

Code generation from Isabelle/HOL theories

Florian Haftmann
with contributions by Lukas Bulwahn and Tobias Nipkow

15 November 2026

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* [9], *OCaml* [8], *Haskell* [11] and *Scala* [5].

To profit from this tutorial, some familiarity and experience with Isabelle/HOL [10] 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 [7].

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*

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 [] s = s
  | fold f (x :: xs) s = fold f xs (f x 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 multiple 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**.

Output is written to a logical file-system within the theory context, with the theory name and "`code`" as overall prefix. There is also a formal session export using the same name: the result may be explored in the Isabelle/jEdit Prover IDE using the file-browser on the URL "`isabelle-export:`".

The file name is determined by the target language together with an optional **file_prefix** (the default is "`export`" with a consecutive number within

the current theory). For *SML*, *OCaml* and *Scala*, the file prefix becomes a plain file with extension (e.g. "`.ML`" for SML). For *Haskell* the file prefix becomes a directory that is populated with a separate file for each module (with extension "`.hs`").

Consider the following example:

export_code *empty dequeue enqueue* **in** *Haskell*
module_name *Example* **file_prefix** *example*

This is the corresponding code:

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

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 Data.Bits ((.&.), (.|.), (.^.));
import qualified Prelude;
import qualified Data.Bits;

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) []) = (case reverse (v : va) of {
                                  y : ys -> (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 §9.

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 ⇒ '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 ⊗ n ⊗ q = m ⊗ (n ⊗ q)
    by (induct m) (simp_all add: add_mult_distrib)
  show 1 ⊗ n = n
    by (simp add: neutral_nat_def)
  show m ⊗ 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 $bexp :: nat \Rightarrow nat$ **where**
$bexp\ n = pow\ n\ (Suc\ (Suc\ 0))$

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

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

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 Data.Bits ((.&.), (.|.), (.^.));
import qualified Prelude;
import qualified Data.Bits;

data Nat = Zero_nat | Suc Nat;

plus_nat :: Nat -> Nat -> Nat;
plus_nat Zero_nat n = n;
plus_nat (Suc m) n = plus_nat m (Suc 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 Zero_nat n = n
  | plus_nat (Suc m) n = plus_nat m (Suc 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.
- How to define partial functions such that code can be generated is explained in §4.
- Inductive predicates can be turned executable using an extension of the code generator §5.
- 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 §8 and §9.
- The target language Scala [5] comes with some specialities discussed in §9.3.
- For exhaustive syntax diagrams etc. you should visit the Isabelle/Isar Reference Manual [14].

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 Isabelle/Pure.

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

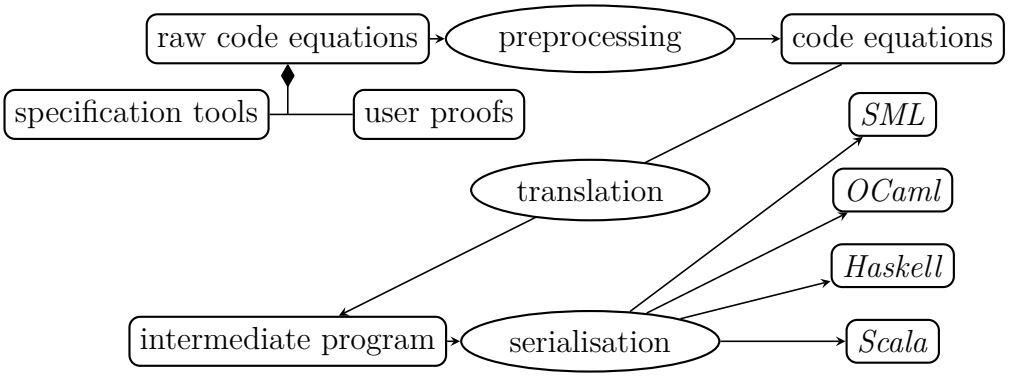

Figure 1: Code generator architecture

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 transfer between expressions suitable for logical reasoning and expressions suitable for execution. As example, take list membership; logically it is expressed as $x \in \mathit{set}\ \mathit{xs}$. But for execution the intermediate set is not desirable. Hence the following specification:

definition *member* :: *'a list* ⇒ *'a* ⇒ *bool*
where
[*code_abbrev*]: *member xs x* ⟷ $x \in \mathit{set}\ \mathit{xs}$

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 *member 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 §8.3) 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. Attribute *code_preproc_trace* allows for low-level tracing:

declare [[*code_preproc_trace*]]

declare [[*code_preproc_trace only*: *distinct remdups*]]

declare [[*code_preproc_trace off*]]

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

```
dequeue (AQueue xs (y : ys)) = (Just y, AQueue xs ys);
```

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* = (=). 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

Explicit partiality is caused by missing patterns (in contrast to partiality caused by nontermination, which is covered in Section 4). Here is an 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 []) =
    (case rev xs of y # ys ⇒ (y, AQueue [] ys))
  strict_dequeue (AQueue xs (y # ys)) = (y, AQueue xs 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 []) = (case reverse xs of {
                                   y : ys -> (y, AQueue [] ys);
                                 });
```

