Tactic-based Optimized Compilation of Functional Programs

Thomas Meyer and Burkhart Wolff

Universität Bremen, Germany
ETH Zürich, Switzerland

Abstract Within a framework of correct code-generation from HOL-specifications, we present a particular instance concerned with the optimized compilation of a lazy language (called MiniHaskell) to a strict language (called MiniML).
Both languages are defined as shallow embeddings into denotational semantics based on Scott's cpo's, leading to a derivation of the corresponding operational semantics in order to cross-check the basic definitions.
On this basis, translation rules from one language to the other were formally derived in Isabelle/HOL. Particular emphasis is put on the optimized compilation of function applications leading to the side-calculi inferring e.g. strictness of functions.
The derived rules were grouped and set-up as an instance of our generic, tactic-based translator for specifications to code.

1 Introduction

The verification of compilers, or at least the verification of compiled code, is known to be notoriously difficult. This problem is still an active research area [3, 4, 12]. In recent tools for formal methods, the problem also re-appears in the form of code-generators for specifications — a subtle error at the very end of a formal development of a software system may be particularly frustrating and damaging for the research field as a whole.

In previous work, we developed a framework for *tactic-based* compilation [5]. The idea is to use a theorem prover itself as a tool to perform source-to-source transformations, controlled by tactic programs, on programming languages embedded into a HOL prover. Since the source-to-source transformations can be derived from the semantics of the program languages embedded into the theorem prover, our approach can guarantee the correctness of the compiled code, provided that the process terminates successfully and yields a representation that consists only of constructs of the target language. Constructed code can be efficient, since our approach can be adopted to optimized compilation techniques, too.

In this paper, we discuss a particular instance of this framework. We present the semantics of two functional languages, a Haskell-like language and an ML-like language for which a *simple* one-to-one translator to SML code is provided. We apply the shallow embedding technique for these languages [1] into standard

denotational semantics — this part of our work can be seen as a continuation of the line of "Winskel is almost right"-papers [8], which formalize proofs of a denotational semantics textbook [11, chapter 9].

As a standard translation, a lazy language can be transformed semantically equivalent via continuation passing style [2] into an eager language. While this compilation is known to produce fairly inefficient code, we also use derived rules for special cases requiring strictness- or definedness analysis. While we admit that the basic techniques are fairly standard in functional compilers, we are not aware of any systematic verification of the underlying reasoning in a theorem prover. Thus, we see here our main contribution.

The plan of the paper is as follows: After a brief outline of the general framework for tactic based compilation and a brief introduction into the used theories for denotational semantics, we discuss the embeddings of MiniHaskell and MiniML into them. These definitions lead to derivations of "classical" textbook operational semantics. In the sequel, we derive transformation rules between these two languages along the lines described by our framework. Then we describe the side-calculus to infer strictness required for optimized compilation; an analogous calculus for definedness is ommitted here.

2 Background

2.1 Concepts and Use of Isabelle/HOL

Isabelle [9] is a generic theorem prover of the LCF prover family; as such, we use the possibility to build programs performing symbolic computations over formulae in a logically safe (conservative) way on top of the logical core engine: this is what our code-generator technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church's higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on higher-order rewriting which we predominantly use to implement the translation phases of our code-generator.

Isabelle's type system provides parametric polymorphism enriched by type classes: It is possible to constrain a type variable $\alpha :: \texttt{order}$ to specify that an operator $_ <= _$ must be declared on any α; this syntactic concept known from languages such as Haskell is extended in Isabelle by semantic constraints: the operator must additionally fulfill the properties of a partial order.

The proof engine of Isabelle is geared towards rules of the form $A_1 \Rightarrow (\ldots \Rightarrow (A_n \Rightarrow A_{n+1}) \ldots)$ which can be interpreted as "from assumptions A_1 to A_n, infer conclusion A_{n+1}". This corresponds to the textbook notation

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

which we use throughout this paper.

Inside these rules, the meta-quantifier $\bigwedge$ is used to capture the usual side-constraint "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables.

2.2 The Framework for Code-Generation

Our generic framework [5] is designed to cope with various executability notions and to provide technical support for them. The following diagram in figure 1 represents the particular instance of the general framework discussed in this paper.

