Correct Code-Generation in a Generic Framework

Burkhart Wolff[1] and Thomas Meyer[2]

[1] Institut für Informatik, Albert-Ludwigs-Universität Freiburg
wolff@informatik.uni-freiburg.de
[2] Bremen Institute of Safe Systems (BISS), FB 3, Universität Bremen
tm@informatik.uni-bremen.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 a framework to describe such notions, a method to formally investigate them and instantiate it for three executable languages, 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

This paper is concerned with the combination of specification notations and program notions, or more precisely, data-view oriented specification formalisms and functional programming languages. While many specification formalisms (such as B, KIV/VSE, but also COQ [3, 5, 2]) come with a built-in notion of program or executability, formalisms like CASL, Z or HOL[1, 20, 15] do not. For them, there is a need to fill the gap to programs and their verification — in fact, our original motivation for this work has its roots in our interest in generic formal transformational development and its implementation in the TAS-system [13].

Still, the question arises, why not simply using a specification formalism with a built-in program notion? We see three major reasons for this: First, there are several choices in setting up a program notion — why should a general purpose language like HOL impose just one? Second, with a flexible technique of integrating *arbitrary* program notions, one can choose one very close to a particular programming language paving the way for the verification of language-specific optimisations. Third, we are concerned with the correctness of the compilation; in the past, many code-generators of tools with built-in program notions turned out error-prone, indicating that the task has been underestimated.

Thus, we are interested in a *method* to establish and formally investigate concrete program notions, together with their compilation to code. Our program notions will have the property that the question "is it a program?" – often deliberately left to a non-formalised meta-level – is decidable. Since code-generators in formal development environments serve as a bridge between a theorem prover and the outside world, it is not completely possible to *verify* a code-generator inside a logic; the proof must be extended by meta-logical arguments, which are preferably as *minimal* and *simple* as possible. Finally, we are interested in a technical *framework* supporting the method, providing a prototypical evaluator inside the logic and a code-generator for external code that is intended to produce the same result as the evaluator. The technique should support datatype and higher-order function declarations.

We provide a formal and (in its crucial parts) formally provable method for defining and investigating *program notions* and such a technical framework for *correct code-generation* inside HOL. The choice for HOL – being largely equivalent to Z (see [18] for details) – is motivated by its greater flexibility to embed other formal methods, such that there is a greater potential for reuse of this work for these embeddings. In this paper, we will exemplify this framework for three different program notions and its code-generation for languages (and compilation schemes) with increasing semantic complexity. The first language is the language of well-founded recursion *Wf* without exceptions; this language is particularly interesting since many library definitions in various HOL-systems can thus be converted into code yielding implementations for end systems or animations of specifications. The second language *Fix* is a call-by-name functional programming language with (one) exception – here, the subtle issues between lazy and eager evaluation have to be covered during the code-generation. Third, we use the relational language *Lfp*; this language is designed as an executional fragment of the specification language Z (see [20], [12]).

On the technical side, our framework is supported by a generic implementation in form of an SML functor that allows the reuse of common functionality in code-generators. As target language, we chose SML. The motivation for this choice is both conceptual and pragmatic. Conceptually, SML has the advantage of a formally defined semantics [14] that can be used (although in a massaged form) as a basis for correctness proofs. Pragmatically it is most compelling to have an SML code-generator since most HOL implementations are based on SML. In particular, we picked Isabelle/HOL as our implementation basis.

2 The Conceptual Framework for Correct Code-Generation

2.1 Fundamentals

The following diagram may illustrate the basic concepts of our work in more detail. Here, with *logic* we denote the set of terms of our programming or specification language – in our case, this will be higher-order logic (HOL) including some conservative extensions such as library theories or specification language encodings. A subset of these terms will be *abstract programs* which again will be a superset of the *abstract values*.

The logic is mirrored by the corresponding sets of (concrete) *programming language* terms and its subset of (concrete) *values*. Both worlds are connected by the code-generation function *convert* that is required to be total on the domain of abstract programs and has the term set *code* as range.

The two relations $\rightarrow_A$ and $\rightarrow_C$ represent the operational semantics of the two languages; we require that they represent partial functions from programs to values. We define a *program notion* as a configuration consisting of a concrete set of abstract program terms and abstract value terms, a set of target programs and concrete value terms, the two fixed evaluation relations $\rightarrow_A$ and $\rightarrow_C$ and the function *convert*. If all components of one program notion include all components of another, we will say that the former has a greater *coverage* than the latter (we owe this terminology to [6]). We will call the code-generation for a program notion *correct* if the above diagram commutes, i.e. iff $convert((\rightarrow_A)t) = (\rightarrow_C)(convert\ t)$. Note that both $\rightarrow_A$ and $\rightarrow_C$ should be undefined for the same t. Provided we have a correct program notion, we can thus convert an abstract program into code and compute the value of program terms outside of our theorem prover, which can be significantly more efficient and should be just the final product of a formal program development we are interested in.

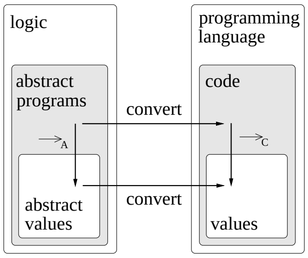

Fig. 1. Basic Concepts

2.2 Correct Code-Generation: A Critique

There are some fundamental problems with the provability of correctness in the sense of the previous section. Since code-generators serve as a bridge between a theorem prover and the outside world, it is not completely possible to *verify* a code-generator inside a logic; the proof must be based on meta-logical arguments. 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 propose two technical design principles to increase trustworthiness of code-generation:

- *convert* should be implemented as a primitive, one-to-one converter
- from the definition of $\rightarrow_A$, a number of theorems should be derived that mirror the rules of $\rightarrow_C$ (as given in the definition of our language) syntactically.

