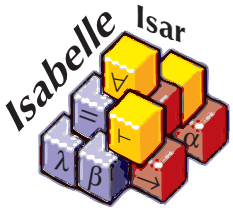

Code generation from Isabelle/HOL theories

Florian Haftmann with contributions from Lukas Bulwahn

27 August 2014

Abstract

This tutorial introduces the code generator facilities of Isabelle/HOL. They empower the user to turn HOL specifications into corresponding executable programs in the languages SML, OCaml, Haskell and Scala.

1 Introduction

This tutorial introduces the code generator facilities of *Isabelle/HOL*. It allows to turn (a certain class of) HOL specifications into corresponding executable code in the programming languages *SML* [8], *OCaml* [7], *Haskell* [11] and *Scala* [10].

To profit from this tutorial, some familiarity and experience with *HOL* [9] and its basic theories is assumed.

1.1 Code generation principle: shallow embedding

The key concept for understanding Isabelle's code generation is *shallow embedding*: logical entities like constants, types and classes are identified with corresponding entities in the target language. In particular, the carrier of a generated program's semantics are *equational theorems* from the logic. If we view a generated program as an implementation of a higher-order rewrite system, then every rewrite step performed by the program can be simulated in the logic, which guarantees partial correctness [6].

1.2 A quick start with the Isabelle/HOL toolbox

In a HOL theory, the **datatype** and **definition**/**primrec**/**fun** declarations form the core of a functional programming language. By default equational theorems stemming from those are used for generated code, therefore "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 : sig
  type 'a queue
  val empty : 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
  val enqueue : 'a -> 'a queue -> 'a queue
end = struct

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

fun fold f (x :: xs) s = fold f xs (f x s)
  | fold f [] s = s;

fun rev xs = fold (fn a => fn b => a :: b) xs [];

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 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*, *OCaml* and *Scala* 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 the corresponding code:

```
module Example(Queue, empty, dequeue, enqueue) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/),
  (**), (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++),
  (!!), Eq, error, id, return, not, fst, snd, map, filter, concat,
  concatMap, reverse, zip, null, takeWhile, dropWhile, all, any, Integer,
```

```
  negate, abs, divMod, String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

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) = reverse (v : va);
  } in (Just y, AQueue [] ys);

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

}
```

For more details about **export-code** see §7.

1.3 Type classes

Code can also be generated from type classes in a Haskell-like manner. For illustration here an example from abstract algebra:

class *semigroup* =
  fixes *mult* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\otimes$ 70)
  assumes *assoc*: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$

class *monoid* = *semigroup* +
  fixes *neutral* :: $'a$ (**1**)
  assumes *neutl*: $\mathbf{1} \otimes x = x$
    and *neutr*: $x \otimes \mathbf{1} = x$

instantiation *nat* :: *monoid*
begin

primrec *mult-nat* **where**
    $0 \otimes n = (0{::}nat)$
  | $Suc\ m \otimes n = n + m \otimes n$

definition *neutral-nat* **where**
  $\mathbf{1} = Suc\ 0$

lemma *add-mult-distrib*:

fixes n m q :: *nat*
shows $(n + m) \otimes q = n \otimes q + m \otimes q$
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* ⇒ *'a* ⇒ *'a* **where**
pow 0 *a* = **1**
| *pow* (*Suc n*) *a* = *a* ⊗ *pow n a*

This we use to define the discrete exponentiation function:

definition *bexp* :: *nat* ⇒ *nat* **where**
bexp n = *pow n* (*Suc* (*Suc* 0))

The corresponding code in Haskell uses that language's native classes:

```
module Example(Nat, bexp) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/),
  (**), (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++),
  (!!), Eq, error, id, return, not, fst, snd, map, filter, concat,
  concatMap, reverse, zip, null, takeWhile, dropWhile, all, any, Integer,
  negate, abs, divMod, String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

data Nat = Zero_nat | Suc Nat;

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

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

neutral_nat :: Nat;
neutral_nat = Suc Zero_nat;
```

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

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

instance Semigroup Nat where {
  mult = mult_nat;
};

instance Monoid Nat where {
  neutral = neutral_nat;
};

