JFP **22** (1): 9–30, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000019 First published online 29 February 2012

*A compiled implementation of normalisation by evaluation**

KLAUS AEHLIG
Institut für Informatik, Ludwigs-Maximilians-Universität München

FLORIAN HAFTMANN† and TOBIAS NIPKOW
Fakultät für Informatik, Technische Universität München
(*e-mail:* nipkow@in.tum.de)

Abstract

We present a novel compiled approach to Normalisation by Evaluation (NBE) for ML-like languages. It supports efficient normalisation of open λ-terms with respect to β-reduction and rewrite rules. We have implemented NBE and show both a detailed formal model of our implementation and its verification in Isabelle. Finally we discuss how NBE is turned into a proof rule in Isabelle.

1 Introduction

Symbolic normalisation of terms with respect to user provided rewrite rules is one of the central tasks of any theorem prover. Several theorem provers (see Section 6) provide especially efficient normalisers which have been used to great effect (Nipkow *et al.*, 2006; Gonthier, 2008) in carrying out massive computations during proofs. Existing implementations perform normalisation of open terms either by compilation to an abstract machine or by Normalisation by Evaluation, NBE for short. The idea of NBE is to carry out the computations by translating into some underlying functional implementation language, evaluating there and translating back. The key contributions of this paper are:

1. A novel compiled approach to NBE that exploits the pattern matching already available in most functional languages, while allowing the normalisation of open λ-terms with respect to β-reduction and a set of (possibly higher-order) rewrite rules.
2. A formal model and correctness proof[1] of our approach in Isabelle/HOL (Nipkow *et al.*, 2002), as well as a formal proof that any output obtained, is indeed a normal term.

* This is a revised and extended version of (Aehlig *et al.*, 2008).

† Supported by DFG Grant Ni 491/10

[1] Available online at afp.sf.net

NBE has been available at the user-level since Isabelle 2007, both to obtain the normal form t' of some open term t, which is its main application, and as a proof rule that yields the theorem $t = t'$.

We give some examples of simple symbolic computations that exhibit key properties of the approach. For a start, we can evaluate symbolic terms. For example, `rev [a,b,c]` (where `rev` reverses a list) yields `[c,b,a]`. Terms need not be of base type. For example, the append function on lists, which is defined by recursion on its first argument, can be partially evaluated: `append []` yields `λx. x`. We can also declare additional equations, which have been proved as lemmas, as evaluation rules. For example, making `(xs @ ys) @ zs = xs @ (ys @ zs)` (where `@` is the infix syntax for append) an evaluation rule lets `(xs @ [a,b]) @ [c,d]` normalise to `xs @ [a,b,c,d]`. As a similar example, declaring `- (- x) = x` as an evaluation rule, the term `(λx. - x) ^^ 5` (where `^^` is function iteration) reduces to the unary `-`.

Isabelle also has a compilation-based evaluator for ground terms (Haftmann & Nipkow, 2010). That evaluator suffices for the efficiency-critical applications cited above (Nipkow *et al.*, 2006; Gonthier, 2008) which employ *reflection*: the syntactification of formulas of the logic, yielding ground terms that are evaluated/decided with the help of a decision procedure programmed and verified in the logic. Hence we are prepared to accept reduced performance for open terms (Section 4).

Since our concrete implementation is part of the Isabelle theorem prover whose system implementation language is ML, we choose ML as implementation language, too; this allows us to carry out evaluation in the same system environment as the theorem prover itself using runtime compiler invocation. Hence the stack of needed system software is not increased. This architecture is a design choice, no prerequisite: any language in the ML family, including Haskell, is suitable for evaluation, and evaluation could also be carried out in an external process or on a remote machine. For brevity, throughout the paper we refer to the underlying implementation language as ML, though.

The guiding principle of our realisation of NBE is to off-load as much work as possible onto ML – not just substitution but also pattern matching. Thus the word "compiled" in the title refers to both the translation from the theorem prover's λ-calculus into ML and from ML to some byte or machine code.

2 Normalisation by evaluation in ML

Normalisation by Evaluation uses the evaluation mechanism of an underlying implementation language to normalise terms, typically of the λ-calculus. By means of an evaluation function $[\cdot]_\xi$, or, alternatively, by compiling and running the compiled code, terms are embedded into this implementation language. In other words, we now have a native function in the implementation language. Then, a function $\downarrow$, which acts as an "inverse of the evaluation functional" (Berger & Schwichtenberg, 1991), serves to recover terms from the semantics. This process is also known as "type-directed partial evaluation" (Danvy, 1996).

Normalisation by Evaluation is best understood by assuming a semantics consisting of a reduction relation $\rightarrow$ on terms, a denotational semantics $[.]$ mapping terms to some domain, and a function $\downarrow$ in the other direction, all enjoying the following two properties.

- *Soundness:* if $r \rightarrow s$ then $[r]_\xi = [s]_\xi$, for any valuation ξ.
- *Reproduction:* there is a special valuation $\uparrow$ such that for any term r in normal form with respect to $\rightarrow$ we have $\downarrow [r]_\uparrow = r$.

These properties ensure that $\downarrow [r]_\uparrow$ actually yields a normal form of r if it exists. Indeed, let $r \rightarrow^* s$ with s normal; then $\downarrow [r]_\uparrow =\downarrow [s]_\uparrow = s$.

2.1 λ-calculus in ML

We implement untyped normalisation by evaluation (Aehlig & Joachimski, 2004) in ML. To do so, we need to construct a model of the untyped λ-calculus, i.e. a data type containing its own function space. Moreover, in order to make the reproduction property possible, our model ought to include some syntactical elements in it like constructors for free variables of our term language. Fortunately, ML allows data types containing their own function space. So we can simply define a universal type `Univ` like the following:

```
datatype Univ =
    Const of string * Univ list
  | Var of vname * Univ list
  | Clo of (Univ list -> Univ) * Univ list * int
```

Here `vname` is the type of variable names uniquely identifying free variables. We do not make any assumptions about this type, besides that we efficiently can check for equality and obtain new elements. In our implementation, we use `int`, but this is mere convenience, not a necessity of the method.