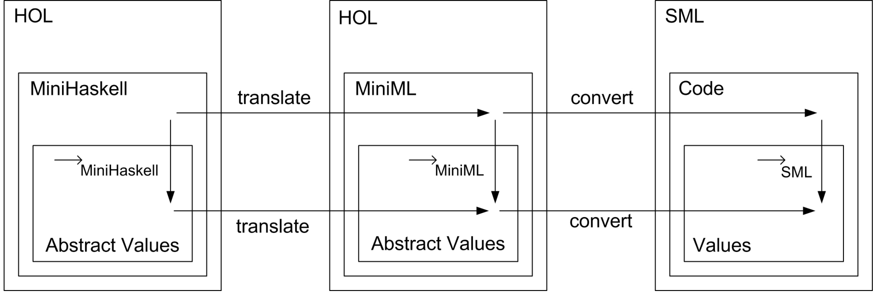

Figure 1. Basic Concepts

Here, the left block represents the language MiniHaskell, the center block the language MiniML, which are both presented as conservative shallow embedding into a theory of Scott Domains described in Section 2.3. A subset of both languages are the set of *abstract values*. The embeddings are mirrored by the corresponding terms of a (concrete) programming language, i.e. SML, and its subset of (concrete) *values* like e.g. the integers 1,2,3,.... The first two worlds are connected by the translate function, that consists of several tactics that control the translation process by source-to-source translation rules. The latter two worlds are connected by the code-generation function convert provided by our framework that is required to be total on the domain of abstract programs.

The three relations $\rightarrow_{\text{MiniHaskell}}$, $\rightarrow_{\text{MiniML}}$ and $\rightarrow_{\text{SML}}$ represent the operational semantics of the three languages. We require that they represent partial functions from programs to values. These operational semantics serve as cross-check of our denotational definitions of the language; in particular, $\rightarrow_{SML}$ can be compared against an (abstracted) version of the *real* SML semantics [6] in order to validate convert. Making these two diagrams commute (while the first commutation is based on formal proofs presented in this paper) constitutes the correctness of our overall translation process.

2.3 Denotational Semantics in HOL

The cornerstone of any denotational semantics is its fixpoint theory that gives semantics to systems of (mutual) recursive equations. The well-known Scott-Stratchey-approach is based on complete partial orders (*cpo*'s); variants thereof have also been used in standard semantics textbooks such as [11] to give semantics to the languages we discuss here (cf. chapter 9).

Several versions of denotational semantics theories are available for Isabelle [7, 10]. In both, the type class mechanism is used in order to model cpo's, which provide a least element $\perp$ and completeness on any type belonging to class `cpo`. This is essentially captured in the theory [10] underlying this work in the axiomatic class definition

```
axclass
  cpo < cpo0
  least        ⊥ ≤ x
  complete     directed X ⇒ (∃ b. X <<| b)
```

i.e. completeness means that for any directed set (any non-empty set where two elements have a supremum) there exists a least upper bound.

Moreover, in this type class a number of key concepts such as definedness and strictness of a function and making a function strict are defined:

```
DEF        :: α::cpo0 ⇒ bool           DEF x ≡ x≠⊥
is_strict  :: (α::cpo0 ⇒ β::cpo0) ⇒ bool
              is_strict f ≡ (f ⊥ = ⊥)
strictify  :: (α ⇒ β::cpo) ⇒ α ⊥ ⇒ β
              strictify f x ≡ if DEF(x) then f(x) else ⊥
```

Further, a type constructor can be defined that assigns to each type τ a *lifted type* $\tau_\perp$ by disjointly adding the $\perp$-element. All types lifted by this type constructor are automatically in the type class `cpo` but not necessarily vice versa. The function $\lfloor_\rfloor : \alpha \to \alpha_\perp$ denotes the injection, the function $\lceil_\rceil : \alpha_\perp \to \alpha$ its inverse, extended by $\lceil\perp\rceil = \perp$.