```
strict_dequeue (AQueue xs (y : ys)) = (y, AQueue xs ys);
```

In some cases it is desirable to state 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. An attempt to generate code for *strict_dequeue'* would make the code generator complain that *empty_queue* has no associated code equations. In most situations unimplemented constants indeed indicated a broken program; however 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 []) =
  (if null xs then empty_queue
    else strict_dequeue (AQueue [] (reverse xs)));
```

```
strict_dequeue (AQueue xs (y : ys)) = (y, AQueue xs ys);
```

This feature however is rarely needed in practice. Note that the HOL default setup already includes

declare [[*code abort*: *undefined*]]

– hence *undefined* can always be used in such situations.

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:

fun *fib* :: *nat* ⇒ *nat* **where**
  fib 0 = 0
| *fib* (*Suc* 0) = *Suc* 0
| *fib* (*Suc* (*Suc* *n*)) = *fib* *n* + *fib* (*Suc* *n*)

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 Zero_nat = (Suc Zero_nat, Zero_nat);
```

```
fib_step (Suc n) = (case fib_step n of {
                      (m, q) -> (plus_nat m q, m);
```

```
                    });

fib :: Nat -> Nat;



fib Zero_nat = Zero_nat;



fib (Suc n) = fst (fib_step n);
```

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

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 lists consisting of distinct elements.

The specification of *'a dlist* itself can be found in theory *HOL−Library.Dlist*.

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

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

list_of_dlist

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

lemma [*code abstype*]:
Dlist (*list_of_dlist dxs*) = *dxs*
by (*fact Dlist_list_of_dlist*)

Note that so far the invariant on representations (*distinct*) has never been mentioned explicitly: the invariant is only referred to implicitly: all values in set {*xs*. *list_of_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:

```
structure Example : sig
  type 'a equal
  type 'a dlist
  val empty : 'a dlist
  val list_of_dlist : 'a dlist -> 'a list
  val inserta : 'a equal -> 'a -> 'a dlist -> 'a dlist
  val remove : 'a equal -> 'a -> 'a dlist -> 'a dlist
end = struct

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

datatype 'a dlist = Dlist of 'a list;

fun eq A_ a b = equal A_ a b;

val empty : 'a dlist = Dlist [];

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

fun insert A_ x xs = (if member A_ xs x then xs else x :: xs);

fun list_of_dlist (Dlist x) = x;

fun inserta A_ x dxs = Dlist (insert A_ x (list_of_dlist dxs));

fun remove1 A_ x [] = []
  | remove1 A_ x (y :: xs) =
    (if eq A_ x y then xs else y :: remove1 A_ x xs);

fun remove A_ x dxs = Dlist (remove1 A_ x (list_of_dlist dxs));