Of course, this syntactic correspondence of the rules of $\rightarrow_A$ and $\rightarrow_C$ does not formally guarantee that they are identical because its symbols are interpreted in different contexts. Moreover, correspondence between $\rightarrow_A$ and $\rightarrow_C$ does not guarantee the correct implementation of $\rightarrow_C$ (i.e. compiler correctness with respect to $\rightarrow_C$ – its specification – is assumed throughout this paper). Still, both design principles together force a shift of many compilation oriented activities in the overall translation into tactic theorem proving based on derived rules and thus into the safe core of an LCF-style theorem prover.

3 Preliminaries: SML, *AbstractSML* and Isabelle

Clearly, the direct use of the formal operational semantics of SML described in [14] would be most convincing for our goal of bridging the gap between a logic and a programming language. Unfortunately, since SML is a very rich language, a presentation of the formal semantics is still too complex even if restricted to the relevant language fragment; this is mostly due to the fact that SML contains imperative constructs which are out of the scope of this paper. Instead, we use the more "vanilla" operational semantics of an eager language following closely [22]. We consider the task of bridging the gap between "Real-SML" in the sense of [14] and our language – which we call *AbstractSML* – as routine.

3.1 Expressions of *AbstractSML*

The key ingredient of this operational semantics is an inductively defined subset of all terms, the so called set of *canonical forms.* The judgment $t \in C_\tau$ states that t is a canonical form of type τ:

Ground type:	$C_{int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and $b \in C_{bool} = \{true, false\}$
Product type:	pairs of canonical forms are canonical, i.e. $\langle t_1, t_2\rangle \in C_{\tau_1 * \tau_2}$ if $t_1 \in C_{\tau_1}$ and $t_2 \in C_{\tau_2}$
Function type:	closed abstractions are canonical forms, i.e. $(fn\ x \Rightarrow t) \in C_{\tau_1 \to \tau_2}$ if $(fn\ x \Rightarrow t) : \tau_1 \to \tau_2$ and t closed

We can now give the rules for the evaluation relation of the form

$$t \to_C c$$

where t is a typable term and c is a canonical form, meaning t evaluates to c. In the following, c, c_1, c_2 and c_3 range over canonical forms.

$$c \to_C c \qquad \text{(identity)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{t_1 \ \mathbf{op}\ t_2 \to_C c_1 \ \mathbf{op}\ c_2} \qquad \text{(operations)}$$

$$\frac{t_1 \to_C true \quad t_2 \to_C c_2}{\mathbf{IF}\ t_1\ \mathbf{THEN}\ t_2\ \mathbf{ELSE}\ t_3 \to_C c_2} \qquad \text{(ite-true)}$$

$$\frac{t_1 \to_C false \quad t_3 \to_C c_2}{\mathbf{IF}\ t_1\ \mathbf{THEN}\ t_2\ \mathbf{ELSE}\ t_3 \to_C c_2} \qquad \text{(ite-false)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\langle t_1, t_2\rangle \to_C (c_1, c_2)} \qquad \text{(product)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\mathbf{fst}\langle t_1, t_2\rangle \to_C c_1} \qquad \text{(fst)}$$

$$\frac{t_1 \to_C c_1 \quad t_2 \to_C c_2}{\mathbf{snd}\langle t_1, t_2\rangle \to_C c_2} \tag{snd}$$

$$\frac{t_1 \to_C \mathbf{fn}\ x \Rightarrow t \quad t_2 \to_C c_2 \quad t[x := c_2] \to_C c}{t_1\ t_2 \to_C c} \tag{function}$$

$$\frac{t_1 \to_C c_1 \quad t_2[x := c_1] \to_C c}{\mathbf{LET}\ x = t_1\ \mathbf{IN}\ t_2 \to_C c} \tag{let}$$

$$\begin{gathered}\mathbf{REC}\ y.\ (\mathbf{fn}\ x_1 \Rightarrow \ldots x_n \Rightarrow t) \to_C \\ \mathbf{fn}\ x_1 \Rightarrow \ldots x_n t[y := \mathbf{REC}\ y.\ (\mathbf{fn}\ x_1 \Rightarrow \ldots x_n \Rightarrow t)]\end{gathered} \tag{rec}$$

With these rules, we also implicitly introduce the syntax of our target language.

3.2 Declarations in *AbstractSML*

In order to scale up to larger units, a programming language comprises mechanisms to extend the set of initially defined operator symbols, also called the *basic environment*, by user defined ones. We will make this more precise and formally define an *environment* Γ as a set of *type judgments* $x{::}\tau$ assigning to an identifier x a type τ. For *AbstractSML*, the base environment Γ_0 comprises judgments such as $-2 :: int, true :: bool$ or $+ :: int * int \to int$. A term t *conforms to* Γ, if all free identifiers in t are declared in Γ and if t is typable. The effect of a declaration:

$$\mathbf{VAL}\ x = t \tag{ValDec}$$

is to extend the implicit environment Γ (on which we make the assumption that t must conform to Γ and therefore have some type τ) to the environment $\Gamma' = \Gamma \backslash \{x :: \_\} \cup \{x :: \tau\}$. Moreover, the evaluation relation $\to_C$ is extended by the rule:

$$\frac{t \to_C c}{x \to_C c} \tag{Unfold}$$

For datatypes, we proceed analogously. The declaration

$$\mathbf{DATATYPE}\ s = \mathbf{con}_1 :: \tau_1 \mid \ldots \mid \mathbf{con}_{1_n} :: \tau_n \tag{TypeDecl}$$