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

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 – the same example in *SML*:

```
structure Example : sig
  type nat
  val bexp : nat -> nat
end = struct

datatype nat = Zero_nat | Suc of nat;

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

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

val neutral_nat : nat = Suc Zero_nat;

type 'a semigroup = {mult : 'a -> 'a -> 'a};
val mult = #mult : 'a semigroup -> 'a -> 'a -> 'a;

type 'a monoid = {semigroup_monoid : 'a semigroup, neutral : 'a};
val semigroup_monoid = #semigroup_monoid : 'a monoid -> 'a semigroup;
val neutral = #neutral : 'a monoid -> 'a;

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

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

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

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.

1.4 How to continue from here

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, to understand situations where problems occur or to increase the scope of code generation beyond default, it is necessary to gain some understanding how the code generator actually works:

- The foundations of the code generator are described in §2.
- In particular §2.6 gives hints how to debug situations where code generation does not succeed as expected.
- The scope and quality of generated code can be increased dramatically by applying refinement techniques, which are introduced in §3.
- Inductive predicates can be turned executable using an extension of the code generator §4.
- If you want to utilize code generation to obtain fast evaluators e.g. for decision procedures, have a look at §6.
- You may want to skim over the more technical sections §5 and §7.
- The target language Scala [10] comes with some specialities discussed in §7.1.
- For exhaustive syntax diagrams etc. you should visit the Isabelle/Isar Reference Manual [12].

Happy proving, happy hacking!

2 Code generation foundations

2.1 Code generator architecture

The code generator is actually a framework consisting of different components which can be customised individually.

Conceptually all components operate on Isabelle's logic framework *Pure*. Practically, the object logic *HOL* provides the necessary facilities to make use of the code generator, mainly since it is an extension of *Pure*.

The constellation of the different components is visualized in the following picture.

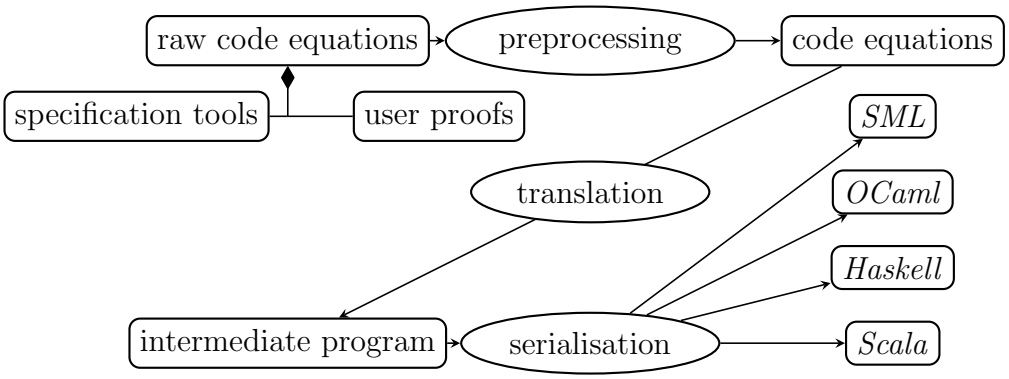

Figure 1: Code generator architecture

Central to code generation is the notion of *code equations*. 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).

- Starting point of code generation is a collection of (raw) code equations in a theory. It is not relevant where they stem from, but typically they were either produced by specification tools or proved explicitly by the user.

- These raw code equations can be subjected to theorem transformations. This *preprocessor* (see §2.2) can apply the full expressiveness of ML-based theorem transformations to code generation. The result of preprocessing 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 last two 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.

2.2 The pre- and postprocessor

Before selected function theorems are turned into abstract code, a chain of definitional transformation steps is carried out: *preprocessing*. The preprocessor consists of two components: a *simpset* and *function transformers*.

The preprocessor simpset has a disparate brother, the *postprocessor simpset*. In the theory-to-code scenario depicted in the picture above, it plays no role. But if generated code is used to evaluate expressions (cf. §6), the postprocessor simpset is applied to the resulting expression before this is turned back.

The pre- and postprocessor *simpsets* can apply 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.

Pre- and postprocessor can be setup to broker between expressions suitable for logical reasoning and expressions suitable for execution. As example, take list membership; logically is is just expressed as $x \in \mathit{set}\ \mathit{xs}$. But for execution the intermediate set is not desirable. Hence the following specification:

definition *member* :: $'a\ \mathit{list} \Rightarrow 'a \Rightarrow \mathit{bool}$
where
[*code-abbrev*]: $\mathit{member}\ \mathit{xs}\ x \longleftrightarrow x \in \mathit{set}\ \mathit{xs}$

The *code-abbrev attribute* declares its theorem a rewrite rule for the postprocessor and the symmetric of its theorem as rewrite rule for the preprocessor. Together, this has the effect that expressions $x \in \mathit{set}\ \mathit{xs}$ are replaced by $\mathit{member}\ \mathit{xs}\ x$ in generated code, but are turned back into $x \in \mathit{set}\ \mathit{xs}$ if generated code is used for evaluation.

Rewrite rules for pre- or postprocessor may be declared independently using *code-unfold* or *code-post* respectively.

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 used in theory *Code-Abstract-Nat* (see §5.3) uses this interface.
The current setup of the pre- and postprocessor may be inspected using the **print-codeproc** command. **code-thms** (see §2.3) provides a convenient mechanism to inspect the impact of a preprocessor setup on code equations.

2.3 Understanding code equations

As told in §1.1, the notion of code equations is vital to code generation. Indeed most problems which occur in practice can be resolved by an inspection of the underlying code equations.

It is possible to exchange the default code equations for constants by explicitly proving alternative ones:

```
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 *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 null xs then (Nothing, AQueue [] [])
    else dequeue (AQueue [] (reverse xs)));
```

You may note that the equality test $xs = []$ has been replaced by the predicate *List.null xs*. This is due to the default setup of the *preprocessor*.

This possibility to select arbitrary code equations is the key technique for program and datatype refinement (see §3).

Due to the preprocessor, there is the distinction of raw code equations (before preprocessing) and code equations (after preprocessing).

The first can be listed (among other data) using the **print-codesetup** command.

The code equations after preprocessing are already are blueprint of the generated program and can be inspected using the **code-thms** command:

code-thms *dequeue*

This prints a table with the code equations for *dequeue*, including *all* code equations those equations depend on recursively. These dependencies themselves can be visualized using the **code-deps** command.

2.4 Equality

Implementation of equality deserves some attention. Here an example function involving polymorphic equality:

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*)

During preprocessing, the membership test is rewritten, resulting in *List.member*, which itself performs an explicit equality check, as can be seen in the corresponding *SML* code:

```
structure Example : sig
  type 'a equal
  val collect_duplicates :
    'a equal -> 'a list -> 'a list -> 'a list -> 'a list
end = struct

type 'a equal = {equal : 'a -> 'a -> bool};
val equal = #equal : 'a equal -> 'a -> 'a -> bool;

fun eq A_ a b = equal A_ a b;

fun member A_ [] y = false
  | member A_ (x :: xs) y = eq A_ x y orelse member A_ xs y;

fun collect_duplicates A_ xs ys [] = xs
  | collect_duplicates A_ xs ys (z :: zs) =
    (if member A_ xs z
      then (if member A_ ys z 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 *equal* with a corresponding operation *equal-class.equal* such that *equal-class.equal* = *op*