end; (*struct Example*)
```

To reduce manual work for datatype refinement, **lift_definition** is a valuable tool. See the corresponding section in [14].

See further [6] for the meta theory of datatype refinement involving invariants.

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

4 Partial Functions

We demonstrate three approaches to defining executable partial recursive functions, i.e. functions that do not terminate for all inputs. The main points are the definitions of the functions and the inductive proofs about them.

Our concrete example represents a typical termination problem: following a data structure that may contain cycles. We want to follow a mapping from *nat* to *nat* to the end (until we leave its domain). The mapping is represented by a list *ns* :: *nat list* that maps *n* to *ns* ! *n*. The details of the example are in some sense irrelevant but make the exposition more realistic. However, we hide most proofs or show only the characteristic opening.

The list representation of the mapping can be abstracted to a relation. The order (*ns* ! *n*, *n*) is the order that *wf* expects.

definition *rel* :: *nat list* ⇒ (*nat* ∗ *nat*) *set* **where**
rel ns = *set*(*zip ns* [0..<*length ns*])

lemma *finite_rel*[*simp*]: *finite*(*rel ns*)

This relation should be acyclic to guarantee termination of the partial functions defined below.

4.1 Tail recursion

If a function is tail-recursive, an executable definition is easy:

partial_function (*tailrec*) *follow* :: *nat list* ⇒ *nat* ⇒ *nat* **where**
follow ns n = (*if n* < *length ns then follow ns* (*ns*!*n*) *else n*)

Informing the code generator:

declare *follow.simps*[*code*]

Now *follow* is executable:

value *follow* [1,2,3] 0

For proofs about *follow* we need a *wf* relation on (*ns*, *n*) pairs that decreases with each recursive call. The first component stays the same but must be acyclic. The second component must decrease w.r.t *rel*:

definition *rel_follow* = *same_fst* (*acyclic o rel*) *rel*

lemma *wf_follow*: *wf rel_follow*

This is how you start an inductive proof about *follow* along *rel_follow*:

lemma *acyclic*(*rel ms*) $\Longrightarrow$ *follow ms m* = *n* $\Longrightarrow$ *length ms* $\leq$ *n*
proof (*induction* (*ms*,*m*) *arbitrary*: *m n rule*: *wf_induct_rule*[*OF wf_follow*])

4.2 Option

If the function is not tail-recursive, not all is lost: if we rewrite it to return an *option* type, it can still be defined. In this case *Some x* represents the result *x* and *None* represents represents nontermination. For example, counting the length of the chain represented by *ns* can be defined like this:

partial_function (*option*) *count* :: *nat list* $\Rightarrow$ *nat* $\Rightarrow$ *nat option* **where**
count ns n
= (*if n* < *length ns then do* {*k* $\leftarrow$ *count ns* (*ns*!*n*); *Some* (*k*+1)} *else Some* 0)

We use a Haskell-like *do* notation (import *HOL−Library.Monad_Syntax*) to abbreviate the clumsy explicit
case count ns (*ns* ! *n*) *of None* $\Rightarrow$ *None* | *Some k* $\Rightarrow$ *Some* (*k* + 1).
The branch *None* $\Rightarrow$ *None* represents the requirement that nontermination of a recursive call must lead to nontermination of the function.

Now we can prove that *count* terminates for all acyclic maps:

lemma *acyclic*(*rel ms*) $\Longrightarrow$ $\exists$ *k*. *count ms m* = *Some k*
proof (*induction* (*ms*,*m*) *arbitrary*: *ms m rule*: *wf_induct_rule*[*OF wf_follow*])

4.3 Subtype

In this approach we define a new type that contains only elements on which the function in question terminates. In our example that is the subtype of all *ns* :: *nat list* such that *rel ns* is acyclic. Then *follow* can be defined as a total function on that subtype.

The subtype is not empty:

lemma *acyclic_rel_Nil*: *acyclic*(*rel* [])

Definition of subtype *acyc*:

typedef *acyc* = {*ns*. *acyclic*(*rel ns*)}

morphisms *rep_acyc abs_acyc*
using *acyclic_rel_Nil* **by** *auto*

This defines two functions *rep_acyc* :: *acyc* ⇒ *nat list* and *abs_acyc* :: *nat list* ⇒ *acyc*. Function *abs_acyc* is only defined on acyclic lists and not executable for that reason. Type *dlist* in Section 2.5 is defined in the same manner.

The following command sets up infrastructure for lifting functions on *nat list* to *acyc* (used by **lift_definition** below) [14].

setup_lifting *type_definition_acyc*

This is how *follow* can be defined on *acyc*:

function *follow*2 :: *acyc* ⇒ *nat* ⇒ *nat* **where**
*follow*2 *ac n*
= (*let ns* = *rep_acyc ac in if n* < *length ns then follow*2 *ac* (*ns*!*n*) *else n*)
by *pat_completeness auto*

Now we prepare for the termination proof. Relation *rel_follow*2 is almost identical to *rel_follow*.

definition *rel_follow*2 = *same_fst* (*acyclic o rel o rep_acyc*) (*rel o rep_acyc*)

lemma *wf_follow*2: *wf rel_follow*2

Here comes the actual termination proof:

termination *follow*2
proof
show *wf rel_follow*2

next
show ⋀*ac n ns*. ⟦ *ns* = *rep_acyc ac*; *n* < *length ns* ⟧
⟹ ((*ac*, *ns* ! *n*), (*ac*, *n*)) ∈ *rel_follow*2

qed

Inductive proofs about *follow*2 can now simply use computation induction:

lemma *follow*2 *ac m* = *n* ⟹ *length* (*rep_acyc ac*) ≤ *n*
proof (*induction ac m arbitrary*: *n rule*: *follow*2.*induct*)

A complication with the subtype approach is that injection into the subtype (function *abs_acyc* in our example) is not executable. But to call *follow*2, we need an argument of type *acyc* and we need to obtain it in an executable manner. There are two approaches.

In the first approach we check wellformedness (i.e. acyclicity) explicitly. This check needs to be executable (which *acyclic* and *rel* are). If the check fails, [] is returned (which is acyclic).

lift_definition *is_acyc* :: *nat list* ⇒ *acyc* **is**
λ*ns*. *if acyclic*(*rel ns*) *then ns else* []