produces the new environment $\Gamma' = (\Gamma \backslash \{\mathbf{con}_j :: \_\}) \cup \{\mathbf{con}_j :: \tau_j\}$. For a legal datatype declaration, we assume s to be a fresh type and the τ_i to have either the type s or $\tau_i' \to s$. Moreover, Γ' will be extended by an additional constant symbol $CASE_s$ (case distinction) for which the evaluation relation is extended by rules of the following scheme:

$$\frac{f_i \to_C c}{\mathbf{CASE}_s\ \mathbf{con}_i\ f_1 \ldots f_n \to_C c} \tag{CaseMatchCon}$$

$$\frac{t \to_C c_1 \quad f_i\ t \to_C c_2}{\mathbf{CASE}_s\ (\mathbf{con}_i\ t)\ f_1 \ldots f_n \to_C c_2} \tag{CaseMatchFun}$$

Note that we also introduce a new set of canonical forms C_s.

This concludes the definition of the "dynamic" part of our target language *AbstractSML*. From here, we can already foresee the major technical requirements for our code-generator: It must be

- able to deduce the underlying datatypes from a sequence of input theorems
- find a sequence of the input theorems that corresponds to a series of declarations
- able to convert each of the input theorems to declarations, i.e. check that the expressions on the right hand side of the declaration are indeed an *AbstractSML* expression.

Below, we turn to the logical environment (implemented in SML) in which our programming languages will be represented and in which the non-trivial conversion into code will be performed.

3.3 Isabelle

Isabelle [10] is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fr"ankel 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).

Isabelle belongs to the family of LCF-style theorem provers. This essentially means that the abstract data type "thm" (protected by the SML type discipline) contains all the formulas accepted by Isabelle as theorems. thm-objects can only be constructed via operations of the logical kernel of Isabelle. This architecture allows for user-programmed extensions of Isabelle without corrupting the logical kernel.

In the sequel, all Isabelle input and output will be denoted in this `font` throughout this paper. For the mathematical symbols $\forall$, $\exists$, $\wedge$ and $\vee$ we use the Isabelle notations `!`, `?`, `&` and `|`.

3.4 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 [7]; a more recent presentation is [4]. HOL is a classical logic with equality formed over the usual logical connectives $\neg, \wedge, \vee, \Rightarrow$ 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.

Most HOL-systems were used in a particular methodology: Since adding arbitrary axioms to a basic logical system like HOL is extremely untrustworthy, these systems support particular schemes of axioms – so called *conservative extensions* – that ensure consistency when building up larger libraries (see [15]). Following common usage, we will use the term HOL-theory also for all its conservative extensions.

4 The Generic Framework

In this section, we will explain the overall structure of our generic coding scheme in more detail. The technical aspects of our generic coder are discussed later in section 5. As a starting point,we will refine the diagram of Fig. 1 and develop a separation of our coding scheme into different phases.

In accordance to Fig. 1, we subdivide *abstract programs* into two languages $\langle X\rangle Lang$ and $\langle X\rangle AbstractSML$ – here, X stands as a placeholder for concrete program notions (such as *Wf*, *Fix*, *Lfp* to be discussed later). $\langle X\rangle Lang$ is the language that is the source of the coding process. Since we instantiate our target language with SML throughout this paper, we use $\langle X\rangle AbstractSML$, which is the semantic interface to our target language SML. The abstract operational semantics $\rightarrow_C$ must only be established for $\langle X\rangle AbstractSML$. In such a setting, the coding process can now be described as a sequence of six compilation phases. Except for the last one, the phase *convert* already discussed in the introduction, they consist of tactics based on derived rules. In more detail, the five phases are:

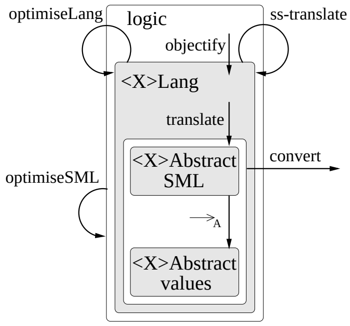

Fig. 2. The redefined coding scheme

- *objectify* attempts to convert an arbitrary term of our logic in a term of our programming language. This phase is the key-ingredient for increasing the coverage of the coder.
- *optimiseLang* is used for language-dependent compiler optimisations, e.g. data type dependent rules like the associativity of concatenation of lists, which improves the efficiency of the evaluation since `(a @ b) @ c` is usually more expensive than `a @ (b @ c)`.
- *ss-translate* is a source-to-source translation preparing the next phase called *translate*. We allow *ss-translate* to produce terms that contain language constructs belonging to the $\langle X\rangle AbstractSML$-language.
- *translate* maps (impure) $\langle X\rangle Lang$-terms to $\langle X\rangle AbstractSML$.
- *optimiseSML* can be used as code-optimisation on the *AbstractSML*-level.

In the next chapter, we will describe three instantiations of our generic scheme mapping X to *Wf*, *Fix* and *Lfp*, representing a program notion for total functions, functional programming and logic programs.

5 The Instances `WF`, `FIX` and `LFP`

We are now ready for our main task, the representation of three different programming languages in HOL, their proof of correctness in our sense, and the development of derived rules and tactics that perform the major part of the coding.

We will demonstrate our technique in most detail for the first language, which is also the simplest one. In the theory `WFLang`, besides the syntax, we define the semantics of the operators and language constructs of our language in terms of HOL. Subsequently, in the theory `WFSML`, the semantics of language constructs of SML in terms of `WFLang`. Due to the deliberate simplicity of `WF` (that is designed to form a reasonable subset of HOL-formulas that should be converted into SML-code), these two language mappings are mostly trivial, resulting in a correctness proof (i.e. a derivation of $\rightarrow_C$ that is extremely simple).

