A Generic Code Generator for Isabelle/HOL[1]

T. Meyer†, B. Wolff‡

†Universität Bremen, FB3
Postfach 330440
D-28334 Bremen
tm@informatik.uni-bremen.de

‡Albert-Ludwigs Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Softwaretechnik
wolff@informatik.uni-freiburg.de

Abstract. One major motivation for theorem provers is the development of verified programs. In particular, synthesis- or transformational development techniques aim at a formalised conversion of the original specification to a final formula meeting some notion of executability. We present three such notions, based on three different forms of recursion (two denotational and one based on well-founded recursion) and develop their theory in Isabelle/HOL. These theories serve as a semantic interface for a generic code-generator which is set up for each program notion with an individual code-scheme for SML.

1 Introduction

Formal Program Development, be it based on transformation or synthesis techniques, is centred around a notion of development that can be represented by a sequence of specifications

$$SP_1 \rightsquigarrow ... \rightsquigarrow SP_n.$$

Pragmatically, SP_1 is intended to be the *requirement specification* corresponding as closest possible to an abstract informal requirement analysis. In contrast, the *executable specification* SP_n is so concrete and syntactically restricted that it can be considered as an abstract program. In this paper we show, how the idea to use Higher Order Logic (HOL) as technical framework for the implementation of formal program development (based on VDM[AF 97], CSP[TW 97] or Z[KSW 96b]) can be extended to a generic environment for code generation as well.

Code-generators serve as bridge between a theorem prover and the outside world. Hence, it is not completely possible to *verify* a code-generator inside a logical embedding; their proof must be based on meta-logical, model-theoretic arguments similar to the soundness proofs of the basic rules of a logic (cf. [GM 93]). However, we will show how the first principle of LCF-style theorem prover design, namely the enclosure of extra-logical machinery in a *minimal* kernel, can be adopted to the construction of external code in order to increase the *trustworthiness* of such a code-generation. We will provide three formalised notions of execution serving as small interfaces between the logical and the extra logical side. The translation of the constructs of these interfaces is intended to be one to one, while program optimisations should be performed by tactics on the logical side.

1 This work has been supported by the German Ministery for Education and Research (BMBF) as part of the project **UniForM** under grant No. FKZ 01 IS 521 B2.

While trustworthiness and correctness is clearly an overall concern for the generated code, a trade-off has to be made between *coverage* (the portion of specification language which can be successfully translated into code) and *efficiency* (the speed of the resulting code) — we follow the terminology established in [BB 96] here.

From the informal requirements for code generation stated above follows already the necessity of some form of semantic bracket that separates executable from non-executable parts of a specification. As a motivation for this, let us consider the following example due to T. Nipkow: We assume the following two theorems (possibly axioms):

```
"fac 0 = 1"          "fac n = n * fac (n-1)"
```

where fac is assumed to have the type fac: $\mathbb{Z}\rightarrow\mathbb{Z}$. Now, an naive code generation could convert this into the following SML-code:

```
fun fac 0 = 1 | fac n = n * fac (n-1)
```

This piece of code is simple; unfortunately, it is simply wrong. If we consider fac -1, the program above diverges, as any functional programmer will expect. However, from the two theorems above (based on the theory of integers and transitively on the theory of natural numbers), it can be formally proven that any model for these axioms above must fulfil fac -1 = 0 ![2] Of course, no simple, hence trustworthy code-generator can deduce this fact automatically and produce code accordingly. The deeper cause for this failure of our naive code generator is that SML-interpretations assume a fixed orientation of the equations corresponding to the *least fixpoint semantics* of an equation system. This has little to do with the concepts of *truth* and *validity* of theorems underlying HOL. Hence, some construct on the logical side like:

```
letrec   fac 0 = 1;
         fac n = n * fac (n-1)
in fac -1 end
```

is needed (and has to be semantically founded within HOL) that suits better to the semantics underlying our target language SML. We will refer to these constructs as a *recursion* or *program notions.*

We will introduce three program notions in this paper. The first based on well-founded recursion corresponds to total correctness and allows the conversion of most functions defined in the HOL-library into executable code. The other two correspond to partial correctness and provide semantics for recursion based on the least-fixpoint operator on sets or on complete partial orders (cpo's). These two enable program notions for Z (suitable for [KSW 96b], for example) or functional programming languages (such as HASKELL or SML).