This works because we can easily prove that for any *ns*, the λ-term produces an acyclic list. But it requires the possibly expensive check *acyclic* (*rel ns*).

definition *follow_test ns n* = *follow*2 (*is_acyc ns*) *n*

The relation is acyclic (a chain):

value *follow_test* [1,2,3] 1

In the second approach, wellformedness of the argument is guaranteed by construction. In our example [1..<*n*+1] represents an acyclic chain $i \mapsto i{+}1$

lemma *acyclic_chain*: *acyclic* (*rel* [1..<*n*+1])

lift_definition *acyc_chain* :: *nat* ⇒ *acyc* **is** λ*n*. [1..<*n*+1]

definition *follow_chain m n* = *follow*2 (*acyc_chain m*) *n*

value *follow_chain* 5 1

The subtype approach requires neither tail-recursion nor *option* but you cannot easily modify arguments of the subtype except via existing functions on the subtype. Otherwise you need to inject some value into the subtype and that injection is not computable.

5 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 ⟹ *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.

5.1 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* **.**

5.2 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* ⟹ *lexordp r xs ys*

lemma [*code_pred_intro*]:
append u (*a* # *v*) *xs* ⟹ *append u* (*b* # *w*) *ys* ⟹ *r a b*
⟹ *lexordp r xs ys*

code_pred *lexordp*

5.3 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* ⇒ *o* ⇒ *bool*, *i* ⇒ *o* ⇒ *bool*, *o* ⇒ *i* ⇒ *bool* and *i* ⇒ *i* ⇒ *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* ⇒ *o* ⇒ *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*] 1 {*n*. *tranclp succ* 10 *n*}
values [*mode*: *o* ⇒ *i* ⇒ *bool*] 1 {*n*. *tranclp succ n* 10}

5.4 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*.

5.5 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.

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

There is a rich palette of evaluation techniques, each comprising different aspects:

Expressiveness. Depending on the extent to which symbolic computation is possible, the class of terms which can be evaluated can 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*)

Considerable performance can be achieved using evaluation in ML, at the cost of being restricted to ground results and a layered stack of code to be trusted, including a user's specific code generator setup.

Evaluation is carried out in a target language *Eval* which inherits from *SML* but for convenience uses parts of the Isabelle runtime environment. Hence soundness depends crucially on the correctness of the code generator setup; this is one of the reasons why you should not use adaptation (see §8) frivolously.

6.2 Dynamic evaluation

Dynamic evaluation takes the code generator configuration "as it is" at the point where evaluation is issued and computes a corresponding result. Best example is the **value** command for 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.

A particular technique may be specified in square brackets, e.g.

value [*nbe*] 42 / (12 :: *rat*)

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

Term reconstruction in ML

Results from evaluation in ML must be turned into Isabelle's internal term representation again. Since that setup is highly configurable, it is never assumed to be trustable. Hence evaluation in ML provides no full term reconstruction but distinguishes the following kinds:

Plain evaluation. A term is normalized using the vanilla term reconstruction from ML to Isabelle; this is a pragmatic approach for applications which do not need trustability.

Property conversion. Evaluates propositions; since these are monomorphic, the term reconstruction is fixed once and for all and therefore trustable – in the sense that only the regular code generator setup has to be trusted, without relying on term reconstruction from ML to Isabelle.

The different degree of trustability is also manifest in the types of the corresponding ML functions: plain evaluation operates on uncertified terms, whereas property conversion operates on certified terms.

The partiality principle

During evaluation exceptions indicating a pattern match failure or a non-implemented function are treated specially: as sketched in §2.5, such exceptions can be interpreted as partiality. For plain evaluation, the result hence is optional; property conversion falls back to reflexivity in such cases.

Schematic overview

	simp	*nbe*	*code*
interactive evaluation plain evaluation	**value** [*simp*]	**value** [*nbe*]	**value** [*code*] `Code_Evaluation.dynamic_value`
evaluation method property conversion	*code_simp*	*normalization*	*eval* `Code_Runtime.dynamic_holds_conv`
conversion	`Code_Simp.dynamic_conv`	`Nbe.dynamic_conv`	

6.3 Static evaluation

When implementing proof procedures using evaluation, in most cases the code generator setup is appropriate "as it was" when the proof procedure was written in ML, not an arbitrary later potentially modified setup. This is called static evaluation.

Static evaluation using *simp* and *nbe*

For *simp* and *nbe* static evaluation can be achieved using `Code_Simp.static_conv` and `Nbe.static_conv`. Note that `Nbe.static_conv` by its very nature requires an invocation of the ML compiler for every call, which can produce significant overhead.

Intimate connection between logic and system runtime

Static evaluation for *eval* operates differently – the required generated code is inserted directly into an ML block using antiquotations. The idea is that generated code performing static evaluation (called a *computation*) is compiled once and for all such that later calls do not require any invocation of the code generator or the ML compiler at all. This topic deserves a dedicated chapter §7.

7 Computations

7.1 Prelude – 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;
›
```