The language mappings in the subsequent instances will contain more and more complexity demonstrating the flexibility of our concepts. `FIX` will introduce exception and recursion partiality into the language, while `LFP` extends it by some controlled form of backtracking. Intuitively speaking, the "distance" expressed in the mapping between $\langle X \rangle Lang$ and $\langle X \rangle SML$ grows during the sequence of these instantiations.

5.1 The instance `WF`

The theory `WFLang` is based on HOL-theories providing basic semantics for boolean and numerical operators:

```
WFLang = WF + ... +
consts
(* library of basic operations *)
TRUE :: bool
...
ZERO :: nat
...
```

Analogously, we declare the operator symbols `FALSE`, `NOT`, `AND`, `OR`, `ONE`, `TWO`, `SUC`, `LEQ`, etc. These constant symbols represent the sets of canonical forms C_{bool} and C_{nat}. Note that the constant ϵ `x.True` (also called `arbitrary`) cannot be included in the set of canonical forms since the Hilbert-Operator is interpreted by definition differently in any model of an HOL-formula; correct code, however, will have to produce values that exist in all models. On top of the denotations for canonical forms, we will now define the operators of our language such as:

```
LESS :: [nat,nat] -> bool
        LESS a b = a < b
...
```

As a consequence of the fact that `arbitrary` is not a canonical form, we must rule out partial operators like `div` or `hd` (on lists) from the language `WF`. A special case for the operators is the equality, which is declared polymorphically for all canonical terms, but has to be constrained to a class `EQ` of base types and cartesian products over them, ruling out the function types, for which $\rightarrow_C$ cannot be used to compute `EQ` and is therefore also ruled out in SML:

```
EQ :: ['a:EQ,'a] -> bool
```

The definition of these constructs one-to-one corresponds to the HOL operations, as a consequence of the design of `WF`:

```
TRUE_def  "TRUE = True"
...
ZERO_def  "ZERO = 0"
```

Next, we define the basic datatypes `unit` and `pair`: the operators `UNIT`, `PAIR`, `FST`, and `SND` that were trivially represented by their HOL-counterparts (), `(_,_)`, `fst` and `snd` respectively.

Now may now turn to the core of `WFLang`, i.e. the main language constructs. Again, the constant definitions are straight-forward mappings to standard operations:

```
^!       :: ('a => 'b) => 'a => 'b
            f ^! x     == f x
Lam      :: ('a => 'b) => ('a => 'b)
            Lam f      == f
IF       :: [bool, 'a, 'a] => 'a
            (IF a THEN b1 ELSE b2) == (if a then b1 else b2)
LET      :: ['a, 'a => 'b] => 'b
            LET s f    == f ^! s
REC      :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
            REC(m)(f) == wfrec m f
```

The characterising feature of each program notion is the notion of recursion, i.e. some instance of the general scheme:

$YF = F(YF)$, where F must fulfill some requirement A

The "workhorse" for most definitions of total functions in the library of HOL is the well-founded recursion. Even the definitions of primitive recursive functions such as concatenation on lists is internally mapped to well-founded recursion. Thus, it is suggestive to define REC_m (parameterized by a well-founded ordering m) in `WFLang` by

```
wfrec :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
```

developed in the theory `WF` in the library in Isabelle/HOL. The main result of the theory of well-founded recursion is:

```
wf(r) = wfrec r H a = H (cut (wfrec r h) r a) a                [wfrec]
```

where the predicate `wf : ('a * 'a)set -> bool` states the well-foundedness of a relation 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. ϵ`x.True`) for all larger values.

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

```
!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. Well-foundedness and coherence together establish the desired fixpoint property for `wfrec` along `[wfrec]`. The problem with our representation of REC_m is that we need well-foundedness and coherence, i.e. additional semantic information for each occurance of REC_m in an abstract program assuring that the fixpoint property holds – this problem will reappear in different form in our other program notions. This leads to the definition of a kind of SML-like statement that contains the code plus the semantic information necessary to establish the fixpoint property of the recursor:

```
WFProg :: [('a * 'a)set, 'c, (('a => 'b) => ('a => 'b)),
           ('a => 'b) => 'c] => bool
WFProg m f F E == (f = (let x = (wfrec m f) in (E x)
                  & (wf m)
                  & (!a. cut(wfrec m F) m a = (wfrec m F))))
```

The following syntactic sugar paraphrases this complex definition as an SML-like statement annotated with semantic information:

```
val f = let fun F in E measure m
```

This completes the definition of `WFLang`. We turn now to our semantic interface to SML, called `WFSML`, which is defined as a theory extension of `WFLang`.

```
WFSML = WFLang +
```

and provides definitions of operators `TRUE'`, `FALSE'`, `NOT'`, ..., `ZERO'`, `ONE'`, ..., `LESS'`, ..., `UNIT'`, `PAIR'`, ..., all defined identical to their unprimed counterparts from `WFLang`. Here, we only show the definitions for the core language constructs:

```
^!'      :: ('a => 'b) => 'a => 'b
            f ^!' x == f ^! x
Lam'     :: ('a => 'b) => ('a => 'b)
            Lam' f == f
IF'      :: [bool, 'a, 'a] => 'a
            (IF' a THEN' b1 ELSE' b2) == (IF a THEN b1 ELSE b2)
LET'     :: ['a, 'a => 'b] => 'b
            LET' s f == LET s f
REC'     :: ('a * 'a)set => (('a=>'b) => ('a=>'b)) => 'a => 'b
            REC' == REC
```

As a next step, we show the correctness of our language representation for SML, i.e. that we can derive the operational semantics of *AbstractSML* rules (in the sense of chapter 2). First, we define the evaluation relation $\rightarrow_A$ by the semantical equality:

```
WFSML_CT = WFSML +
constdefs     eval     :: ['a,'a] => bool
              eval s t == (s = t)
syntax        -A->     :: ['a,'a] => bool
translations  s -A-> t == eval s t
end
```

The predicate `cf c` (`c` is a canonical form) is simply set to `true` in `WF` since we leave this check to the meta-level. Now we derive the operational semantics:

```
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         (LESS' t1 t2) -A-> (c1 < c2)
[| cf c2; t1 -A-> TRUE'; t2 -A-> c2 |] ==>
         (IF' t1 THEN' t2 ELSE' t3) -A-> c2
[| cf c2; t1 -A-> FALSE'; t3 -A-> c2 |] ==>
         (IF' t1 THEN' t2 ELSE' t3) -A-> c2
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         PAIR' t1 t2 -A-> (c1,c2)
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         FST'(PAIR' t1 t2) -A-> c1
[| cf c1; cf c2; t1 -A-> c1; t2 -A-> c2 |] ==>
         SND'(PAIR' t1 t2) -A-> c2
[| cf c; cf c2; t1 -A-> (LAM' x. t x); t2 -A-> c2;
         (t(c2)) -A-> c |] ==>
         (t1 ^!' t2)  -A-> c
[| cf c; cf c1; t1 -A-> c1 ; (t2(c1)) -A-> c |] ==>
         LET' t1 (%x. t2 x)  -A-> c
[| ! a. cut (wfrec m f) m a = (wfrec m f); wf m |] ==>
         REC m (% X. (LAM' x1. f X x1)) -A->
         (LAM' x1. f (REC m (%X. (LAM' x1. f X x1))) x1)
```

and we are home and dry! By syntactical correspondence, we check that our derived formal rules for $\to_C$ correspond to the rules $\to_C$ in chapter 3.1.

The remaining steps are merely technical: First, the code-generator has to be set up to generate the definitions for the implicit `CASE'`-rules (cf. section 3.2) and to generate code for datatypes including the involved recursors. Second, we have to describe the compilation phases in the sense of chapter 4. For `WF`, these phases are fairly trivial – for `ss-translate` and `optimiseSML` we use just the identity and for `optimiseLang` just a simplification tactic for some set of equations like associativity on lists. For `translate` we use a simplification tactic that folds all definitions of `WFSML` from right to left. In this setting, the preparational `objectify` is the most complex phase. We use it to convert equations from primitive recursive definitions like the following for the concatenation of lists (drawn from the Isabelle/HOL-library):

```
primrec "op @" list
  [] @ ys   = ys
  (x#xs)@ys = x # (xs @ ys)
```

into the term of the abstract programming language:

```
val (op @) = let fun f x ys = CASE x OF
                                []  => ys
                              | (x # xs) => x # (f ^! xs ^! ys)
             in f measure length
```

Constructing this representation also requires reasoning over the internal representations of datatypes and subterm orderings used in Isabelle/HOL's datatype package.

5.2 The instance `FIX`

The language `WF`, constrained to total functions, had to rule out values like `hd []` or `3 div 0`. When admitting partial functions, there is the well-known choice for the semantics of function application reflecting *call-by-value* evaluation or *call-by-name* evaluation (cf. [22]) . An appropriate semantical framework for tackling these issues is denotational semantics, which we use as basis for our second program notion `FIX`.

There are many known formalisations of denotational semantics in HOL-systems. For Isabelle/HOL, there is most notably HOLCF [17]. Instead, we will use the generic theory of Scott-cpo's `Fix.thy` described in [21]. Both theories have much in common and could be exchanged in this context with minor effort; we preferred our own version mostly due to its lightweightness.

In the following, we briefly review `Fix.thy` and its pivotal definitions. cpo's are introduced by a sequence of *axiomatic class* definitions, i.e. an extension of the Haskell class system with semantic constraints in Isabelle. For instance, the class of partial order types `order` is defined by the statement:

```
axclass order < 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 $\perp$, directed sets, upperbounds, least upperbounds etc., the class `order` is extended to the class `cpo`. In this class, the predicate for continuity `cont::('a::cpo -> 'b::cpo) -> bool` and the fixpoint operator `fix::('a::cpo -> 'a) -> 'a` (defined as least upperbound of the directed set of function iterations) were defined and provide as main result:

```
cont f => fix f == f (fix f)                         [knaster-tarski]
```

which will give semantics to our recursor `REC` in our program notion `FIX` defined in `FixLang.thy`:

```
FixProg f F == f = fix F & cont F
```

Having settled the fundamental questions, we turn now to the basic datatype representations `bool` and `nat`. In order to embed them into cpo's, we employ the well-known lifting technique into flat domains by defining the type constructor:

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

After identifying `down` with $\perp$ 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, pairs and function spaces are shown to be instances of `cpo` provided that both arguments resp. the last argument of the type-constructors are instances of `cpo`. The basic types `Bool`, `Nat` and `Unit` in `FixLang` were defined by lifting the underlying corresponding types of the HOL-library via `up`. The definitions of the basic operations are straight-forward as strict extensions of the underlying HOL-operations. In contrast to `WF`, however, we can define partial functions analogously to `DIV`:

```
constdef DIF :: [Nat, Nat] -> Nat
         DIV == strictify(%x::nat. strictify(%y.
                                       if y=0 then UU
                                       else lift(x div y)))
```