On cpo's, the usual fixpoint combinator `fix` is defined that is shown to posses the crucial fixpoint property

$$\frac{\texttt{cont}\ f}{\texttt{fix}\ f = f(\texttt{fix}\ f)}$$

for all functions f that are continuous. Further, there is the usual induction principle for all fixpoints of all types belonging to class `cpo`:

$$\frac{\texttt{cont}\ f \quad \texttt{adm}\ P \quad \bigwedge x.\ \begin{array}{c}P(x)\\ \vdots\\ P(f\ x)\end{array}}{P(\texttt{fix}\ f)}$$

where the second-order predicate `adm` for *admissibility* captures that a predicate P holds for a fixpoint if it holds for any approximation of it. `adm` distributes over universal quantification, conjunction and disjunction, but not necessarily over negation. Being defined is an admissible predicate, being total not. As a consequence of induction, we derived a kind of bi-simulation principle:

$$\frac{\texttt{cont}\ f \quad \texttt{cont}\ f' \quad \bigwedge x.\ \begin{array}{c}[P\ x]\\ \vdots\\ f\ x = f'\ x\end{array} \quad \bigwedge x.\ \begin{array}{c}[P\ x]\\ \vdots\\ P(f\ x)\end{array} \quad \texttt{adm}\ P}{\texttt{fix}\ f = \texttt{fix}\ f'}$$

which is the key for the proof of several crucial inference principles over recursive programs to be described in the subsequent sections. If some property P is invariant through execution of the body f, then P can be assumed for the "inner call" when proving the bodies f and f' equivalent over them.

3 The Semantics of MiniHaskell and MiniML

3.1 The Denotational Semantics of MiniHaskell

Based on the theories of denotational semantics, we define our first contribution — the formal definition of the lazy language MiniHaskell. The types of basic operations like `Bool` were lifted from HOL types

```
types
  Bool = bool⊥    Nat = nat⊥    Unit = unit⊥
```

and basic constants such as `TRUE` or `ONE` are defined accordingly by

```
TRUE :: Bool    TRUE ≡ ⌊True⌋
ONE  :: Nat     ONE  ≡ ⌊1⌋
```

The core of the MiniHaskell semantics consists of the definitions for the abstraction, application, conditional and the `LET`-construct. As well-known in the literature, an important difference between the denotational theory and the object language has to be made: the abstraction in MiniHaskell is a *value* — a so-called *closure* — and not a function space. Thus, a naive identification of the object language `LAM` with the meta language λ results in a completely wrong model of the operational behaviour: the expression `LAM` x. `ONE DIV ZERO` should be a *value*, i.e. different from λ`x. 1 DIV 0`, which is just λ`x.`$\bot$ or just $\bot$ in the function space. Consequently, the *lifted* function space is used, defined by:

```
types (α,β) ⇛ = (α ⇒ β)⊥
```

which results in the following definitions for the abstraction

```
Lam :: (α::cpo ⇒ β::cpo) ⇒ (α ⇛ β)
       Lam F ≡ ⌊F⌋
```

and its inverse, the application

```
▷ₗ :: (α::cpo ⇛ β::cpo) ⇒ α ⇒ β
      F ▷ₗ x ≡ ⌈F⌉ x
```

where we may write `LAM` x. $P\ x$ for `Lam` P. The `LET` construct is just a syntactical shortcut and defined by the application. The remaining definitions of the conditional and the recursor are standard:

```
If :: [Bool, α::cpo, α] ⇒ α
      IF x THEN y ELSE z ≡ case x of
                              ⌊v⌋ ⇒ if v then y else z
                            | ⊥   ⇒ ⊥
REC :: (α::cpo ⇒ α) ⇒ α
       REC f ≡ fix f
```

The basic operations of MiniHaskell are just strictified versions of the elementary operations of HOL. The paradigmatic example for a 1-ary and a 2-ary function are defined as follows:

```
SUC :: Nat ⇒ Nat
       SUC ≡ strictify(λx. ⌊Suc x⌋)
^<^ :: [Nat, Nat] ⇒ Bool
       (op ^<^) ≡ strictify(λx. strictify(λy. ⌊x<y⌋))
```

An example for a partial function is DIV:

```
DIV :: [Nat, Nat] ⇒ Nat
       DIV ≡ strictify(λx.
                 strictify(λy. if y=0 then ⊥
                               else ⌊x div y⌋))
```