The antiquotation *code* takes the name of a constant as argument; the required code is generated transparently and the corresponding constant names are inserted for the given antiquotations. This technique 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.

7.2 The concept of computations

Computations embody the simple idea that for each monomorphic Isabelle/HOL term of type τ by virtue of code generation there exists an corresponding ML type T and a morphism $\Phi :: \tau \to T$ satisfying $\Phi\ (t_1 \cdot t_2) = \Phi\ t_1 \cdot \Phi\ t_2$, with $\cdot$ denoting term application.

For a given Isabelle/HOL type τ, parts of Φ can be implemented by a corresponding ML function $\varphi_\tau :: \mathit{term} \to T$. How?

Let input be a constant $C :: \tau$.
Then $\varphi_\tau\ C = f_C$ with f_C being the image of C under code generation.

Let input be an application $(t_1 \cdot t_2) :: \tau$.
Then $\varphi_\tau\ (t_1 \cdot t_2) = \varphi_\tau\ t_1\ (\varphi_\tau\ t_2)$.

Using these trivial properties, each monomorphic constant $C : \overline{\tau}_n \to \tau$ yields the following equations:

$$\begin{aligned}
&\varphi_{(\tau_1 \to \tau_2 \to \ldots \to \tau_n \to \tau)}\ C = f_C \\
&\varphi_{(\tau_2 \to \ldots \to \tau_n \to \tau)}\ (C \cdot t_1) = f_C\ (\varphi_{\tau_1}\ t_1) \\
&\ldots \\
&\varphi_\tau\ (C \cdot t_1 \cdot \ldots \cdot t_n) = f_C\ (\varphi_{\tau_1}\ t_1) \ldots (\varphi_{\tau_n}\ t_n)
\end{aligned}$$

Hence a computation is characterized as follows:

- Let *input constants* denote a set of monomorphic constants.
- Let τ denote a monomorphic type and *'ml* be a schematic placeholder for its corresponding type in ML under code generation.
- Then the corresponding computation is an ML function of type `Proof.context -> term -> 'ml` partially implementing the morphism $\Phi :: \tau \to T$ for all *input terms* consisting only of input constants and applications.

The charming idea is that all required code is automatically generated by the code generator for givens input constants and types; that code is directly inserted into the Isabelle/ML runtime system by means of antiquotations.

7.3 The *computation* antiquotation

The following example illustrates its basic usage:

```
ML ‹
  local

  fun int_of_nat @{code 0 :: nat} = 0
    | int_of_nat (@{code Suc} n) = int_of_nat n + 1;

  in

  val comp_nat = @{computation nat terms:
    plus :: nat ⇒ nat ⇒ nat times :: nat ⇒ nat ⇒ nat
    sum_list :: nat list ⇒ nat prod_list :: nat list ⇒ nat
    datatypes: nat nat list}
    (fn post => post o HOLogic.mk_nat o int_of_nat o the);

  end
›
```

- Antiquotations occurring in the same ML block always refer to the same transparently generated code; particularly, they share the same transparently generated datatype declarations.
- The type of a computation is specified as first argument.
- Input constants are specified the following ways:
  - Each term following *terms*: specifies all constants it contains as input constants.
  - Each type following *datatypes*: specifies all constructors of the corresponding code datatype as input constants. Note that this does not increase expressiveness but succinctness for datatypes with many constructors. Abstract type constructors are skipped silently.
- The code generated by a *computation* antiquotation takes a functional argument which describes how to conclude the computation. What's the rationale behind this?
  - There is no automated way to generate a reconstruction function from the resulting ML type to a Isabelle term – this is in the responsibility of the implementor. One possible approach for robust term reconstruction is the *code* antiquotation.

- Both statically specified input constants and dynamically provided input terms are subject to preprocessing. Likewise the result is supposed to be subject to postprocessing; the implementor is expected to take care for this explicitly.
- Computations follow the partiality principle (cf. §6.2): failures due to pattern matching or unspecified functions are interpreted as partiality; therefore resulting ML values are optional.

Hence the functional argument accepts the following parameters

- A postprocessor function `term -> term`.
- The resulting value as optional argument.

The functional argument is supposed to compose the final result from these in a suitable manner.

A simple application:

ML_val ‹
  comp_nat ***context term*** ‹*sum_list* [*Suc* 0, *Suc* (*Suc* 0)] ∗ *Suc* (*Suc* 0)›
›

A single ML block may contain an arbitrary number of computation antiquotations. These share the *same* set of possible input constants. In other words, it does not matter in which antiquotation constants are specified; in the following example, *all* constants are specified by the first antiquotation once and for all:

ML ‹
  local

  fun int_of_nat @{*code* 0 :: *nat*} = 0
    | *int_of_nat* (@{*code Suc*} *n*) = *int_of_nat n* + 1;

  in

  val comp_nat = @{*computation nat terms*:
    plus :: *nat* ⇒ *nat* ⇒ *nat times* :: *nat* ⇒ *nat* ⇒ *nat*
    sum_list :: *nat list* ⇒ *nat prod_list* :: *nat list* ⇒ *nat*
    replicate :: *nat* ⇒ *nat* ⇒ *nat list*
    datatypes: *nat nat list*}
    (*fn post* => *post o HOLogic.mk_nat o int_of_nat o the*);

```
  val comp_nat_list = @{computation nat list}
    (fn post => post o HOLogic.mk_list typ‹nat› o
      map (HOLogic.mk_nat o int_of_nat) o the);

  end
›
```

7.4 Pitfalls when specifying input constants

Complete type coverage. Specified input constants must be *complete* in the sense that for each required type τ there is at least one corresponding input constant which can actually *construct* a concrete value of type τ, potentially requiring more types recursively; otherwise the system of equations cannot be generated properly. Hence such incomplete input constants sets are rejected immediately.

Unsuitful right hand sides. The generated code for a computation must compile in the strict ML runtime environment. This imposes the technical restriction that each compiled input constant f_C on the right hand side of a generated equations must compile without throwing an exception. That rules out pathological examples like *undefined* :: *nat* as input constants, as well as abstract constructors (cf. §3.3).