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

2.5 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*) (*cases xs*, *simp-all split*: *list.split*)

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) = reverse 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 split*: *list.splits*)

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

Normally, if constants without any code equations occur in a program, the code generator complains (since in most cases this is indeed an error). But such constants can also be thought of as function definitions 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 the *code* attribute with *abort*:

declare [[*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 "Foundations.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 null xs then empty_queue
    else strict_dequeue (AQueue [] (reverse xs)));
```

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

2.6 If something goes utterly wrong

Under certain circumstances, the code generator fails to produce code entirely. To debug these, the following hints may prove helpful:

Check with a different target language. Sometimes the situation gets more clear if you switch to another target language; the code generated there might give some hints what prevents the code generator to produce code for the desired language.

Inspect code equations. Code equations are the central carrier of code generation. Most problems occurring while generating code can be traced to single equations which are printed as part of the error message. A closer inspection of those may offer the key for solving issues (cf. §2.3).

Inspect preprocessor setup. The preprocessor might transform code equations unexpectedly; to understand an inspection of its setup is necessary (cf. §2.2).

Generate exceptions. If the code generator complains about missing code equations, in can be helpful to implement the offending constants as exceptions (cf. §2.5); this allows at least for a formal generation of code, whose inspection may then give clues what is wrong.

Remove offending code equations. If code generation is prevented by just a single equation, this can be removed (cf. §2.3) to allow formal code generation, whose result in turn can be used to trace the problem. The most prominent case here are mismatches in type class signatures ("wellsortedness error").

3 Program and datatype refinement

Code generation by shallow embedding (cf. §1.1) allows to choose code equations and datatype constructors freely, given that some very basic syntactic properties are met; this flexibility opens up mechanisms for refinement which allow to extend the scope and quality of generated code dramatically.

3.1 Program refinement

Program refinement works by choosing appropriate code equations explicitly (cf. §2.3); as example, we use Fibonacci numbers:

$$\begin{aligned}&\textbf{fun } \mathit{fib} :: \mathit{nat} \Rightarrow \mathit{nat} \textbf{ where}\\ &\quad \mathit{fib}\ 0 = 0\\ &\mid \mathit{fib}\ (\mathit{Suc}\ 0) = \mathit{Suc}\ 0\\ &\mid \mathit{fib}\ (\mathit{Suc}\ (\mathit{Suc}\ n)) = \mathit{fib}\ n + \mathit{fib}\ (\mathit{Suc}\ n)\end{aligned}$$

The runtime of the corresponding code grows exponential due to two recursive calls:

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

A more efficient implementation would use dynamic programming, e.g. sharing of common intermediate results between recursive calls. This idea is expressed by an auxiliary operation which computes a Fibonacci number and its successor simultaneously:

definition *fib-step* :: *nat* ⇒ *nat* × *nat* **where**
fib-step n = (*fib* (*Suc n*), *fib n*)

This operation can be implemented by recursion using dynamic programming:

lemma [*code*]:
fib-step 0 = (*Suc* 0, 0)
fib-step (*Suc n*) = (*let* (*m*, *q*) = *fib-step n in* (*m* + *q*, *m*))
by (*simp-all add*: *fib-step-def*)

What remains is to implement *fib* by *fib-step* as follows:

lemma [*code*]:
fib 0 = 0
fib (*Suc n*) = *fst* (*fib-step n*)
by (*simp-all add*: *fib-step-def*)

The resulting code shows only linear growth of runtime:

```
fib_step :: Nat -> (Nat, Nat);
fib_step (Suc n) = let {
                     (m, q) = fib_step n;
                   } in (plus_nat m q, m);
fib_step Zero_nat = (Suc Zero_nat, Zero_nat);

fib :: Nat -> Nat;
fib (Suc n) = fst (fib_step n);
fib Zero_nat = Zero_nat;
```

3.2 Datatype refinement

Selecting specific code equations *and* datatype constructors leads to datatype refinement. As an example, we will develop an alternative representation of the queue example given in §1.2. The amortised representation is convenient for generating code but exposes its "implementation" details, which may be cumbersome when proving theorems about it. Therefore, here is 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.

The prerequisite for datatype constructors is only syntactical: a constructor must be 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; then κ is its corresponding datatype. The HOL datatype package by default registers any new datatype with its constructors, but this may be changed using **code-datatype**; the currently chosen constructors can be inspected using the **print-codesetup** command.

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 [] []
  by (simp add: AQueue-def empty-def)

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

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)
  by (simp-all add: AQueue-def)
```

It is good style, although no absolute requirement, to provide code equations for the original artefacts of the implemented type, if possible; in our case, these are the datatype constructor *Queue* and the case combinator *case-queue*:

lemma *Queue-AQueue* [*code*]:
Queue = *AQueue* []
by (*simp add*: *AQueue-def fun-eq-iff*)

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

The resulting code looks as expected:

```
structure Example : sig
  type 'a queue
  val empty : 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
  val enqueue : 'a -> 'a queue -> 'a queue
  val queue : 'a list -> 'a queue
  val case_queue : ('a list -> 'b) -> 'a queue -> 'b
end = struct

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

fun fold f (x :: xs) s = fold f xs (f x s)
  | fold f [] s = s;

fun rev xs = fold (fn a => fn b => a :: b) xs [];

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

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);

fun queue x = AQueue ([], x);

fun case_queue f (AQueue (xs, ys)) = f (ys @ rev xs);

end; (*struct Example*)
```

The same techniques can also be applied to types which are not specified as datatypes, e.g. type *int* is originally specified as quotient type by means of **typedef**, but for code generation constants allowing construction of binary numeral values are used as constructors for *int*.