As top-level constructs, we introduce the following two program definition constructs:

```
VAL  :: [α,α] ⇒ bool
        VAL f E ≡ (f = E)
FUN  :: [α::cpo, α ⇒ α] ⇒ bool
        FUN f F ≡ (f = REC(F)) ∧ cont F
```

This means that a recursive program is representable by the recursor REC of the language MiniHaskell under the condition, that the representing functional F is continuous. The Isabelle syntax engine is set up to parse also mutual recursive function definitions as a combination of fix and pairing. For example, a mutual recursive program in the object language MiniHaskell looks as follows:

```
fun fac x = IF x^=^ZERO THEN ONE ELSE x*(fac ▷l (x-ONE))
and add_fac x y = x+fac ▷l y
and suc_fac a   = add_fac ▷l ONE ▷l a;
```

Note, that the operators (op +), (op -) and (op *) are the overloaded (strictified) variants from MiniHaskell.

3.2 Lazy Operational Semantics of MiniHaskell

In the following, we derive the operational semantics presented in [11] in order to validate our denotational definitions. The basic concept of this operational semantics is a notion of terms representing values, called *canonical forms*. The judgment $t \in C_\tau$ states that a term t is a canonical form of type τ. It is defined by the following structural induction on the type τ:

Ground type: $n \in C_{\texttt{int}} = \{\texttt{ZERO},\ \texttt{ONE},\ \texttt{TWO},\ \ldots\}$ and $b \in C_{\texttt{bool}} = \{\texttt{TRUE},\ \texttt{FALSE}\}$

Function type: Closed abstractions are canonical forms, i.e. $(\texttt{LAM}\ x.\ t) \in C_{\tau_1 \to \tau_2}$ if t is closed

Note, that we can not give an inductive definition for canonical forms since we use a shallow embedding (the types presented above are represented on the meta-level). Nevertheless, by defining the evaluation relation $\to_l$ as equivalent to the

logical equality (i.e. evaluation must be correct), we can now derive the rules for the evaluation relation and check that they have the appropriate form $t \rightarrow_l c$, where t is a typeable closed 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:

$$c \rightarrow_l c \qquad \frac{t_1 \rightarrow_l c_1 \quad t_2 \rightarrow_l c_2}{t_1 \ op \ t_2 \rightarrow_l c_1 \ op \ c_2}$$

$$\frac{t_1 \rightarrow_l \texttt{TRUE} \quad t_2 \rightarrow_l c_2}{(\texttt{IF}\ t_1\ \texttt{THEN}\ t_2\ \texttt{ELSE}\ t_3) \rightarrow_l c_2} \qquad \frac{t_1 \rightarrow_l \texttt{FALSE} \quad t_3 \rightarrow_l c_3}{(\texttt{IF}\ t_1\ \texttt{THEN}\ t_2\ \texttt{ELSE}\ t_3) \rightarrow_l c_3}$$

$$\frac{t_1 \rightarrow_l \texttt{LAM}\ x.\ t \quad t[x := t_2] \rightarrow_l c}{t_1 \rhd_l t_2 \rightarrow_l c} \qquad \frac{t_2[x := t_1] \rightarrow_l c}{(\texttt{LET}\ x = t_1\ \texttt{IN}\ t_2 \rightarrow_l c)}$$

$$\texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t) \rightarrow_l \texttt{LAM}\ x.\ t\,[y := \texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t)]$$

As can be expected, the rule for canonical forms expresses that canonical forms evaluate to themselves. A key rule is that for the evaluation of applications: the evaluation of an application proceeds by the substitution of the argument into the function body; the treatment of the $\texttt{LET}\ x = t_1\ \texttt{IN}\ t_2$ is analogously. The rule for recursive definitions unfolds the recursion $\texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t)$ once, leading immediately to an abstraction $\texttt{LAM}\ x.\ t\,[y := \texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t)]$, and so a canonical form.

3.3 The Denotational Semantics of MiniML

Our semantic interface to the "real" SML target language, the language MiniML, differs with two regards from MiniHaskell:

1. syntactically, all constant symbols in MiniML are followed by a prime, e.g. ZERO', ONE', in order to distinguish them from their counterparts in MiniHaskell. This is for the sake of presentation only.
2. semantically, the two constructs application and LET differ from their counterparts in MiniML.