All three cases have to cope with some form of *partiality*. We distinguish *exception partiality* (hd []) from *recursion partiality* (see non-termination for fac

---

2 **Proof-sketch**: We prove by induction over n: fac(-1) = (if odd(n+1) then 1 else -1) * n! * fac(-$\lfloor$n+1$\rfloor$) where ! is the usual faculty-function on natural numbers and $\lfloor . \rfloor$ is the embedding of natural numbers into integer. As a consequence of this lemma, f(-1) must be devisable by all n! Hence, from ($\forall$ x:nat. c = k * x) = (c=0 $\wedge$ k=0) follows that fac (x) must be 0 for all negative x.

above). By its essence, exception partiality can be avoided by another definition of the functions: hd could be defined to produce an optional value etc., restricting the code generator to a smaller coverage. In contrast, recursion partiality is unavoidable; it is well-known that the class of partial functions is strictly larger than the class of total functions; and this gain in expressive power is desirable in programming practice in order to write, for example, universal interpreters in a programming language.

In the HOL-library, exception and recursion partiality is modelled by the Hilbert operator $\iota x{:}\tau.\ P\ x$ (choose an arbitrary x of type τ such that P holds). In any model of a theory, the expression ($\iota x{:}\tau.$ True) is hence given a different interpretation. This gives rise to the following *correctness notion* for generated code:

> Let A be an executable term of type τ. If all models of the theory of A agree on one value, v say, then the interpreter of the code must terminate and yield v. Otherwise the interpreter may diverge (recursion partiality), raise an exception or produce a value v' that is valid in some model of A.

This paper proceeds as follows: After a short introduction to Isabelle and HOL, we develop in three subsequent sections what we consider the main contribution of this paper: the set-up of three semantics for recursion, their syntactic support and the generic code-generator together with the nitty-gritty details of its three instantiations. We conclude with a discussion of experimental data gained from simple benchmark tests.

2 Isabelle and HOL

2.1 Isabelle

Isabelle [Isabelle] is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

In the sequel, all Isabelle input and output will be denoted in this FONT throughout this paper — enriched by the usual mathematical notation for ∀, ∃,... instead of ASCII-transcriptions.

Isabelle belongs to the family of LCF-style theorem provers. This means it is a set of data types and function definitions in the ML-environment (or: "data-base"). The crucial one is the abstract data type "thm" (protected by the ML type discipline) that contains the formulas accepted by Isabelle as theorems. thm-objects can only be constructed via operations of the logical kernel of Isabelle. This architecture allows to provide user-programmed extensions of Isabelle without corrupting the logical kernel.

2.2 Higher Order Logic (HOL)

In this section, we will give a short overview of the concepts and the syntax. Our logical language HOL goes back to [Chu 40]; a more recent presentation is [And 86]. In the formal methods community, it has achieved some acceptance, especially in hardware-verification. HOL is a classical logic with equality formed over the usual logical connectives ¬, ∧, ∨, ⇒ and = for negation, conjunction, disjunction, implication and equality. It is based on total functions denoted by λ-abstractions like "$\lambda x.x$". Function application is denoted by $f\,a$. Every term in the logic must be typed, in order to avoid Russels paradox. Isabelle's type discipline incorporates polymorphism with type-classes (as in Haskell). HOL extends predicate calculus in that universal and existential quantification $\forall$x. P x rsp. $\exists$x. P x can range over functions.

2.3 SML

Standard ML (a readable description of the language is [Pau 91]) is a highly standardised, impure functional programming language with a powerful module system. For the latest standardised definition (which also includes a rich library), several implementations are available. Historically, ML has been deeply connected with the development of theorem provers in the LCF-family and is still the basis for many provers, in particular Isabelle. In this environment, SML is therefore a natural choice for a code generator as target language.

SML is based on strict evaluation, i.e. terms like (fn x => 0) (hd []) (corresponding to (λx.0)(hd [])) yield an exception, since the argument of the application is evaluated in any case. This is in clear conflict with the β-reduction of HOL, that will reduce the term above to 0, of course. Hence, a correct code-generation will have to delay the evaluation of arguments. On the other hand, a strictness-analysis on the object-logical side can be easily and safely implemented using appropriate tactics, attempting to "kick out" as many standard-applications in program terms and replace them by strict applications. This greatly enhances the efficiency of the generated code and may exemplify the potential of optimisations.

3 Three Semantics for Recursion

The core of each execution notion is a notion of recursion, i.e. some instance of the characterising scheme:

Y F = F (Y F) where F must fulfil some requirement A.

In the following three sections, we will present instantiations of this scheme, i.e. some operation for the recursion combinator Y and a corresponding A.