This approach however fails if the representation of a type demands invariants; this issue is discussed in the next section.

3.3 Datatype refinement involving invariants

Datatype representation involving invariants require a dedicated setup for the type and its primitive operations. As a running example, we implement a type *'a dlist* of list consisting of distinct elements.

The first step is to decide on which representation the abstract type (in our example *'a dlist*) should be implemented. Here we choose *'a list*. Then a conversion from the concrete type to the abstract type must be specified, here:

$$Dlist :: 'a\ list \Rightarrow 'a\ dlist$$

Next follows the specification of a suitable *projection*, i.e. a conversion from abstract to concrete type:

$$list\text{-}of\text{-}dlist :: 'a\ dlist \Rightarrow 'a\ list$$

This projection must be specified such that the following *abstract datatype certificate* can be proven:

lemma [*code abstype*]:
  $Dlist\ (list\text{-}of\text{-}dlist\ dxs) = dxs$
  by (*fact Dlist-list-of-dlist*)

Note that so far the invariant on representations ($distinct :: 'a\ list \Rightarrow bool$) has never been mentioned explicitly: the invariant is only referred to implicitly: all values in set $\{xs.\ list\text{-}of\text{-}dlist\ (Dlist\ xs) = xs\}$ are invariant, and in our example this is exactly $\{xs.\ distinct\ xs\}$.

The primitive operations on *'a dlist* are specified indirectly using the projection *list-of-dlist*. For the empty *dlist*, *Dlist.empty*, we finally want the code equation

$$Dlist.empty = Dlist\ []$$

This we have to prove indirectly as follows:

lemma [*code*]:
list-of-dlist Dlist.empty = []
by (*fact list-of-dlist-empty*)

This equation logically encodes both the desired code equation and that the expression *Dlist* is applied to obeys the implicit invariant. Equations for insertion and removal are similar:

lemma [*code*]:
list-of-dlist (*Dlist.insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
by (*fact list-of-dlist-insert*)

lemma [*code*]:
list-of-dlist (*Dlist.remove x dxs*) = *remove*1 *x* (*list-of-dlist dxs*)
by (*fact list-of-dlist-remove*)

Then the corresponding code is as follows:

```
module Example(Dlist, empty, list_of_dlist, inserta, remove) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/),
  (**), (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++),
  (!!), Eq, error, id, return, not, fst, snd, map, filter, concat,
  concatMap, reverse, zip, null, takeWhile, dropWhile, all, any, Integer,
  negate, abs, divMod, String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

newtype Dlist a = Dlist [a];

empty :: forall a. Dlist a;
empty = Dlist [];

member :: forall a. (Eq a) => [a] -> a -> Bool;
member [] y = False;
member (x : xs) y = x == y || member xs y;

insert :: forall a. (Eq a) => a -> [a] -> [a];
insert x xs = (if member xs x then xs else x : xs);

list_of_dlist :: forall a. Dlist a -> [a];
list_of_dlist (Dlist x) = x;

inserta :: forall a. (Eq a) => a -> Dlist a -> Dlist a;
inserta x dxs = Dlist (insert x (list_of_dlist dxs));

remove1 :: forall a. (Eq a) => a -> [a] -> [a];
remove1 x [] = [];
remove1 x (y : xs) = (if x == y then xs else y : remove1 x xs);

remove :: forall a. (Eq a) => a -> Dlist a -> Dlist a;
remove x dxs = Dlist (remove1 x (list_of_dlist dxs));

}
```

See further [5] for the meta theory of datatype refinement involving invariants.

Typical data structures implemented by representations involving invariants are available in the library, theory *Mapping* specifies key-value-mappings (type $('a, 'b)$ *mapping*); these can be implemented by red-black-trees (theory *RBT*).

4 Inductive Predicates

The *predicate compiler* is an extension of the code generator which turns inductive specifications into equational ones, from which in turn executable code can be generated. The mechanisms of this compiler are described in detail in [3].

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.

4.1 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*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as concat*,
$o \Rightarrow o \Rightarrow i \Rightarrow$ *bool as split*,
$i \Rightarrow o \Rightarrow i \Rightarrow$ *bool as suffix*) *append* **.**

4.2 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 $\Longrightarrow$ *tranclp r a b*
tranclp r a b $\Longrightarrow$ *r b c* $\Longrightarrow$ *tranclp 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 $\Longrightarrow$ *tranclp r a b*
r a b $\Longrightarrow$ *tranclp r b c* $\Longrightarrow$ *tranclp 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 tranclp.prems*] **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:

lexordp r ?xs ?ys $\longleftrightarrow$
$(\exists\, a\ v.\ ?ys = ?xs\ @\ a\ \#\ v \vee$
$\quad (\exists\, u\ a\ b\ v\ w.\ r\ a\ b \wedge ?xs = u\ @\ a\ \#\ v \wedge ?ys = u\ @\ b\ \#\ w))$

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

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

4.3 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 *succ* :: *nat* ⇒ *nat* ⇒ *bool* **where**
  succ 0 (*Suc* 0)
| *succ x y* ⟹ *succ* (*Suc x*) (*Suc y*)

code-pred *succ* **.**

For this, the predicate compiler can infer modes $o \Rightarrow o \Rightarrow bool$, $i \Rightarrow o \Rightarrow bool$, $o \Rightarrow i \Rightarrow bool$ and $i \Rightarrow i \Rightarrow bool$. The invocation of **values** $\{n.\ tranclp\ succ\ 10\ n\}$ loops, as multiple modes for the predicate *succ* are possible and here the first mode $o \Rightarrow o \Rightarrow bool$ 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*: $i \Rightarrow o \Rightarrow bool$] 1 $\{n.\ tranclp\ succ\ 10\ n\}$
values [*mode*: $o \Rightarrow i \Rightarrow bool$] 1 $\{n.\ tranclp\ succ\ n\ 10\}$

4.4 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*.