In the sequel, we turn to the semantic issues. In most cases, the semantics of the strict and the lazy constructs are the same. This holds for basic operators like NOT' or SUC' as well as the abstraction, the conditional and the REC' construct. This justifies logical equations such as NOT' $\equiv$ NOT etc.

The crucial difference between the two languages is the strict application. As usual, its denotational definition in MiniML is given by:

```
▷s :: (α::cpo ⇛ β::cpo) ⇒ α ⇒ β
      F ▷s x ≡ if x = ⊥ then ⊥
               else if F = ⊥ then ⊥ else ⌈F⌉ x
```

The LET' construct is defined as usual in terms of abstraction and strict application (enforcing the evaluation of the let-expression prior to the evaluation of its body).

3.4 Eager Operational Semantics of MiniML

The rules for the strict evaluation relation $\rightarrow_s$ is derived analogously to the lazy one $\rightarrow_l$. Therefore, we can focus on the differences to MiniHaskell, which are just the rules for the different constructs for the strict application and `LET'`. In contrast to MiniHaskell, the arguments are first evaluated before performing a substitution:

$$\frac{t_1 \rightarrow_s \texttt{LAM'}\ x.\ t \quad t_2 \rightarrow_s c_2 \quad t[x := c_2] \rightarrow_s c}{t_1 \rhd_s t_2 \rightarrow_s c}$$

$$\frac{t_1 \rightarrow_s c_1 \quad t_2[x := c_1] \rightarrow_s c}{(\texttt{LET'}\ x = t_1\ \texttt{IN'}\ t_2) \rightarrow_s c}$$

This concludes our definition and validation of the two languages MiniHaskell and MiniML in terms of a (pre-existing) theory of denotational semantics. In the following, we turn to the semantic translation between these languages by means of derived rules.

4 The Semantic Translation

Between the considered languages, the translation of most language constructs is a trivial rewriting due to semantic equivalence. The challenge, however, is the translation of the lazy application to the strict one, and, on the larger scale, the translation of lazy *user-defined* definition constructs to one or more strict versions.

The default solution is well-known and simple: each expression is *delayed* i.e. converted into a closure, and all basic operations were enabled to apply its argument first to the unit-element () in order to *force* the argument closure and to produce an elementary value only when finally needed. Thus, any lazy application can be simulated by an strict one, provided that arguments of applications have been sufficiently delayed.

However, the default solution is fairly inefficient since it delays *any* computation. Therefore, optimizations are mandatory. The principle potentials for such optimizations are

1. the strictness of the function to be applied to an argument (i.e. the argument is used under all possible evaluations) or
2. the definedness of the argument (i.e. delaying is inherently unnecessary).

The concepts discussed above were made precise by a number of combinators which serve either as coding primitive (such as the combinator `delay` and `force`) or as combinators such as `forcify` that represents intermediate states of the translation. We will derive rules that allow to "push" `forcify` combinators throughout a program and thus perform the translation.

In the following, we present these concepts formally. First, we introduce the type constructor `del` for representing delayed, i.e. suspended values:

```
types
  α del = Unit ⇛ α
```

The `delay`-constructor and the corresponding suspension destructor `force` can both be defined completely in terms of our target language MiniML:

```
delay :: α::cpo ⇒ α del
         delay f ≡ (LAM' x. f)
force :: (α::cpo)del ⇒ α
         force f ≡ (f ▷s UNIT')
```

Both combinators may remain in final program representations and are treated as primitive by the translation function convert.

It turns out that from these definitions the characteristic theorem

$$\texttt{force (delay } e) = e$$

can be derived as could be expected.

Now we define the `forcify` combinator that converts a function into its counterpart that deals with delayed values:

```
forcify :: (α ⇛ β::cpo) ⇒ (α del ⇛ β)
           forcify f ≡ LAM' x. ⌈f⌉(force x)
```

While the `delay` and `force` combinator can be understood as a primitive that can be coded by the converter, `forcify` is a combinator that is uncodable. It is only used internally in the source-to-source translation and has to disappear at the end.