The definition of the language constructs of our functional language `FixLang` is also straight-forward in denotational terms of our cpo-theory `Fix`. As example, we only show the application, that is directly mapped to the HOL-application since we already have proven that the standard function space has cpo-structure:

```
constdef ^!     :: ('a -> 'b::cpo) -> 'a -> 'b
         F ^! x == F x"
```

At this point, we conclude our presentation of `FixLang` and turn to the semantical interface `FixSML` of our call-by-value target language. Here, a crucial point is an adequate denotational representation of abstractions that were treated as *closures* in the operational semantics. In order to distinguish `LAM x.`$\perp$`::'b`, that is equal to the least element in the function space $\perp$`::'a`$\rightarrow$`'b`, from `LAM' x.`$\perp$, that is the *closure* of the computation yielding $\perp$ (as in SML) and hence a canonical form or a value, `LAM'` must lift any function (see also [22], pp.188). Thus, it is suggestive to introduce an own type constructor for the lifted function space $\rightarrow!$ and define:

```
FixSML = FixLang +
types  ('a,'b) "=>!" = "('a => 'b) up"
...
constdefs
  ^!'       :: ['a=>'b::cpo,'a] => 'b
  F ^! x    == if x = UU then UU
                         else if F = UU then UU
                              else (drop F) x
  Lam'      :: ('a::cpo =>'b::cpo) => ('a =>! 'b)
  Lam' f    == lift f
  ...
```

where drop is just the inverse to lift. The definitions for IF', LET', REC' etc. are straight-forward and omitted together with the mappings of the FixLang-operators to the FixSML-operators, which are just appropriate liftings w.r.t. $\to$!.

The key question for the code-generation in FIX is the translation of the call-by-name versions of Lam and ^! to their call-by-value counterparts Lam' and ^!' respectively (for pairing and projection, the situation is similar). The key for a solution is the definition of the suspension resp. the forcing functions (see also [9]):

```
delay      :: 'a::cpo => 'a del
delay f    == (LAM! x. f)
force      :: ('a::cpo)del => 'a
force f    == (f ^! UNIT)
```

where 'a del is a type synonym for Unit ->! 'a. Note that delay and force are already "pure SML" and can thus be converted easily. These definition leads to the following derived theorem, that allows the exchange of all lazy applications by eager ones:

```
(f ^! a) = ((forcify f) ^!' (delay a))                    [lazy2eager]
```

where forcify is defined by LAM! x. f (force x)). The translation is possible by using this rule in *all* applications in FixLang; however, this technique leads to quite inefficient code. A remedy to this problem – well known from compiler-construction [9] – is a strictness-analysis that we mimic in our approach by the following derived rules:

```
is_strict f ==> (f ^? a) = ((lift f) ^! a)
is_strict(LAM! x. x)
is_strict(%x. UU)
is_strict(strictify f)
is_strict(NOT)
is_strict(SUC)
[| !a. is_strict f |] ==> is_strict (%x. f^! x ^! a)
[| !a. is_strict (f ^! a) |] ==> is_strict ((lift f) ^!' a)
is_strict f ==> is_strict (%x. (IF (f x) THEN (g x) ELSE (h x)))
...
```

where is_strict f is defined by f $\perp$=$\perp$. The first rule of the list above represents our optimised translation, that requires strictness, while the other rules try to establish this property (note that the list is incomplete). For all applications, where this did not succeed, the rule lazy2eager is applied.

We now briefly describe the overall technical organization of the coding in phases: *objectify* maps applications, abstractions, and constructs like if_then_else from HOL to *FixLang*-terms. *optimiseLang* and *optimiseSML* are again set to identity. The translation from ^! to ^!' (as described above) is a classical source-to-2-translation and goes to *ss-translate*. Finally, *translate* is used to map basic operators from *FixLang* to *FixSML*. A little example may illustrate the steps in more detail:

```
((% x y. y) (DIV ONE ZERO) TWO)
```

$\Downarrow$ {*objectify*}

```
(LAM!x y. y ^! (DIV ^! ONE ^! ZERO) ^! TWO)
```

$\Downarrow$ {*ss_translate*}

```
(LAM!'x. lift (LAM!y. y) ^!'
          delay (lift(lift DIV ^!' ONE) ^!' ZERO') ^!' TWO)
```

$\Downarrow$ {*translate*}

```
(LAM!'x y. y ^!' delay (DIV' ^!' ONE' ^!' ZERO') ^!' TWO')
```

The function `%x y. y` is strict in its second, but not in its first argument. Hence, the evaluation of the first argument must be delayed – which happens to be undefined in this example. For the second argument, no suspension is needed and can be avoided for efficiency reasons.

Finally, we turn to the question of correctness of this coding scheme. We define the relation $\to_A$ in `FIXSML_CT` analogously to `WFSML_CT` (see previous section) except that we define canonical forms `cf` as "not being $\perp$". Thus, we consider cases like `DIV ONE ZERO` as an *exception*. In `FIXSML_CT`, we derive all operational rules of section 5.1. (although they are based for a completely different semantic interpretation). Additionally, we also have operational rules that cover exceptional behaviour:

```
[| ~cf c1; t1 -A-> c1; t2 -A-> X |] ==> (t1 ^!' t2) -A-> UU
[| ~cf c2; t1 -A-> X; t2 -A-> c2 |] ==> (t1 ^!' t2) -A-> UU
```

Although these rules do not appear in [22], they can be justified by the *exception convention* (cf. [14], pp. 40) in the SML-standard. With this proof of correctness, which is still fairly simple in Isabelle/HOL but no longer trivial as in *Wf*, we conclude the presentation of the coding scheme for *Fix*.