4.5 Further Examples

Further examples for compiling inductive predicates can be found in `~~/src/HOL/Predicate_Compile_Examples/Examples.thy`. There are also some examples in the Archive of Formal Proofs, notably in the *POPLmark−deBruijn* and the *FeatherweightJava* sessions.

5 Adaptation to target languages

5.1 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 separately.
- Application is extremely tedious since there is no abstraction which would allow for a static check, making it easy to produce garbage.
- Subtle errors can be introduced unconsciously.

However, even if you ought refrain from setting up adaptation yourself, already *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.

5.2 The adaptation principle

Figure 2 illustrates what "adaptation" is conceptually supposed to be:

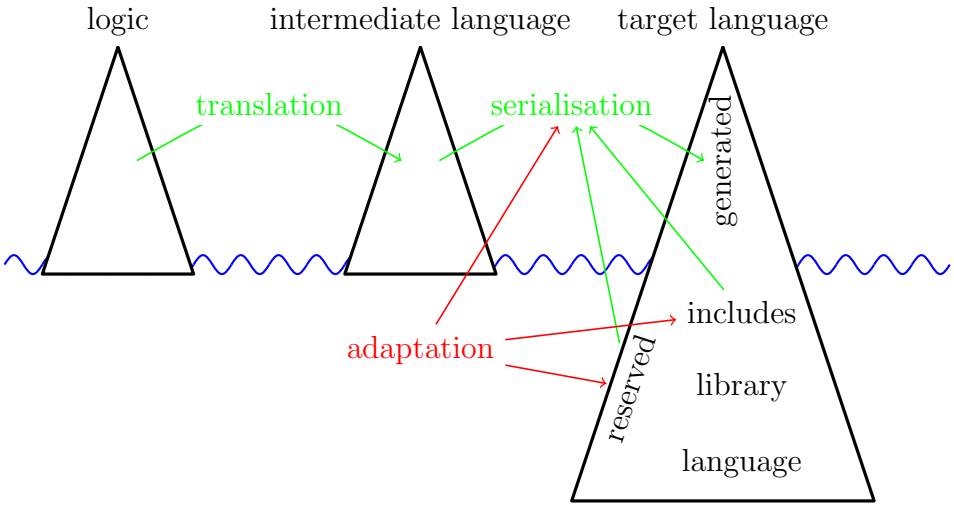

Figure 2: The adaptation principle

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 §5.4)
- 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 §5.6); 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.

5.3 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 in `~~/src/HOL/Library`; beside being useful in applications, they may serve as a tutorial for customising the code generator setup (see below §5.4).

Code-Numeral provides additional numeric types *integer* and *natural* isomorphic to types *int* and *nat* respectively. Type *integer* is mapped to target-language built-in integers; *natural* is implemented as abstract type over *integer*. Useful for code setups which involve e.g. indexing of target-language arrays. Part of *HOL−Main*.

Code-Target-Int implements type *int* by *integer* and thus by target-language built-in integers.

Code-Binary-Nat implements type *nat* using a binary rather than a linear representation, which yields a considerable speedup for computations. Pattern matching with 0 / *Suc* is eliminated by a preprocessor.

Code-Target-Nat implements type *nat* by *integer* and thus by target-language built-in integers. Pattern matching with 0 / *Suc* is eliminated by a preprocessor.

Code-Target-Numeral is a convenience theory containing both *Code-Target-Nat* and *Code-Target-Int*.

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

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. Part of *HOL−Main*.

IArray provides a type *'a iarray* isomorphic to lists but implemented by (effectively immutable) arrays *in SML only*.

5.4 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 : sig
  type nat
  type boola
  val in_interval : nat * nat -> nat -> boola
end = struct

datatype nat = Zero_nat | Suc of nat;

datatype boola = True | False;

fun conj p True = p
  | conj p False = False
  | conj True p = p
  | conj False p = False;

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

fun in_interval (k, l) n = conj (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_printing
  type_constructor bool ⇀ (SML) "bool"
| constant True ⇀ (SML) "true"
| constant False ⇀ (SML) "false"
| constant HOL.conj ⇀ (SML) "_ andalso _"
```

The **code-printing** command takes a series of symbols (contants, type constructor, ...) together with target-specific 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. Each "_" in a serialisation expression is treated as a placeholder for the constant's or the type constructor's arguments.

```
structure Example : sig
  type nat
  val in_interval : nat * nat -> nat -> bool
end = struct

datatype nat = Zero_nat | Suc of nat;

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

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_printing
  constant HOL.conj ⇀ (SML) infixl 1 "andalso"
```

```
structure Example : sig
  type nat
  val in_interval : nat * nat -> nat -> bool
end = struct

datatype nat = Zero_nat | Suc of nat;

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

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_printing
  type_constructor prod ⇀ (SML) infix 2 "*"
| constant 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 detail 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.