In principle, it is *also* possible to represent data-types via such a characterising scheme (this is done for fully conservative data-type packages such as the one developed by Melham or Völker). Since the Isabelle standard package simply characterises data types by a collection of axioms, we decided to convert them one-to-one to SML-data-types.

For all instances, we can provide a uniform program notion formalised by the following definition scheme:

program f F A ≡ (c = Y F ∧ A F)

stating that f is a (recursive) program over body F meeting the recursion constraint A. In the next chapter, we will provide syntactic sugar for this semantic representation (supported by Isabelle's pretty-printing machinery), that will programs look like mutual recursive equation systems with (some simple form of) pattern-matching. Thus, it is possible to develop Isabelle rules that produce from an arbitrary formula a "program" interactively during the proof-process.

3.1 The Language WFREC

Well-founded recursion is the "workhorse" in a world of total functions such as HOL. In Isabelle/HOL, the recursion combinator for this recursion notion is called $wfrec::(\alpha * \alpha)set \to ((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$ developed in the theory WF.thy in the library.

The main result of the theory of well-founded recursion is:

$$wf(r) \Rightarrow wfrec\ r\ H\ a = H\ (cut\ (wfrec\ r\ H)\ r\ a)\ a \qquad [wfrec]$$

where the predicate $wf:(\alpha * \alpha)set \to bool$ states the well-foundedness of a relation (represented as set of pairs) and where cut f r x constructs a function that is identical to f for all smaller values than x w.r.t. the ordering r and undefined (i.e. ιx.True) for all larger values.

wfrec is already close to our desired recursion scheme. The missing link is the concept of *coherence*:

$$coherence\ H\ r \equiv \forall a.\ H\ (cut\ (wfrec\ r\ H)\ r\ a)\ a = H\ (wfrec\ r\ H)\ a$$

which essentially states that the body H uses the function wfrec r H (hence the recursive "call") always with smaller arguments.

Obviously, we have:

$$[|\ wf(r);\ coherence\ H\ r\ |] \Longrightarrow wfrec\ r\ H = H\ (wfrec\ r\ H)$$

which leads to our first instance for a program notion:

$$prog\ f\ F\ r \equiv f = wfrec\ r\ F \wedge wf\ r \wedge coherence\ F\ r$$

3.2 The Language LFP

The theory of least fix-points on sets is defined as follows:

```
Lfp = mono + Prod +
constdefs
  lfp :: [α set → α set] → α set
  "lfp(f) ≡ Inter({u. f(u) <= u})"
```

where the main result:

```
lfp_Tarski    "mono(f) ⇒ lfp(f) = f(lfp(f))"
```

is already exactly what we want:

$$prog\ f\ F \equiv f = lfp\ F \wedge mono\ F$$

In order to use lfp for (partial) recursive functions, we will have to provide some infra-structure for functions represented by their graph:

```
type          α <=> β = (α x β) set
constdefs     "%^" :: α <=> β → α → β
     apply_def "f %^ a = ι b. (a,b) ∈ R"
```

This definition of functions by their graphs opens the way for notions like "total functions", "partial functions" etc. and is exactly the same as used in the Z-Mathematical Toolkit as described in [KSW 96b]. Hence, this is the interface for execution notions appropriate for Z-Specifications.

3.3 The Language FIX

There are many formalisations of denotational semantics in HOL-systems or even in constructive type systems (based on synthetic domains) known in the literature. For Isabelle/HOL, there is most notably HOLCF [Reg 94]. Instead, we will use the generic cpo-theory Fix.thy described in [TW 97]. Both theories have inevitably much in common, but also some differences: Fix is constructed as a lightweight, minimal theory of cpo's (only several hundred lines of proof-code), that is generic with respect to the ordering (in fact, its main motivation for [TW 97] was the ability to exchange the standard-ordering by something so weird like the metric-space ordering on processes ...). In contrast, HOLCF is essentially a rich theory of continuous function spaces on fixed built-in orderings; consequently, HOLCF needs an own application operator ^ giving program-denotations a distinctly different syntactic flavour than FIX.