Preprocessing. For consistency, input constants are subject to preprocessing; however, the overall approach requires to operate on constants C and their respective compiled images f_C.[1] This is a problem whenever preprocessing maps an input constant to a non-constant.

To circumvent these situations, the computation machinery has a dedicated preprocessor which is applied *before* the regular preprocessing, both to input constants as well as input terms. This can be used to map problematic constants to other ones that are not subject to regular preprocessing. Rewrite rules are added using attribute *code_computation_unfold*. There should rarely be a need to use this beyond the few default setups in HOL, which deal with literals (see also §7.8).

7.5 Pitfalls concerning input terms

No polymorphism. Input terms must be monomorphic: compilation to ML requires dedicated choice of monomorphic types.

[1] Technical restrictions of the implementation enforce this, although those could be lifted in the future.

No abstractions. Only constants and applications are admissible as input; abstractions are not possible. In theory, the compilation schema could be extended to cover abstractions also, but this would increase the trusted code base. If abstractions are required, they can always be eliminated by a dedicated preprocessing step, e.g. using combinatorial logic.

Potential interfering of the preprocessor. As described in §7.4 regular preprocessing can have a disruptive effect for input constants. The same also applies to input terms; however the same measures to circumvent that problem for input constants apply to input terms also.

7.6 Computations using the *computation_conv* antiquotation

Computations are a device to implement fast proof procedures. Then results of a computation are often assumed to be trustable and thus wrapped in oracles (see [14]), as in the following example:[2]

```
ML ‹
  local

  fun raw_dvd (b, ct) =
    instantiate ‹x = ct and y = ‹if b then cterm ‹True› else cterm ‹False››
      in cterm ‹x ≡ y› for x y :: bool›;

  val (_, dvd_oracle) = Theory.setup_result (Thm.add_oracle (binding ‹dvd›,
raw_dvd));

  in

  val conv_dvd = @{computation_conv bool terms:
    Rings.dvd :: int ⇒ int ⇒ bool
    plus :: int ⇒ int ⇒ int
    minus :: int ⇒ int ⇒ int
    times :: int ⇒ int ⇒ int
    0 :: int 1 :: int 2 :: int 3 :: int −1 :: int
  } (K (curry dvd_oracle))
```

[2]The technical details how numerals are dealt with are given later in §7.8.

```
  end
›
```

- Antiquotation *computation_conv* basically yields a conversion of type `Proof.context -> cterm -> thm` (see further [13]).
- The antiquotation expects one functional argument to bridge the "untrusted gap"; this can be done e.g. using an oracle. Since that oracle will only yield "valid" results in the context of that particular computation, the implementor must make sure that it does not leave the local ML scope; in this example, this is achieved using an explicit *local* ML block. The very presence of the oracle in the code acknowledges that each computation requires explicit thinking before it can be considered trustworthy!
- Postprocessing just operates as further conversion after normalization.
- Partiality makes the whole conversion fall back to reflexivity.

```
ML_val ‹
  conv_dvd context cterm ‹7 dvd ( 62437867527846782 :: int)›;
  conv_dvd context cterm ‹7 dvd (−62437867527846783 :: int)›;
›
```

An oracle is not the only way to construct a valid theorem. A computation result can be used to construct an appropriate certificate:

```
lemma conv_div_cert:
  (Code_Target_Int.positive r ∗ Code_Target_Int.positive s)
    div Code_Target_Int.positive s ≡ (numeral r :: int) (is ?lhs ≡ ?rhs)
proof (rule eq_reflection)
  have ?lhs = numeral r ∗ numeral s div numeral s
    by simp
  also have numeral r ∗ numeral s div numeral s = ?rhs
    by (rule nonzero_mult_div_cancel_right) simp
  finally show ?lhs = ?rhs .
qed

lemma positive_mult:
  Code_Target_Int.positive r ∗ Code_Target_Int.positive s =
    Code_Target_Int.positive (r ∗ s)
  by simp
```

```
ML ‹
  local

  fun integer_of_int (@{code int_of_integer} k) = k

  val cterm_of_int = Thm.cterm_of context o HOLogic.mk_numeral o
integer_of_int;

  val divisor = Thm.dest_arg o Thm.dest_arg;

  val evaluate_simps = map mk_eq @{thms arith_simps positive_mult};

  fun evaluate ctxt =
    Simplifier.rewrite_rule ctxt evaluate_simps;

  fun revaluate ctxt k ct =
    @{thm conv_div_cert}
    |> Thm.instantiate' [] [SOME (cterm_of_int k), SOME (divisor ct)]
    |> evaluate ctxt;

  in

  val conv_div = @{computation_conv int terms:
    divide :: int ⇒ int ⇒ int
    0 :: int 1 :: int 2 :: int 3 :: int
  } revaluate

  end
›

ML_val ‹
  conv_div context
    cterm ‹467824543434999999927777424323422423234234 25 div (7 :: int)›
›
```

The example is intentionally kept simple: only positive integers are covered, only remainder-free divisions are feasible, and the input term is expected to have a particular shape. This exhibits the idea more clearly: the result of the computation is used as a mere hint how to instantiate *conv_div_cert*, from which the required theorem is obtained by performing multiplication using the simplifier; hence that theorem is built of proper inferences, with no oracles involved.