5.5 *Haskell* serialisation

For convenience, the default *HOL* setup for *Haskell* maps the *equal* class to its counterpart in *Haskell*, giving custom serialisations for the class *equal* and its operation *HOL.equal*.

```
code_printing
  type_class equal ⇀ (Haskell) "Eq"
| constant HOL.equal ⇀ (Haskell) infixl 4 "=="
```

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

typedecl *bar*

instantiation *bar* :: *equal*

begin

definition *HOL.equal* (*x*::*bar*) *y* ⟷ *x* = *y*

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

end

```
code_printing
  type_constructor 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:

```
code_printing
  class_instance bar :: "HOL.equal" ⇀ (Haskell) -
```

5.6 Enhancing the target language context

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

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

code_reserved Haskell Errno
```

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

6 Evaluation

Recalling §1.1, code generation turns a system of equations into a program with the *same* equational semantics. As a consequence, this program can be used as a *rewrite engine* for terms: rewriting a term t using a program to a term t' yields the theorems $t \equiv t'$. This application of code generation in the following is referred to as *evaluation*.

6.1 Evaluation techniques

The existing infrastructure provides a rich palette of evaluation techniques, each comprising different aspects:

Expressiveness. Depending on how good symbolic computation is supported, the class of terms which can be evaluated may be bigger or smaller.

Efficiency. The more machine-near the technique, the faster it is.

Trustability. Techniques which a huge (and also probably more configurable infrastructure) are more fragile and less trustable.

The simplifier (*simp*)

The simplest way for evaluation is just using the simplifier with the original code equations of the underlying program. This gives fully symbolic evaluation and highest trustablity, with the usual performance of the simplifier. Note that for operations on abstract datatypes (cf. §3.3), the original theorems as given by the users are used, not the modified ones.

Normalization by evaluation (*nbe*)

Normalization by evaluation [1] provides a comparably fast partially symbolic evaluation which permits also normalization of functions and uninterpreted symbols; the stack of code to be trusted is considerable.

Evaluation in ML (*code*)

Highest performance can be achieved by evaluation in ML, at the cost of being restricted to ground results and a layered stack of code to be trusted, including code generator configurations by the user.

Evaluation is carried out in a target language *Eval* which inherits from *SML* but for convenience uses parts of the Isabelle runtime environment. The soundness of computation carried out there depends crucially on the correctness of the code generator setup; this is one of the reasons why you should not use adaptation (see §5) frivolously.

6.2 Aspects of evaluation

Each of the techniques can be combined with different aspects. The most important distinction is between dynamic and static evaluation. Dynamic evaluation takes the code generator configuration "as it is" at the point where evaluation is issued. Best example is the **value** command which allows ad-hoc evaluation of terms:

value 42 / (12 :: *rat*)

value tries first to evaluate using ML, falling back to normalization by evaluation if this fails.

To employ dynamic evaluation in the document generation, there is also a *value* antiquotation with the same evaluation techniques as **value**.

Static evaluation freezes the code generator configuration at a certain point and uses this context whenever evaluation is issued later on. This is particularly appropriate for proof procedures which use evaluation, since then the behaviour of evaluation is not changed or even compromised later on by actions of the user.

As a technical complication, terms after evaluation in ML must be turned into Isabelle's internal term representation again. Since this is also configurable, it is never fully trusted. For this reason, evaluation in ML comes with further aspects:

Plain evaluation. A term is normalized using the provided term reconstruction from ML to Isabelle; for applications which do not need to be fully trusted.

Property conversion. Evaluates propositions; since these are monomorphic, the term reconstruction is fixed once and for all and therefore trustable.

Conversion. Evaluates an arbitrary term t first by plain evaluation and certifies the result t' by checking the equation $t \equiv t'$ using property conversion.

The picture is further complicated by the roles of exceptions. Here three cases have to be distinguished:

- Evaluation of t terminates with a result t'.
- Evaluation of t terminates which en exception indicating a pattern match failure or a non-implemented function. As sketched in §2.5, this can be interpreted as partiality.

- Evaluation raises any other kind of exception.

For conversions, the first case yields the equation $t = t'$, the second defaults to reflexivity $t = t$. Exceptions of the third kind are propagated to the user.

By default return values of plain evaluation are optional, yielding *SOME* t' in the first case, *NONE* in the second, and propagating the exception in the third case. A strict variant of plain evaluation either yields t' or propagates any exception, a liberal variant captures any exception in a result of type *Exn.result*.

For property conversion (which coincides with conversion except for evaluation in ML), methods are provided which solve a given goal by evaluation.

6.3 Schematic overview

		simp	*nbe*	*code*
dynamic	plain evaluation			`Code_Evaluation.dynamic_value`
	evaluation method	*code-simp*	*normalization*	*eval*
	property conversion			`Code_Runtime.dynamic_holds_conv`
	conversion	`Code_Simp.dynamic_conv`	`Nbe.dynamic_conv`	`Code_Evaluation.dynamic_conv`
static	plain evaluation			`Code_Evaluation.static_value`
	property conversion			`Code_Runtime.static_holds_conv`
	conversion	`Code_Simp.static_conv`	`Nbe.static_conv`	`Code_Evaluation.static_conv`

6.4 Preprocessing HOL terms into evaluable shape

When integration decision procedures developed inside HOL into HOL itself, it is necessary to somehow get from the Isabelle/ML representation to the representation used by the decision procedure itself ("reification"). One option is to hardcode it using code antiquotations (see §6.5). Another option is to use pre-existing infrastructure in HOL: `Reification.conv` and `Reification.tac`

An simplistic example:

datatype *form-ord* = *T* | *F* | *Less nat nat*
  | *And form-ord form-ord* | *Or form-ord form-ord* | *Neg form-ord*

primrec *interp* :: *form-ord* ⇒ *'a*::*order list* ⇒ *bool*
where
  interp T vs ⟷ *True*
| *interp F vs* ⟷ *False*
| *interp* (*Less i j*) *vs* ⟷ *vs* ! *i* < *vs* ! *j*
| *interp* (*And f*1 *f*2) *vs* ⟷ *interp f*1 *vs* ∧ *interp f*2 *vs*

| *interp* (*Or f*1 *f*2) *vs* ⟷ *interp f*1 *vs* ∨ *interp f*2 *vs*
| *interp* (*Neg f*) *vs* ⟷ ¬ *interp f vs*

The datatype *form-ord* represents formulae whose semantics is given by *interp*. Note that values are represented by variable indices (*nat*) whose concrete values are given in list *vs*.