The definition of cpo's is done by a sequence of *axiomatic class* definitions, i.e. an extension of the HASKELL-class system with semantic constraints. For instance, the class of partial order types po is defined by the statement:

```
axclass po < ord
  le_refl      "x <= x"
  le_trans     "[| x <= y; y <= z |] ⇒ x <= z"
  le_antisym "[| x <= y; y <= x |] ⇒ x = y"
```

After the usual definitions for bottom ⊥, directed sets, upper bounds, least upper bounds, the class po is extended to a class cpo. In this class, the predicate for continuity cont: (α::cpo → 'β::cpo) → bool and the fixpoint operator fix : (α::cpo → α) → α (defined as least upperbound of the set of function iterations) were defined and provide as main result:

conf f ⟹ fix f = f (fix f) [knaster]

our third program notion:

prog f F ≡ f = fix F ∧ cont F

Again, some infra-structure is needed to use this recursion notion for the representation of programs. First, one needs in practice an embedding of total data-types into flat cpo's and strict functions over them. This is done with the type-constructor:

```
datatype 'a up = lift 'a | down
```

After identifying down with ⊥ and providing the usual ordering, the fact that each instance of type-constructor up is of class cpo is made explicit to Isabelle's type-system:

```
instance up :: (term)cpo          (. . .)
```

Analogously, the orderings on pairs and function spaces were defined, proven to have the necessary properties and are shown to be instances of cpo provided that one or both arguments of the type-constructor are cpo's:

```
instance fun :: (term,cpo)cpo     (. . .)
instance "*":: (cpo,cpo)cpo       (. . .)
```

On this basis, the type inference will establish the cpo-bookkeeping for all program-terms behind the scene, implicitly constructing orderings and least upper-bounds. In contrast, continuity must be handled explicitly on the level of the program notion:

program f F A ≡ (c = fix F ∧ cont F)

where it is suppressed syntactically (see next chapter).

4 Syntactical Support for (Mutual) Recursion

In order to parse and pretty-print exeutable programs, we need an appropriate setup of Isabelle's syntax machinery. The schema of the concrete syntax for FIX programs is

$$
\begin{array}{ll}
\textbf{Prog} & \mathsf{f}\ x_1\ \ldots\ xk\ \textbf{is} \\
 & f1\ x1_1\ \ldots\ x1_{m_1} = E1\ x1_1\ \ldots\ x1_{m_1}\ f1\ \ldots\ fn; \\
 & f2\ x2_1\ \ldots\ x1_{m_2} = E2\ x2_1\ \ldots\ x2_{m_2}\ f1\ \ldots\ fn; \\
 & \quad \ldots \\
 & fn\ xn_1\ \ldots\ xn_{m_n} = Enxn_1\ \ldots\ xn_{m_n}\ f1\ \ldots\ fn \\
\textbf{main} & E\ f1\ \ldots\ fn\ x1\ \ldots\ xk
\end{array}
$$

where the xi_{m_h} ,s may be patterns. This syntax consists of a program head f with global program variables x1 ... xk, a mutual recursive equation system for equations f1 ... fn and a body E. The syntax for the language LFP is the same whereas the syntax for the language WFREC has a special construct **measure** ord for the termination order.

Technically, this was implemented by an appropriate setup of Isabelle's syntax machinery. For the language FIX, for instance, we intoduced a syntactic constant

_Prog :: [mletbinds, α, β] => β ("Prog _ is _ main _ ")

representing the concrete syntax, where mletbinds is the type of the equation system, α the type of the program head and β the type of the program body.

To this constant we accocioate a translation function parse_prog to translate the parsed concrete syntax into the program's semantic representation described in section 3. For this association we used Isabelle's translation function mechanism. Each translation function is associated with a name, which triggers calls to it. Such names can be constants or type constructors. The translation function parse_prog takes the Isabelle terms for the program head, the equation system and the program body. The computation of the semantic representation establishes the appropriate binding structure for the various variables involved.

For pretty-printing the semantic representation, we defined another translation function print_prog which is associated with the semantic constant program described in the previous section. These translation functions are defined for the languages LFP and WFREC analogously.

5 A Generic Code Generator

5.1 Design Decisions

As already mentioned at several places, we aim at a trustworthy, simple one-to-one translation of a fixed set of basic executable language constructs. The implementor will have to provide for each of these constructs a *coding template* that associates with each pattern instance a code instance; the generic code-generator will complete these templates to a coding function, that checks all syntactic side conditions and generates an SML-structure. The coder will perform a tactic accommodation-function accom that attempts to convert an arbitrary theorem (e.g. a constant definition based on nat_rec) into the preconceived notion of program and to perform additionally optimisations.