5.3 The instance LFP

The language `LFP` is inspired by the semantic embedding of Z in HOL (see [12]) and previous work on animation tools for Z-specifications (see [6]), [8]). Z is based on a typed set-theory – as available in HOL – and represents all functions by their graph, i.e. a set of pairs:

```
LfpLang = Set + Lfp + EqnSyntax + Arith +
types ('a,'b) "<=>" = ('a*'b) set
...
LAM     :: ['a => 'b]   => ('a <=> 'b)
%^      :: ['a<=>'b,'a] => 'b
...
```

This results in a formulation of partial functions that is similar to *Fix* with respect to the necessary conversion between call-by-name-semantics in `LfpLang` to call-by-value semantics in `LfpSML`. In the setting of `LFP`, the recursor `REC` is based on the usual least fix-point `lfp::['a set -> 'a set] -> 'a set` enjoying the property:

```
mono(f) => lfp(f) = f(lfp(f))                         [lfp_Tarski]
```

which is already established in the Isabelle theory `LFP` in the library. This is exactly what we want for our program statements which we define as follows:

```
LfpProg f F == f = lfp F & mono F
```

Although the semantic foundation – as outlined above – is totally different, the coding machinery is similar to the one described in `FIX`. Hence, we will refrain from a further formal presentation of this program notion and concentrate on the new aspects here. In this case, there is a new additional basic datatype `'a set`, that is represented by a lazy list `'a seq` in *AbstractSML* and SML (in both languages, `'a seq` can be defined on top of the already introduced language constructs; see [16]).

When implementing a set by a sequence, we require that the sequences must be duplicate free in order to establish in a simple way evaluation fairness, i.e. any element of a set will be constructed eventually by the evaluation, provided there are "enough tail selections" into the lazy list. These semantic side conditions has to be encapsulated similarly to the side-conditions for the recursor in program-statements. When these side-conditions are fulfilled, it is easy not only to provide a `MAP` and `FILTER` on sequences, but also a `UNION` as an interleave of two sequences. The definitions of `MAP`, `FILTER` and `UNION` can be expressed in terms of a program statement on top of the `LfpLang`; they do not add extra semantical complexity. Thus, the set of natural numbers N characterized by `lfp(%X. UNION{ZERO} (MAP SUC X))` is a program. Moreover, since ZF-expressions like $\{x : N \mid even\ x\}$ can be seen as an equivalent to a `FILTER` on N and is consequently also a program. This turns many definitions in the Z-library, the *Mathematical Toolkit* into programs, among them the definition for cartesian products and finite function spaces.

In order to reveal the power of this program notion, we show the example EightQueens stemming from [11]. In the EightQueens-problem, eight queens must be placed on a chess board with eight files and eight ranks such that no queen attacks any other, i.e. sits in the same file, rank or diagonal. We use Z-Notation here in order to keep the presentation compact:

Lib

$$up, down : SQUARE \rightarrow DIAG$$

$$\begin{array}{l} \forall f : FILE, r : RANK \\ \quad up(f, r) = r - f \\ \quad down(f, r) = r + f \end{array}$$

EightQueens

$$squares : FILE \twoheadrightarrow RANK$$

$$\{squares \lhd up, squares \lhd down\} \subseteq SQUARE \mapsto DIAG$$

where `FILE` and `RANK` are sets from 1 to 8 and `SQUARE` is the set of pairs of `FILE`- and `RANK`-positions a queen may sit in. The function definitions for `up` and `down` map to any queen position its up resp. down diagonal number. The set of EightQueen-solutions is described in the schema *EightQuenns*: Since SQUARE can be viewed as a relation, it is possible to require that each concrete solution squares must be a bijective function $\twoheadrightarrow$). Moreover, if the functions up and down where constrained to the positions of a solution squares, they must yield injective functions.

And here is the point: This example of a fairly declarative specification for a small tricky problem represents a program in LFP. All involved sets (including the set of bijective functions from FILE to RANK) are representable as combinations of MAP, FILTER, and UNION, such that the specification above (based on a small library of LfpLang-programs defining $\twoheadrightarrow$ and $\lhd$ etc.) can be translated into SML-code, that eventually enumerates the set Queens (patience required!).

6 The Generic Code-Generator

In this chapter we will shortly describe the SML-based implementation of the generic code-generator. The general idea is to implement the coder as an SML functor. This functor will be instantiated with three structures that implement our three programming languages WF, FIX and LFP described in chapter 5. The following SML signature shows the language dependent interface to the functor:

```
signature LANGUAGE =
sig
  val target     : theory; (* semantic interface to SML *)
  val lang       : theory; (* syntax and semantics of abstract
                              programming language *)
  val objectify        : thm -> thm
  val optimizeLang     : thm -> thm
  val ss_translate     : thm -> thm
  val translate        : thm -> thm
  val optimizeSML      : thm -> thm
  val convert          : thm -> Absy.absy
  ...
end;
```

Here, target of the Isabelle type theory represents the semantic interface to SML, i.e. the theory that describes the abstract programming language. The theory lang represents the source language of the coding process.

The six coding phases described in chapter 4 are implemented by the corresponding functions objectify, optimiseLang, ss_translate, translate, optimiseSML and convert. All these functions get a theorem of the Isabelle datatype thm that represents a program as argument. Except convert, all functions return again a theorem. The function convert returns the term of a program in the abstract syntax of SML, where term is the basic Isabelle data structure for terms and absy the type for the abstract syntax of SML of New Jersey.