The overall translation process consists of one language translation calculus and three side-calculi — `forcify`-propagation, strictness-reasoning and definedness reasoning, which consist, as mentioned, of derived rules.

4.1 Language Translation Calculus

As mentioned, all but two language constructs have equal semantics can therefore be converted straight-forward by a trivial rewrite rule such as

$$\texttt{SUC} = \texttt{SUC'}$$

The key translation rule for the lazy application has the following form:

$$(f \rhd_l a) = (\texttt{forcify}\ f) \rhd_s (\texttt{delay}\ a)$$

This rule states that a lazy application can always be converted into a strict one; the price is the delay of the argument and the necessary forcification of the function of the application. This rule represents the default translation rule, which is — since resulting in inefficient code — avoided whenever possible. The following two rules represent the optimized alternatives of the default scheme: a lazy application is identical with a strict application if its function is strict or if the argument is known to be defined and the function is not the totally undefined one:

$$\frac{\texttt{is_strict}\ f}{(f \rhd_l a) = (f \rhd_s a)} \qquad \frac{\texttt{DEF}\ a \quad \texttt{DEF}\ f}{(f \rhd_l a) = (f \rhd_s a)}$$

For the `LET'`-construct, these three cases are analogously. The Isabelle proofs of these rules are not very hard but reveal a number of technicalities that are easily overlooked in paper-and-pencil proofs.

These optimized translation rules lead to side-calculi that attempt to infer the necessary information. One of them, the strictness calculus, will be discussed in the following subsections.

4.2 Forcification-Propagation Calculus

In the following, we turn to the key of the default translation to MiniML, the forcification-propagation. The base cases treat identities and constant abstractions as well as basic operators. For the latter, we can assume by construction that they are strict since we only used a particular pattern of their definition built upon `strictify` and HOL-functions.

$$\texttt{forcify}\ (\texttt{LAM}\ x.\ x) = \texttt{LAM}\ x.\ \texttt{force}\ x \qquad \texttt{forcify}\ (\texttt{LAM}\ x.\ c) = \texttt{LAM}\ x.\ c$$

$$\frac{f \equiv \texttt{strictify}\ g}{\texttt{forcify}\ (\texttt{LAM}\ x.\ f\ x) = \texttt{LAM}\ x.\ f\ (\texttt{force}\ x)}$$

$$\frac{\forall f.\ f \equiv \texttt{strictify}\ (\lambda x.\ \texttt{strictify}\ (g\ x))}{\texttt{forcify}\ (\texttt{LAM}\ x.\ f\ c\ x) = \texttt{LAM}\ x.\ f\ c\ (\texttt{force}\ x)}$$

The following rules describe the propagation over the core language constructs for application, abstraction and conditional:

$$\begin{aligned} &\texttt{forcify}\ (\texttt{LAM}\ x.\ ((f\ x) \rhd_l (g\ x))) = \\ &\qquad \texttt{LAM}\ x.\ (\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x) \rhd_l \\ &\qquad\qquad (\texttt{forcify}\ (\texttt{LAM}\ x.\ (g\ x)) \rhd_l x)\ \ ) \end{aligned}$$

$$\begin{aligned} &\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{LAM}\ y.\ (f\ x\ y))) = \\ &\qquad \texttt{LAM}\ x.\ \texttt{LAM}\ y.\ (\texttt{forcify}(\texttt{LAM}\ x.\ (f\ x\ y)) \rhd_l x) \end{aligned}$$

$$\begin{aligned} &\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{IF}\ c\ x\ \texttt{THEN}\ f\ x\ \texttt{ELSE}\ g\ x)) = \\ &\qquad \texttt{LAM}\ x.\ (\ \texttt{IF}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (c\ x)) \rhd_l x) \\ &\qquad\qquad \texttt{THEN}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x) \\ &\qquad\qquad \texttt{ELSE}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (g\ x)) \rhd_l x)\ \ \ ) \end{aligned}$$