There is a close connection between the coding of undefinedness and the coding of the apply and if-then-else operations. In order to see this, let us consider the two examples for exception and recursion partiality:

(1) $(\lambda x.0)(\iota x.\mathrm{True})$ (2) $(\lambda x.0)(\mathrm{fix}\ (\lambda x.x))$

The β-reduction of HOL imposes for both applications, that the result must be defined and 0. For the languages FIX and LFP based on partial correctness, this implies that the code for the application must comply to a call-by-name evaluation strategy, i.e. the evaluation of the argument must be delayed. In the language WFREC based on total correctness, however, recursion is restricted to functions whose values can be constructed operationally by a finite sequence of operation steps — diverging recursive expressions such as (2) are forbidden. Hence, exception partiality is the *only* source of partiality in WFREC. If we suppress exception partiality, the apply can be coded in the far more efficient call-by-value evaluation strategy. One choice to do this is to represent the undefined term ιx.True by an arbitrary fixed dummy value for each type (say 17 for the natural numbers). This is perfectly legal with respect to our correctness-notion pointed out in the introduction, since the coder can "produce a value v' that is valid in *some* model of A". Although, from the perspective of a functional programmer, this code scheme produces sometimes highly counter intuitive results (Suc (ιx.True) = 18!), the gain of efficency seems to outweigh by far this kind of un-intuitivity. For FIX and LFP, where call-by-name has to be used in any case, we prefer to code undefinedness by exceptions.

Hence, the fixed set of basic executable language constructs consists of undef (for exception partiality), apply, apply_strict, if_then_else and prog, that are sensibly discussed *dans l'ensemble.* The implementor will have to provide with each of these *basic* language constructs a code-template in order to instantiate the code-generator.

We decided to implement code-templates as functions, that maps an Isabelle-term (i.e. Isabelle's internal data-structure for its underlying typed λ-calculus) to an abstract syntax absy for SML (drawn from the NJML-compiler). Thus, the code-generator will produce a SML-structure in abstract syntax that can be directly feeded in to the NJML-compiler. This choice was essentially motivated by efficiency and convenience.

5.2 Coding Schemata

In this section, we will describe the code-templates for the basic language constructs for our three code-generator instances. For convenience, the code-templates will be described in a rule-like notation based on concrete syntax. Schema-Variables in patterns will be represented as outlined font. Note that the schema-variables occurring in the target-patterns stand for the *code* for the term matched by the schema-variable in the pattern.

5.3 The Language WFREC

The core-translation for WFREC-programs is straight-forward and one-to-one, since both the source as the target language are bound to strict evaluation. The only critical case is for undefinedness, where a coding-function gen_dummy has to provided that constructs for each type the code for a basic constant. This function depends on the data-type statements declared in the underlying theory.

Basic construct	Pattern	Target
undef	$\iota x::\tau$. True	gen_dummy(τ)
apply	$\mathbb{S}$ $\mathbb{T}$	$\mathbb{S}$ $\mathbb{T}$
apply_strict	- -	- -
equal	$\mathbb{S}$ = $\mathbb{T}$	$\mathbb{S}$ = $\mathbb{T}$
if_then_else	if $\mathbb{S}$ then $\mathbb{T}$ else $\mathbb{U}$	if $\mathbb{S}$ then $\mathbb{T}$ else $\mathbb{U}$
prog	Prog $\mathbb{f}$ $\mathbb{x1}$ ..$\mathbb{xn}$ is $\mathbb{S}$ main $\mathbb{E}$ measure $\mathbb{Q}$	fun $\mathbb{f}$ $\mathbb{x1}$ ..$\mathbb{xn}$ = let fun $\mathbb{S}$ in $\mathbb{E}$ end

Note that with $\mathbb{S}$ in the prog-Pattern, we only support direct recursions for the time being. Since this code-scheme allows the compilation of any application, we will also call it *standard translations.*

5.4 The Language LFP

For the LFP-language, we provide a call-by-name code-scheme. Expressions will be turned into functions, that compute its value *on demand*, i.e. when a unit () is provided. Strict basic operations like the _+_ in the library and the if_then_else can *demand* their arguments if their value is demanded. On the top-level, the prog-node, the demand of the main-value is initiated.