7.7 Computations using the *computation_check* antiquotation

The *computation_check* antiquotation is convenient if only a positive checking of propositions is desired, because then the result type is fixed (*prop*) and all the technical matter concerning postprocessing and oracles is done in the framework once and for all:

ML ‹
  val check_nat = @{*computation_check terms*:
    Trueprop less :: *nat* ⇒ *nat* ⇒ *bool plus* :: *nat* ⇒ *nat* ⇒ *nat*
    times :: *nat* ⇒ *nat* ⇒ *nat*
    0 :: *nat Suc*
  }
›

The HOL judgement *Trueprop* embeds an expression of type *bool* into *prop*.

ML_val ‹
  check_nat ***context*** ***cprop*** ‹*less* (*Suc* (*Suc* 0)) (*Suc* (*Suc* (*Suc* 0)))›
›

Note that such computations can only *check* for *prop*s to hold but not *decide*.

7.8 Some practical hints

Inspecting generated code

The antiquotations for computations attempt to produce meaningful error messages. If these are not sufficient, it might by useful to inspect the generated code, which can be achieved using

declare [[*ML_source_trace*]]

Literals as input constants

Literals (characters, numerals) in computations cannot be processed naively: the compilation pattern for computations fails whenever target-language literals are involved; since various common code generator setups (see §8.3) implement *nat* and *int* by target-language literals, this problem manifests whenever numeric types are involved. In practice, this is circumvented with a dedicated preprocessor setup for literals (see also §7.4).

The following examples illustrate the pattern how to specify input constants when literals are involved, without going into too much detail:

An example for *nat*

```
ML ‹
  val check_nat = @{computation_check terms:
    Trueprop even :: nat ⇒ bool plus :: nat ⇒ nat ⇒ nat
    0 :: nat Suc 1 :: nat 2 :: nat 3 :: nat
  }
›
```

```
ML_val ‹
  check_nat context cprop ‹even (Suc 0 + 1 + 2 + 3 + 4 + 5)›
›
```

An example for *int*

```
ML ‹
  val check_int = @{computation_check terms:
    Trueprop even :: int ⇒ bool plus :: int ⇒ int ⇒ int
    0 :: int 1 :: int 2 :: int 3 :: int −1 :: int
  }
›
```

```
ML_val ‹
  check_int context cprop ‹even ((0::int) + 1 + 2 + 3 + −1 + −2 + −3)›
›
```

An example for *String.literal*

```
definition is_cap_letter :: String.literal ⇒ bool
  where is_cap_letter s ⟷ (case String.asciis_of_literal s
    of [] ⇒ False | k # _ ⇒ 65 ≤ k ∧ k ≤ 90)  ML ‹
  val check_literal = @{computation_check terms:
    Trueprop is_cap_letter datatypes: bool String.literal
  }
›
```

```
ML_val ‹
  check_literal context cprop ‹is_cap_letter (STR ''Q'')›
›
```

Preprocessing HOL terms into evaluable shape

When integrating decision procedures developed inside HOL into HOL itself, a common approach is to use a deep embedding where operators etc. are represented by datatypes in Isabelle/HOL. Then it is necessary to turn generic shallowly embedded statements into that particular deep embedding ("reification").

One option is to hardcode using code antiquotations (see §7.1). Another option is to use pre-existing infrastructure in HOL: `Reification.conv` and `Reification.tac`.

A 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* ∧ *x* < *z*››

By virtue of *interp.simps*, `Reification.conv` provides a conversion which, for this concrete example, yields *x* < *y* ∧ *x* < *z* ≡ *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 thus be evaluated using evaluation.

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

8 Adaptation to target languages

8.1 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.

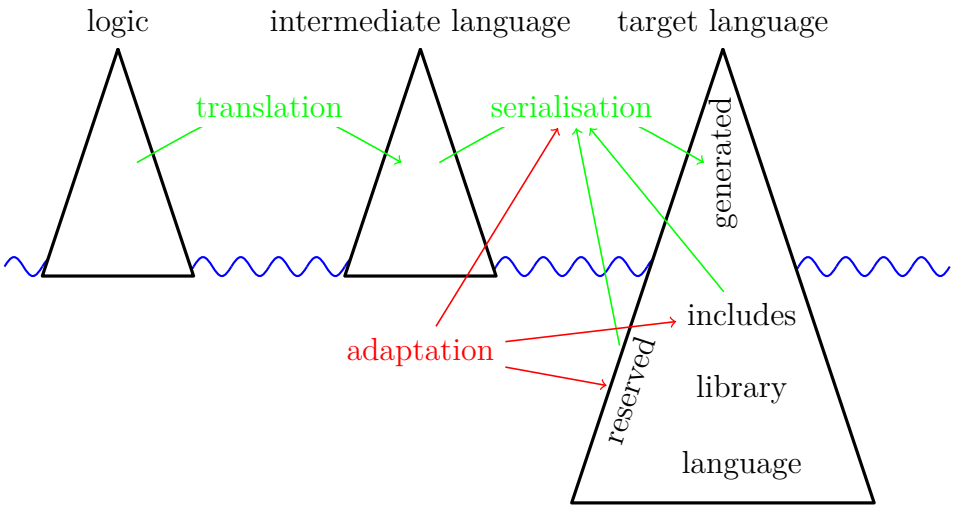

Figure 2: The adaptation principle

8.2 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 §8.4)
- 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 §8.6); 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.