ML ⟨⟨ *val thm* =
Reification.conv @{*context*} @{*thms interp.simps*} @{*cterm* $x < y \wedge x < z$} ⟩⟩

By virtue of *interp.simps*, `Reification.conv` provides a conversion which, for this concrete example, yields $x < y \wedge x < z \equiv$ *interp* (*And* (*Less* (*Suc* 0) (*Suc* (*Suc* 0))) (*Less* (*Suc* 0) 0)) [*z*, *x*, *y*]. Note that the argument to *interp* does not contain any free variables and can this be evaluated using evaluation.

A less meager example can be found in the AFP, session *Regular−Sets*, theory *Regexp-Method*.

6.5 Intimate connection between logic and system runtime

The toolbox of static evaluation conversions forms a reasonable base to interweave generated code and system tools. However in some situations more direct interaction is desirable.

Static embedding of generated code into system runtime – the *code* antiquotation

The *code* antiquotation allows to include constants from generated code directly into ML system code, as in the following toy example:

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 ML 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. Note that the *code* antiquotation may not refer to constants which carry adaptations; here you have to refer to the corresponding adapted code directly.

For a less simplistic example, theory *Approximation* in the *Decision-Procs* session is a good reference.

Static embedding of generated code into system runtime – *code-reflect*

The *code* antiquoation is lightweight, but the generated code is only accessible while the ML section is processed. Sometimes this is not appropriate, especially if the generated code contains datatype declarations which are shared with other parts of the system. In these cases, **code-reflect** can be used:

```
code-reflect Sum-Type
  datatypes sum = Inl | Inr
  functions Sum-Type.sum.projl Sum-Type.sum.projr
```

code-reflect takes a structure name and references to datatypes and functions; for these code is compiled into the named ML structure and the *Eval* target is modified in a way that future code generation will reference these precompiled versions of the given datatypes and functions. This also allows to refer to the referenced datatypes and functions from arbitrary ML code as well.

A typical example for **code-reflect** can be found in the *Predicate* theory.

Separate compilation – *code-reflect*

For technical reasons it is sometimes necessary to separate generation and compilation of code which is supposed to be used in the system runtime. For this **code-reflect** with an optional *file* argument can be used:

```
code-reflect Rat
  datatypes rat = Frct
  functions Fract
    (plus :: rat ⇒ rat ⇒ rat) (minus :: rat ⇒ rat ⇒ rat)
    (times :: rat ⇒ rat ⇒ rat) (divide :: rat ⇒ rat ⇒ rat)
  file examples/rat.ML
```

This merely generates the referenced code to the given file which can be included into the system runtime later on.

7 Further issues

7.1 Specialities of the *Scala* target language

Scala deviates from languages of the ML family in a couple of aspects; those which affect code generation mainly have to do with *Scala*'s type system:

- *Scala* prefers tupled syntax over curried syntax.
- *Scala* sacrifices Hindely-Milner type inference for a much more rich type system with subtyping etc. For this reason type arguments sometimes have to be given explicitly in square brackets (mimicking System F syntax).
- In contrast to *Haskell* where most specialities of the type system are implemented using *type classes*, *Scala* provides a sophisticated system of *implicit arguments*.

Concerning currying, the *Scala* serializer counts arguments in code equations to determine how many arguments shall be tupled; remaining arguments and abstractions in terms rather than function definitions are always curried.

The second aspect affects user-defined adaptations with **code-printing**. For regular terms, the *Scala* serializer prints all type arguments explicitly. For user-defined term adaptations this is only possible for adaptations which take no arguments: here the type arguments are just appended. Otherwise they are ignored; hence user-defined adaptations for polymorphic constants have to be designed very carefully to avoid ambiguity.

Isabelle's type classes are mapped onto *Scala* implicits; in cases with diamonds in the subclass hierarchy this can lead to ambiguities in the generated code:

class *class*1 =
  fixes *foo* :: $'a \Rightarrow 'a$

class *class*2 = *class*1

class *class*3 = *class*1

Here both *class*2 and *class*3 inherit from *class*1, forming the upper part of a diamond.

definition *bar* :: $'a$ :: {*class*2, *class*3} ⇒ $'a$ **where**
bar = *foo*

This yields the following code:

```
object Example {

trait class1[A] {
  val ‘Example.foo‘: A => A
}
def foo[A](a: A)(implicit A: class1[A]): A = A.‘Example.foo‘(a)

trait class2[A] extends class1[A] {
}

trait class3[A] extends class1[A] {
}

def bar[A : class2 : class3]: A => A = (a: A) => foo[A](a)

} /* object Example */
```

This code is rejected by the *Scala* compiler: in the definition of *bar*, it is not clear from where to derive the implicit argument for *foo*.