The signature also provides functions that get information on datatype declarations based on a arbitrary datatype package. The signature of the functor looks as follows:

```
signature CODER =
sig
  exception NoCode of string
  val coder : string * (thm list) -> ()
end;
```

This signature provides the main coding function `coder`. It gets a string representing the name of the SML structure to be generated and a list of theorems that represent the programs to be compiled. There is an exception `NoCode` that will be raised if the process of code-generation fails. First, the function `coder` has to generate a graph representing the call dependencies of the various programs. Based on this graph, `coder` will sort the programs topologically. If any call cycles are detected the exception NoCode is raised. Then, `coder` retrieves the datatype informations and generates the corresponding SML datatypes and recursors. Now the six coding functions can be called. If the function `objectify` rejects a program not to be compilable, again the exception `NoCode` is raised. Finally, `coder` generates the abstract syntax tree of the final program and writes the pretty printed string to a file.

7 Conclusion

First, we have presented a method to formally investigate the correctness of code-generation schemes. Second, we have demonstrated its feasibility by instantiating it for three program notions, ranging from executablility suited for HOL, functional programming and Z. Third, we provide a technical framework for *implementing* trustworthy *code-generators* (i.e. in its crucial parts formally proven correct) based on the set of abstract programs, i.e. the executable sublanguage of HOL, and *AbstractSML*, i.e. a semantic interface to the target language. Our technique is based on a so called *shallow embeddings*, i.e. no explicit syntax is used to represent *AbstractSML*; rather, the semantics is represented via semantic operators directly embedded in HOL.

We argued that the formal proof of correctness of a code-generation in an absolute sense is impossible – at the very end, extra-logical arguments have to be used anyway.

By shifting the formalisation of *canonical forms* partially to the meta-level (including appropriate checks on the SML-level in the implementation), it is possible to stick to shallow embeddings and to make the proofs of correctness *substantially easier*. But there is a price to pay: In our approach, the proof of "canonicity" or normal-formedness is left to extra-logical reasoning. Still, we believe our approach represents a good compromise in the attempt to minimise the set of extra-logical assumptions and to base the phases of a code-generation on derived rules controlled by tactics.

From our experience with the nitty-gritty details of our code-generation schemes, 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. We hope that our technique can contribute to turn the formal investigation of code schemes used in compilers into a routine task.

7.1 Related Work

In the literature, there is a large body of papers in compiler verification. Typically, two explicit abstract syntaxes for the input and output language of the compiler were defined as data types, then a compiler function connects them. The proof of correctness is then based on two semantical interpretation functions and their commutability via the compiling function. In contrast to work along this style of representation – so called deep embeddings – our work is based on *shallow embeddings* for reasons discussed above. Moreover, our work is intended to be a component of a formal development environment. On the basis of shallow abstract-language encoding, far more activities can be founded than just compilation – interactive verification and transformation, for example.

Beyond this classical compiler verification projects, there are attempts to integrate programming- and specification languages. The work of Slind [19] presents a pure syntactical approach to the representation of programs on the level of the input of a theorem prover (Isabelle and HOL). The user can define functions in a very rich and compact functional notation employing powerful pattern-matching, that is parsed away into an 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.

In [6], 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 (*animate* it) one by one. In this view, set comprehensions are constructed by powerdomains, such that the operational view of the animation is deliberately different to the Z standard semantics.

7.2 Future Work

We will attempt to 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 datatypes should be replaced by a more general mechanism, possibly better integrated in a future version of Isabelle.

It is worth investigating the increase of genericity with respect to the target language: It should not be too difficult to develop other converters (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.

References

1. The common algebraic specification language. http://www.brics.dk/Projects/CoFI/.
2. The coq project. http://pauillac.inria.fr/coq/biblio-eng.html.
3. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
4. P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Stockholm Studies in Philosophy. Academic Press, Stockholm, 1986.
5. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with kiv. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering.*, number 1783 in LNCS. Springer, 2000. http://www.informatik.uni-ulm.de/pm/kiv/tools/index.html.
6. P.T. Breuer and J.P. Bowen. Towards correct executable semantics for Z. Availiable online.
7. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56– 68, 1940.
8. W. Grieskamp. *A Set-Based Calculus and its Implementation.* PhD thesis, Technische Universität Berlin, 1999. http://uebb.cs.tu-berlin.de/w̃g/diss.ps.gz.
9. J. Hatcliff and O. Danvy. Thunks and the lambda calculus. *Journal of Functional Programming*, 7:303– 319, 1997.
10. The Isabelle documentation page. www.informatik.tu-muenchen.de/ nipkow/isabelle.
11. J. Kacky. *The Way of Z - Practical Programming with Formal Methods.* Cambridge University Press, 1997.
12. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle. In J. von. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1125 in LNCS, pages 283 – 298. Springer Verlag, 1996.
13. C. Lüth and B. Wolff. Generic window inference with tas. In *Proc. TPHOLs '00*, lncs. Springer — to appear, 2000.
14. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML (revised).* MIT Press, 1997.
15. M.J.C.Gordon and T.M.Melham. *Introduction to HOL: a Theorem Proving Environment for Higher order Logics.* Cambridge University Press, 1993.
16. L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.
17. F. Regensburger. *HOLCF: Eine konservative Einbettung von LCF in HOL.* PhD thesis, Technische Universität München, 1994.
18. T. Santen. On the semantic relation of z and hol. In J. Bowen and A. Fett, editors, *Proc. ZUM '98*, number 1493 in lncs, pages 96—115. Springer, 1998.
19. K. Slind. Function definition in higer order logic. In J. von. Wright, J. Grundy, and J. Harrison, editors, *TPHOLs 96*, number 1125 in LNCS. Springer Verlag, 1996.
20. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. 2nd edition.
21. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer, Berlin, 1997.
22. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.