Of particular interest is also the rule for the propagation of forcification over the `REC` operator, which allows for the generation of recursive program definitions. In particular, applications like `forcify`f are mapped to the reference f', where we assume that for f there has been the previous statement `fun` $f\ x = E$ which has been converted to the code-variant `fun` $f' = \texttt{forcify}\ (\texttt{LAM}\ x.\ E)$. It is automatically proven that this precompiled variant satisfies the property $(\texttt{forcify}\ f) \rhd_s x = (f' \rhd_s x)$ which justifies the mapping mentioned above. Thus, “forcified” calls to previously defined functions were mapped to calls of “forcified” definitions.

$$\begin{aligned} &\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{REC}\ (f\ x))) = \\ &\qquad \texttt{LAM}\ x.\ \texttt{REC}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x) \end{aligned}$$

For n-ary functions, analogous rules have to be derived. Moreover, since any function may be strict in the first argument, but not in the second, or vice versa, or non-strict in all arguments, there are $2^{(n+1)} - 1$ rules for potential forcified code variants for direct recursive functions.

4.3 Strictness Calculus

As already mentioned, optimized applications require the inference of strictness properties of function bodies. Again, the inference rules follow the cases of our programming language. The base cases treat the identity, the special case of the abstraction yielding $\perp$ and operations defined upon `strictify`.

$$\texttt{is_strict}\ (\lambda x.\ x) \quad \texttt{is_strict}\ (\lambda x.\ \perp)$$

$$\frac{f \equiv \texttt{strictify}\ g}{\texttt{is_strict}\ f} \quad \frac{f \equiv \texttt{strictify}\ (\lambda x.\ \texttt{strictify}\ (g\ x))}{\texttt{is_strict}\ (f\ c)}$$

Note, that the case for the lambda abstraction is omitted since

$$\texttt{is_strict}\ (\lambda x.\ \texttt{LAM}\ y.\ (E\ x\ y))$$

simply does not hold: recall that a closure is a canonical form, hence a value different from $\perp$.

Since we suggest a source-to-source translation scheme, the calculus over strictness must cope with terms in which both strict and lazy applications may occur. Therefore, rules for both cases are needed. The inference reduces the applications to semantic functions and substitutes their denotation into it; in the case of the strict application, the argument must be strict in itself:

$$\frac{\texttt{is_strict}\ (\lambda x.\ \lceil f\ x \rceil\ (a\ x))}{\texttt{is_strict}\ (\lambda x.\ ((f\ x) \rhd_l (a\ x)))}$$

$$\frac{\texttt{is_strict}\ (\lambda x.\ \lceil f\ x \rceil\ (a\ x)) \quad \texttt{is_strict}\ (\lambda x.\ (a\ x))}{\texttt{is_strict}\ (\lambda x.\ ((f\ x) \rhd_s (a\ x)))}$$

Note that the computation of the semantic functions $\lceil f\ x \rceil$ requires an own (trivial) side-calculus allowing to "push" $\lceil _ \rceil$ inside; this side-calculus is not presented here.

With respect to the conditional, one gets two cases to establish strictness of the overall construct: either the condition is strict in x or both branches:

$$\frac{\texttt{is_strict}\ f}{\texttt{is_strict}\ (\lambda x.\ (\texttt{IF}\ (f\ x)\ \texttt{THEN}\ (g\ x)\ \texttt{ELSE}\ (h\ x)))}$$

$$\frac{\texttt{is_strict}\ g \quad \texttt{is_strict}\ h}{\texttt{is_strict}\ (\lambda x.\ (\texttt{IF}\ (f\ x)\ \texttt{THEN}\ (g\ x)\ \texttt{ELSE}\ (h\ x)))}$$

The most technical proofs of this paper are behind the rules for inferring strictness of recursive schemes and definition constructs. These schemes — which

perform an implicit induction — are consequences of the bi-simulation briefly presented in Section 2.3:

$$\frac{\texttt{cont}\ F \quad \bigwedge H.\ \begin{matrix}[\texttt{is_strict}\ H] \\ \vdots \\ \texttt{is_strict}\ (F\ H)\end{matrix}}{\texttt{is_strict}\ (\texttt{REC}\ F)}$$

This rule performs (for the 1-ary recursive function) a kind of specialized fixpoint induction proof: If we can establish strictness of the body F provided that a function H replaced in the recursive call is strict, then the recursor `REC` F yields a function that is strict in its first argument. Note, that for the n-ary cases similar rules are needed that are omitted here.