8.3 Common adaptation applications

The *Main* theory of Isabelle/HOL 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 §8.4).

HOL.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*.

HOL.String provides an additional datatype *String.literal* which is isomorphic to lists of 7-bit (ASCII) characters; *String.literal*s are mapped to target-language strings.

Literal values of type *String.literal* can be written as *STR* ″...″ for sequences of printable characters and *STR* 0*x*... for one single ASCII code point given as hexadecimal numeral; *String.literal* supports concatenation ... + ... for all standard target languages.

Note that the particular notion of "string" is target-language specific (sequence of 8-bit units, sequence of unicode code points, ...); hence ASCII is the only reliable common base e.g. for printing (error) messages; more sophisticated applications like verifying parsing algorithms require a dedicated target-language specific model.

Nevertheless *String.literal*s can be analyzed; the core operations for this are *String.asciis_of_literal* and *String.literal_of_asciis* which are implemented in a target-language-specific way; particularly *String.asciis_of_literal* checks its argument at runtime to make sure that it does not contain non-ASCII-characters, to safeguard consistency. On top of these, more abstract conversions like *literal.explode* and *String.implode* are implemented.

Part of *HOL−Main*.

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

Using these adaptation setups the following extensions are provided:

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 *Code_Target_Nat*, *Code_Target_Int* and *Code_Target_Bit_Shifts*-

Code_Bit_Shifts_for_Arithmetic uses the preprocessor to replace arithmetic operations on numeric types by target-language built-in bit shifts whenever feasible.

Code_Abstract_Char implements type *char* by target language integers, sacrificing pattern patching in exchange for dramatically increased performance for comparisons.

8.4 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 Zero_nat n = True
  | less_eq_nat (Suc m) n = less_nat m n
and less_nat n Zero_nat = False
```

```
  | less_nat m (Suc n) = less_eq_nat m n;

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

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

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.

8.5 *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 standard (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) -
```

8.6 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)
 ‹module Errno(errno) where

  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.

9 Further issues

9.1 Runtime environments for *Haskell* and *OCaml*

The Isabelle System Manual [12] provides some hints how runtime environments for *Haskell* and *OCaml* can be set up and maintained conveniently using managed installations within the Isabelle environments.

9.2 Incorporating generated code directly into the system runtime – *code_reflect*

Static embedding of generated code into the system runtime

The *code* antiquotation 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 *HOL.Predicate* theory.

Separate compilation

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_prefix** argument can be used:

```
code_reflect Rat
  datatypes rat
  functions Fract
    (plus :: rat ⇒ rat ⇒ rat) (minus :: rat ⇒ rat ⇒ rat)
    (times :: rat ⇒ rat ⇒ rat) (divide :: rat ⇒ rat ⇒ rat)
  file_prefix rat
```

This generates the referenced code as logical files within the theory context, similar to **export_code**.

9.3 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.

Note also that name clashes can occur when generating Scala code to case-insensitive file systems; these can be avoid using the "(*case_insensitive*)" option for **export_code**.

9.4 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 ⇀ (SML) ABC
| code_module B ⇀ (SML) ABC
| code_module C ⇀ (SML) ABC
```

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

9.5 Locales and interpretation

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

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

```
locale power =
  fixes power :: 'a ⇒ 'b ⇒ 'b
  assumes power_commute: power x ∘ power y = power y ∘ power x
begin
```

Inside that locale we can lift *power* to exponent lists by means of a 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, **global_interpretation** *fun_power*: *power* (λ*n* (*f* :: 'a ⇒ 'a). *f* ^^ *n*)), one could naively 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, a succint solution is available: a dedicated rewrite definition:

```
global_interpretation fun_power: power (λn (f :: 'a ⇒ 'a). f ^^ n)
  defines funpows = fun_power.powers
  by unfold_locales
    (simp_all add: fun_eq_iff funpow_mult mult.commute)
```

This amends the interpretation morphisms such that occurrences of the foundational term *power.powers* (λ*n* (*f* :: 'a ⇒ 'a). *f* ^^ *n*) are folded to a newly defined constant *funpows*.

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;
```

9.6 Parallel computation

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

9.7 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.

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.* https://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 Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2008*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

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

[6] 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.

[7] 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.

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

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

[10] 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.

[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 System Manual.* https://isabelle.in.tum.de/doc/system.pdf.

[13] Makarius Wenzel. *The Isabelle/Isar Implementation.* https://isabelle.in.tum.de/doc/implementation.pdf.

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