Basic construct	Pattern	Target
undef	ιx. True	fn () =>raise SYSTEM
apply	$\mathbb{S}$ ^ $\mathbb{T}$	fn () => ($\mathbb{S}$ ())($\mathbb{T}$)
apply_strict	$\mathbb{S}$ ^! $\mathbb{T}$	fn () => (($\mathbb{S}$()) $\mathbb{T}$())
equal	$\mathbb{S}$ = $\mathbb{T}$	fn () => ($\mathbb{S}$() = $\mathbb{T}$())
if_then_else	if $\mathbb{S}$ then $\mathbb{T}$ else $\mathbb{U}$	fn () =>if $\mathbb{S}$() then $\mathbb{T}$ else $\mathbb{U}$
prog	Prog $\mathbb{f}$ $\mathbb{x1}$ ..$\mathbb{xn}$ is $\mathbb{S}$ main $\mathbb{E}$	fun $\mathbb{f}$ $\mathbb{x1}$ ..$\mathbb{xn}$ = let $\mathbb{S}$ in $\mathbb{E}$() end

Note that the S in the prog-pattern stands for a nested term $\lambda(f_1, ...,f_n).(E_1,..E_n)$ that is converted into an SML-mutual recursive scheme:

fun f_1 () = E_1 and ... and fun f_n ()= E_n.

5.5 The Language FIX

The code-scheme for the FIX-language is verly similar to that of LFP. The essential difference is that the patterns are trigered by terms with different types, in particular in the case of the standard application.

Basic construct	Pattern	Target
undef	UU	fn () =>raise SYSTEM
apply	($\mathbb{S}$::$\alpha\rightarrow\beta$::cpo) $\mathbb{T}$	fn () => ($\mathbb{S}$ ())($\mathbb{T}$)
apply_strict	$\mathbb{S}$ ^! $\mathbb{T}$	fn () => (($\mathbb{S}$()) $\mathbb{T}$ ())
equal	$\mathbb{S}$ =::$\alpha\rightarrow\alpha$::cpo $\mathbb{T}$	fn () => ($\mathbb{S}$() = $\mathbb{T}$())
if_then_else	if $\mathbb{S}$ then $\mathbb{T}$ else $\mathbb{U}$	fn () =>if $\mathbb{S}$() then $\mathbb{T}$ else $\mathbb{U}$
prog	Prog $\mathbb{f}$ $\mathbb{x}1$ ..$\mathbb{x}\mathbb{n}$ is $\mathbb{S}$ main $\mathbb{E}$	fun $\mathbb{f}$ $\mathbb{x}1$ ..$\mathbb{x}\mathbb{n}$ = let $\mathbb{S}$ in $\mathbb{E}$() end

The translation of the equation system S is as discussed in the previous section.

5.6 Interface

The genericity of our code-generator is technically expressed by turning the coder package into a functor. This functor takes a description of the executable language, i.e. a structure of signature LANG and produces a structure of signature CODER:

```
functor CodeGen (l : LANG) : CODER
```

An executable language is characterised as follows:

```
signature LANG = sig
   val prog_thy  :  theory

   type code_fun       = term -> absy
   type code_templ     = code_fun -> code_fun

   val undef           : code_templ
   val apply           : code_templ
   val  apply_strict   : code_templ
   val equal           : code_templ
   val  if_then_else   : code_templ
   val prog            : code_templ
   val lib             : code_templ list
```

```
    val accom      : thm -> thm  (* tactic that attempts to convert
                                  * a thm into a prog *)
    val lib_thms   : thm list    (* programs in initial module *)
  end;
```

Here, prog_thy has to be instantiated with the theory containing the programming notions. Coder functions, i.e. functions of type code_fun, map Isabelle-terms to absy (which is the sum of SML's abstract data-types exp, rule,...). Now, code-templates are functions that take a coder and a term and map them to absy. The following example represents the code-template apply for the language FIX as abstractly described in section 5.5:

```
fun apply C (S $ T) =
    if class_of(type_of(S)) = "cpo" then
       Exp(FnExp[Rule{pat=unit,
           exp=AppExp{function= C S,argument=C T
       ...
    else raise CODER ("Non-executable application: ", S$T)
```

The result of the instantiation will be an SML-structure providing abstract data-types for module's. An initial module lib will be created from the lib_thms.

```
  signature CODER = sig
    type module
    exception CODER of string * term
    val lib        : module
    val coder      : thm * module -> module
    val import     : module list -> module
    . . .             (* export, SML-compiler-calls *)
  end;
```

coder checks if an offered theorem is really an program in the underlying code-scheme and, among other syntactic checks, makes sure that all used functions in the theorem are already implemented in the module.

5.7 Libraries

When using the Isabelle/HOL library with many functions concerning natural numbers, arithmetic, lists and options, one can define a code-templates in lib for each library function to be translated "hard-wired" into the target language.

For the language WFREC, an expression s + t of type nat can be translated directly $\mathbb{s} + \mathbb{t}$ of type int. For the languages FIX and LFP, however, one has to force both arguments $\mathbb{s}() + \mathbb{t}()$.