5 Examples

The calculi are grouped into several sets of rules which were inserted in the Isabelle rewriter. As a result, several tactics are available that perform the translation phases fully automatically.

5.1 Example 1

As a first example, we define a function in MiniHaskell whose body consists of a 2-ary lambda abstraction which is strict in its second argument. Its first argument represents an undefined value ⊥:

```
fun f y = (LAM a b. b) ▷l (DIV x ZERO) ▷l y;
```

The first translation phase is able to derive the strictness in the second argument and replaces the second lazy application by a strict one:

```
fun f y = (LAM a b. b) ▷l (DIV x ZERO) ▷s y;
```

The next translation phase replaces the remaining lazy application by our default translation. Recall that a lazy application can always be converted into a strict one by delaying the argument and forcifying the function of the application. Furthermore, a forcification-propagation is performed:

```
fun f y =
  LAM a b. (LAM a. b ▷s delay a) ▷s
               delay (DIV x ZERO) ▷s y;
```

A one-to-one translation is performed by the following translation phase. Each MiniHaskell construct is replaced by its MiniML counterpart yielding a pure MiniML-program:

```
fun' f y =
  LAM' a b.(LAM' a.b ▷s delay a) ▷s
               delay (DIV' x ZERO') ▷s y;
```

The final translation phase performs an optimization by reducing the MiniML-program to the identity:

```
fun' f y = y;
```

5.2 Example 2

As a second example, we define the factorial function in MiniHaskell representing a recursive function:

```
fun fac x =
  IF (x ^=^ ZERO) THEN ONE ELSE x * (fac ▷l (x - ONE));
```

Here, the first translation phase deduces that the function `fac` is strict in its argument and replaces the lazy application in the recursive call by the strict one:

```
fun fac x =
  IF (x ^=^ ZERO) THEN ONE
                  ELSE x * (fac ▷s (x - ONE));
```

Finally, the next phase replaces each MiniHaskell-construct by its corresponding MiniML-counterpart:

```
fun' fac x =
  IF' (EQ' x ZERO')
     THEN' ONE'
     ELSE' TIMES' x (fac ▷s (MINUS' x ONE'));
```

6 Conclusion

We address a well-known compilation problem of functional programming. We embed the semantics of both languages into a theory of denotational semantics and derive — as a check of these definitions — the corresponding operational semantics of these languages. The resulting strict semantics can be compared with the semantics of SML [6] and recognized as its abstracted version.

Finally, we derived a couple of rewrite rules that describe the translation of both languages as a source-to-source translation, which is prototypically implemented as a tactic-based compiler finally yielding executable code in SML.

Since the proofs of the translation rules are surprisingly simple (with few exceptions that are interesting in themselves), our approach yields a testbed for the implementation of compilers also for richer languages. Furthermore, it is feasible to develop typical libraries such as lists and compile them with our tactic-based compiler once and for all. Further, our approach may also be relevant to boot-strapping schemes when developing a proven correct compiler.

6.1 Further Work

We see the following issues for an extention of our work:

1. *Extending* MiniHaskell: a richer language comprising Cartesian products or lazy data types would help, in particular for the generation of concrete code.
2. *Low level target language*: In principle, our approach can also be applied for the generation of machine-code or JAVA byte-code.

References

[1] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[2] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.

[3] S. Glesner. Using program checking to ensure the correctness of compiler implementations. *JUCS*, 9(3), 2003.

[4] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. Technical report, TUM, March 2003.

[5] T. Meyer and B. Wolff. Correct code-generation in a generic framework. In M. Aargaard, J. Harrison, and T. Schubert, editors, *TPHOLs 2000: Supplemental Proceedings*, OGI Technical Report CSE 00-009, pages 213–230. Oregon Graduate Institute, Portland, USA, July 2000.

[6] R. Milner, M. Tofte, and R. Harper, editors. *The Definition of Standard ML (revised)*. MIT Press, 1997.

[7] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[8] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.

[11] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.

[12] L. Zuck, A. Pnueli, Y. Fang, and B. G. B. A methodology for the translation validation of optimizingcompilers. *JUCS*, 9(3), 2003.