The solution to the problem is to close the diamond by a further class with inherits from both *class*2 and *class*3:

class *class*4 = *class*2 + *class*3

Then the offending code equation can be restricted to *class*4:

lemma [*code*]:
(*bar* :: $'a$::*class*4 ⇒ $'a$) = *foo*
by (*simp only*: *bar-def*)

with the following code:

```
object Example {

trait class1[A] {
  val ‘Example.foo‘: A => A
}
def foo[A](a: A)(implicit A: class1[A]): A = A.‘Example.foo‘(a)

trait class2[A] extends class1[A] {
}
```

```
trait class3[A] extends class1[A] {
}

trait class4[A] extends class2[A] with class3[A] {
}

def bar[A : class4]: A => A = (a: A) => foo[A](a)

} /* object Example */
```

which exposes no ambiguity.

Since the preprocessor (cf. §2.2) propagates sort constraints through a system of code equations, it is usually not very difficult to identify the set of code equations which actually needs more restricted sort constraints.

7.2 Modules namespace

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-identifier
  code-module $A \rightharpoonup (SML)\ ABC$
| **code-module** $B \rightharpoonup (SML)\ ABC$
| **code-module** $C \rightharpoonup (SML)\ ABC$

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

7.3 Locales and interpretation

A technical issue comes to surface when generating code from specifications stemming from locale interpretation.

Let us assume a locale specifying a power operation on arbitrary types:

locale *power* =
  fixes *power* :: $'a \Rightarrow 'b \Rightarrow 'b$
  assumes *power-commute*: $power\ x \circ power\ y = power\ y \circ power\ x$
begin

Inside that locale we can lift *power* to exponent lists by means of specification relative to that locale:

```
primrec powers :: 'a list ⇒ 'b ⇒ 'b where
  powers [] = id
| powers (x # xs) = power x ∘ powers xs

lemma powers-append:
  powers (xs @ ys) = powers xs ∘ powers ys
  by (induct xs) simp-all

lemma powers-power:
  powers xs ∘ power x = power x ∘ powers xs
  by (induct xs)
    (simp-all del: o-apply id-apply add: comp-assoc,
      simp del: o-apply add: o-assoc power-commute)

lemma powers-rev:
  powers (rev xs) = powers xs
    by (induct xs) (simp-all add: powers-append powers-power)

end
```

After an interpretation of this locale (say, **interpretation** *fun-power*: *power* (λ*n* (*f* :: 'a ⇒ 'a). *f* ^^ *n*)), one would expect to have a constant *fun-power.powers* :: *nat list* ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a for which code can be generated. But this not the case: internally, the term *fun-power.powers* is an abbreviation for the foundational term *power.powers* (λ*n* (*f* :: 'a ⇒ 'a). *f* ^^ *n*) (see [2] for the details behind).

Fortunately, with minor effort the desired behaviour can be achieved. First, a dedicated definition of the constant on which the local *powers* after interpretation is supposed to be mapped on:

```
definition funpows :: nat list ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a where
  [code del]: funpows = power.powers (λn f. f ^^ n)
```

In general, the pattern is $c = t$ where c is the name of the future constant and t the foundational term corresponding to the local constant after interpretation.

The interpretation itself is enriched with an equation $t = c$:

```
interpretation fun-power: power λn (f :: 'a ⇒ 'a). f ^^ n where
```

power.powers $(\lambda n\ f.\ f\ \hat{}\hat{}\ n) = funpows$
by *unfold-locales*
(*simp-all add*: *fun-eq-iff funpow-mult mult.commute funpows-def*)

This additional equation is trivially proved by the definition itself.

After this setup procedure, code generation can continue as usual:

```
funpow :: forall a. Nat -> (a -> a) -> a -> a;
funpow Zero_nat f = id;
funpow (Suc n) f = f . funpow n f;

funpows :: forall a. [Nat] -> (a -> a) -> a -> a;
funpows [] = id;
funpows (x : xs) = funpow x . funpows xs;
```

7.4 Parallel computation

Theory *Parallel* in `~~/src/HOL/Library` contains operations to exploit parallelism inside the Isabelle/ML runtime engine.

7.5 Imperative data structures

If you consider imperative data structures as inevitable for a specific application, you should consider *Imperative Functional Programming with Isabelle/HOL* [4]; the framework described there is available in session *Imperative-HOL*, together with a short primer document.

7.6 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 fundamental ML interfaces.

Managing executable content

ML

Reference

```
Code.read_const: theory -> string -> string
Code.add_eqn: thm -> theory -> theory
Code.del_eqn: thm -> theory -> theory
Code_Preproc.map_pre: (Proof.context -> Proof.context) -> theory -> theory
Code_Preproc.map_post: (Proof.context -> Proof.context) -> theory -> theory
Code_Preproc.add_functrans:
    string * (Proof.context -> (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_type: theory -> string
    -> ((string * sort) list * (string * ((string * sort) list * typ list)) list) * bool
Code.get_type_of_constr_or_abstr: theory -> string -> (string * bool) option
```

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

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

Data depending on the theory's executable content

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.

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 `Code_Data`; on instantiation of this functor, the following types and operations are required:

type T
val empty: T

T the type of data to store.

empty initial (empty) data.

An instance of `Code_Data` provides the following interface:

change: *theory* $\to (T \to T) \to T$
change-yield: *theory* $\to (T \to 'a * T) \to 'a * T$

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

change-yield update with side result.

References

[1] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs '08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2008.

[2] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* http://isabelle.in.tum.de/doc/locales.pdf.

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

[4] 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.

[5] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. Data refinement in isabelle/hol. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2013.

[6] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

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

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

[9] 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.

[10] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[11] 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/.

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