Of course, each such library template is an additional assumption over the SML-compiler and its implemented environment (take care with _-_!), and it is therefore desirable to avoid such templates and replace them by generated code themselves. The slot to do this is the lib_thms-section, where proven executable functions can be given that were used to define an own library-environment independently from the SML-environment. However, the trade-off between trustworthiness and efficiency can be scaled by the instantiation of the code-generator.

5.8 Tactical Support

For each theorem to be coded into a module, the function accom is called. This function can convert, for example, definitions for primitive recursive functions (that were represented in Isabelle internally by specialized recursors derived from each data-type declaration) into one of the program notions. Appropriate tactics can hence greatly enhance the coverage of a code-generator instantiation.

Of course, these tactics will sometimes fail and represent an additional filter. In our current implementation, it is for example the definition for div:

```
m div n ≡ wfrec (trancl pred_nat)
                (λf j. if j < n then 0 else Suc (f(j-n))) m
```

The reason is that our tactics fails to prove coherence: A small hands-on proof revealed that from this definition of div follows: n div 0 = Suc(ιx. True), and this is produced by the cut-operation (see section 3.1) that can not be eliminated in the case of division by 0!

5.9 Optimizations

Another way to use accom is to perform pre-compile optimizations. Having a powerful theorem-proving environment available, it is relatively easy to safely implement many optimization techniques known from compiler construction by tactical programming.

For FIX, for example, one can add a couple of strictness- and definedness-propagation rules that enable to replace standard-applications by strict-applications. This has sometimes a dramatic effect on the execution-speed.

6 Experimental Data

6.1 A Toy Example in Code-Generation

We will demonstrate the resulting ML-code for the fibonacci function in the laguages WFREC and FIX. We used "hard-wired" code-templates for the arithmetic operations.

The fibonacci-program in the language WFREC looks straight forward. The termination order pred_nat is the is the successor relation on natural numbers.

```
Prog wf_fib_fun x is
  fib x = if x = 0 then 0
          else if x = 1 then 1
               else fib(x-1) + fib(x-2)
main fib x
measure pred_nat
```

The corresponding generated code looks as expected. This is a result of the relatively trivial coding schema for WFREC programs.

```
fun wf_fib_fun x =
 let   fun fib x = if (x = 0) then 0
                   else if (x = 1) then 1
                        else ((fib (x - 1)) + (fib (x - 2)))
 in (fib x)
 end
```

The program in the language FIX looks more complicated. We have to lift total datatypes into flat cpo's and strict functions over them. This is done by the constructor lift and the function lift_f described in section 3.3. (A decent language support would substantially improve parsing and printing of these programs, of course; the following example is chosen for the sake of the demonstration and not intended for end-users):

```
Prog fix_fib_fun x is
  fib x = if lift_f (op =) x (lift 0) then (lift 0)
           else if lift_f (op =) x (lift 1) then (lift 1)
                else lift_f (op +) (fib^(lift_f (op -) x (lift 1)))
                                   (fib^(lift_f (op -) x (lift 2)))
main (fib x)
```

Not too surprisingly, the generated code is very hard to read due to the code delaying and forcing the evaluation:

```
fun fix_fib_fun x =
 let
   fun fib ()=
     fn x => if (x () = 0) then (fn () => 0)
             else if (x () = 1) then (fn () => 1)
                  else (fn () => ((fib ()) (fn () => (x () - 1)) ()) +
                                 ((fib ()) (fn () => (x () - 2)) ()))
 in ((fn () => ((fib ()) x)) ()
 end
```

The efficiency of the generated ML-code differs quite substantially. While the computation of (wf_fib_fun 35) takes 5 seconds, the computation of (fix_fib_fun (fn () => 35) ()) takes 337 seconds on a SPARC 10.

6.2 Code-Generation for Library-Theories

Our current implementation based on Isabelle94-7 can code all functions from the theory of Lists and Options (the used code-generater was based on templates for arithmetic functions). Since the generated code is very similar to the SML-List structure, there is no measurable difference in efficiency.[3]

7 Conclusion

We have presented a generic code generator as extension to HOL-based implementation environments used for formal methods. The generator attempts to be as close as possible to the guidelines of LCF-style prover design, in the sense that it allows to minimize the set of extra-logical assumptions. The generic generator is instantiated for three program notions, ranging from executablility in the style of HOL, Z or (strict or lazy) functional programming languages.

