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 purge: *theory* $\rightarrow$ *string list option* $\rightarrow$ $T \rightarrow T$

T the type of data to store.

empty initial (empty) data.

purge thy consts propagates changes in executable content; *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: *theory* $\rightarrow$ T
change: *theory* $\rightarrow$ $(T \rightarrow T) \rightarrow T$
change-yield: *theory* $\rightarrow$ $(T \rightarrow {'a} * T) \rightarrow {'a} * T$

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!

References

[1] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In *Theorem Proving in Higher Order Logics*, pages 131–146, 2009.

[2] 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.

[3] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkk, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics: TPHOLs 2008*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[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*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

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