---

3 We would like to test our implementation with more convincing program examples. Unfortunately, most papers on benchmarks for functional programing languages we know of involve arithmetic on real numbers. Hints welcome!

Of course, the underlying concepts and theories for our program notions are by no means new, and can not be. Neither, our list of program notions is far from beeing complete — for HOL-CSP [TW 96], for example, further execution notions incorporating different forms of non-determinism and concurrency are imaginable, and for Regensburgers HOLCF [Reg 94], yet another instantiation (similar to FIX, but different in the details like its treatment of application) can be conceived. Comes to think of it, there is a real plethora of execution notions, and we think it is necessary to have a technical framework for their implementation and to control their correctness respectively their trustworthyness. From our perspective, it is not fully understandable why code-generation is traditionally treated as a side-issue; bugs in a code-generator are as damaging for the overall correctness than bugs in the logical engine of a prover.

Related Work

The work of Slind [Sli 96] presents a pure front-end approach to the representation of programs on the level of the theory-files. The user can define functions in a very rich and compact functional notation employing powerful pattern-matching, that is parsed away into a WFREC-style semantic representation when the input-file is loaded. As a consequence, it is not possible to *derive* programs during theorem proving, which was one of our major goals.

The work of [LL 91] and similarly [Nei 92] represent a quite classical approach to characterize an executable subset of a specification language, VDM in this case. In contrast to our approach, they avoid recursion notions on the object-logical side and have to live with additional meta-logical (and unproven) assumptions.

In [BB 93], a bridge from code-generation to specification animation is built. Here, the idea is to represent sets in Z-specifications (corresponding exactly to sets in HOL) by enumeration functions, that produce the elements of a set (*animates* it) one after another. In this view, set comprehensions are a powerful functional notation for sorted lists. Of course, the difference to the Z standard semantics is discussed.

Future Work

We will improve the portability of the code-generator to other SML-Compilers (such as POLY or Harlekin) and, to a lesser extent, to other SML-based theorem prover environments like LAMBDA or HOL/HOL. Moreover, our actual ad-hoc treatment of data-types[4] should be replaced by a more general mechanism, possibly better integrated in a future version of Isabelle.

It is worth investigating to increase the genericity with respect to the target language: It should not be too difficult to develop other converters from `absy` (or other, more general intermediate abstract syntaxes) to languages like C++, Java or Haskell; for the latter, a code-scheme could be conceived supporting some type classes of Isabelle.

Another line of extension is the development of executable languages including state- or exception based computations.

---

4 In our current implementation, the data-type information is extracted from the *.thy files...

References

[And 86] P.B. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[AF 97] S. Agerholm, J. Frost: Towards an Integrated CASE and Theorem Proving Tool for VDM-SL. In: J. Fitzgerald, C.B. Jones, P. Lucas (Eds): FME 97. LNCS 1313, 1997.

[LL 91] P.G. Larsen, P.B. Lassen: An executable subset of Meta-IV with loose specification. In Prehn, Toetenel (eds): Formal Software Development Methods, Vol 1. LNCS 551, 1991.

[Nei 92] G. O'Neill: Automatic Translation of VDM Specifications into Standard ML programs. The Computer Journal, 35(6):623-624,1992.

[BB 93] P.T. Breuer, J. P. Bowen: Towards Correct Executable Semantics for Z. Available online.

[Chu 40] A. Church: A formulation of the simple theory of types. Journal of Symbolic Logic, 5, 1940, pp. 56-68.

[Isabelle] The Isabelle-documentation page: www4.informatik.tu-muenchen.de /~nipkow/isabelle. Alternatively: http//www.cl.cam.ac.uk/Research- /HVG/isabelle.html.

[KSW 96b] Kolyang, T. Santen, B. Wolff: A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy and J. Harrison (eds): Theorem Proving in Higher/Order Logics — 9th International Conference, LNCS 1125, pp. 283-298, 1996.

[KSW 96b] Kolyang, T. Santen, B. Wolff: A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy and J. Harrison (eds): Theorem Proving in Higher/Order Logics — 9th International Conference, LNCS 1125, pp. 283-298, 1996.

[Pau 91] L. C. Paulsen: ML for the Working Programmer. Cambridge University Press, 1991.

[Reg 94] F. Regensburger: HOLCF: Eine konservative Einbettung von LCF in HOL. Phd thesis, Technische Universität München. 1994.

[TW 97] H. Tej, B. Wolff: A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. In: P. Lucas (ed): FME 97. LNCS 1315.