Note how the constructors of the data type allow to distinguish between an element of `Univ` that is a function and one that is not. In type-directed partial evaluation such a tagging is not needed, as the type of the argument already tells what to expect. On the other hand, this need of anticipating the type of arguments restricts the implementation to a typing discipline that can easily be embedded into the simple types. Our untyped approach is flexible enough to work with any form of rewrite system, including polymorphically typed ones.

The data type `Univ` represents normal lambda terms. So, we have a constructor for variables and a constructor for functions, but we do not have a constructor for application. Instead, we have a function `apply: Univ -> Univ -> Univ` discussed below.

`Const` serves to embed constructors of data types of the underlying theory; they are identified by the string argument. Normal forms can have the shape $C\, t_1 \ldots t_k$ of a constructor C applied to several (normal) arguments. Therefore, we allow `Const` to come with a list of arguments, for convenience of the implementation in reverse order. In a similar manner, the constructor `Var` is used to represent expressions of the form $x\, t_1 \ldots t_k$ with x, a variable.

The constructor Clo represents partially applied functions. More precisely, "Clo (f, [a_k,. . . ,a_1], n)" represents the $(n+k)$-ary function f applied to $a_1, \ldots, a_k$. This expression needs another n argument before f can be evaluated. In the case of the pure λ-calculus, n would always be 1 and f would be a value obtained by using (Standard) ML's function abstraction "fn x => . . . ". Of course, ML's understanding of the function space is bigger than just the functions that can be obtained by evaluating a term in our language. For example, recursion can be used to construct representations for infinite terms. Nevertheless our modelling is faithful, as we only need that Univ contains enough elements, not that it be in one-to-one correspondence with our term calculus. During our normalisation process, only functions that can be named by a term will occur as arguments to Clo.

As mentioned, application is realised by an ML-function apply. With the discussed semantics in mind, it is easy to construct such a function: in the cases that $C\,t_1 \ldots t_k$ or $x\,t_1 \ldots t_k$ is applied to a value s, we just add it to the list. In the case of a partially applied function applied to some value s we either, in case more than one argument is still needed, collect this argument or, in case this was the last argument needed, we apply the function to its arguments.

```
fun apply (Clo (f, xs, 1)) x = f (x :: xs)
  | apply (Clo (f, xs, n)) x = Clo (f, x :: xs, n - 1)
  | apply (Const (name, args)) x = Const (name, x :: args)
  | apply (Var (name, args)) x = Var (name, x :: args)
```

It should be noted that the first case in the above definition is the one that triggers the actual work: compiled versions of the functions of the theory are called. As discussed above, our semantic universe Univ allows only normal values. Therefore, this call carries out all the normalisation work.

2.2 Translation of functions

As an example, consider the function append on lists as defined in Isabelle/HOL

```
fun append :: "'a list => 'a list => 'a list" where
  "append Nil          bs = bs" |
  "append (Cons a as) bs = Cons a (append as bs)"
```

and assume "append (append as bs) cs = append as (append bs cs)" has been proved, which is associativity of append. Compiling all these equations yields the following ML code:

```
fun append [v_cs, Const ("append", [v_bs, v_as])] =
        append [append [v_cs, v_bs], v_as]
  | append [v_bs, Const ("Cons", [v_as, v_a])] =
        Const ("Cons", [append [v_bs, v_as], v_a])
  | append [v_bs, Const ("Nil", [])] =
        v_bs
  | append [v_a, v_b] =
        Const ("append", [v_a, v_b])
```

The second and third clauses of the function definition are in one-to-one correspondence with the definition of the function append in the theory; the first clause represents associativity. The arguments, both on the left and right side, are in reverse order; this is in accordance with our semantics that $f a_1 \dots a_n$ is implemented as "f [a_n,... , a_1]".

The last clause is a *default clause* fulfilling the need that the ML pattern matching covers all arguments that will ever occur. But our equations, in general, do not cover all cases. The constructor Var for variables is an example for a possible argument usually not covered by any rewrite rule. In this situation where we have all arguments for a function but no rewrite rule is applicable, no redex was generated by the last application—and neither will be by applying this expression to further arguments, as we have already exhausted the arity of the function. Therefore, we can use the append function as a constructor. Using (the names of) our compiled functions as additional constructors in our universal data type is a necessity of normalising open terms. In the presence of variables not every term reduces to one built up from only canonical constructors; instead, we might obtain normal forms with functions like append. Using them as additional constructors is the obvious way to represent these normal forms in our universal semantics.

Keeping this reproduction case in mind, we can understand the first clause. If the first argument is of the form append, in which case it cannot further be simplified, we can use associativity. Note that we are actually calling the append function, instead of using a constructor; in this way we ensure to produce a normal result.

2.3 Translating back

As discussed, values of type Univ represent normal terms. Therefore, we can easily implement the ↓-function which will be called term in our implementation. The function term returns a normal term representing a given element of Univ. Its definition is as follows:

```
term (Const (cnm, [vn,...,v1])) = c (term(v1)) ... (term(vn))
term (Var   (vnm, [vn,...,v1])) = x (term(v1)) ... (term(vn))
term (Clo args) = λx. term(apply (Clo args) x)
```

In the Const/Var case, c/x is the constant/variable named by *cnm*/*vnm*. Keep in mind that arguments are in reverse order in the implementation. In the Clo case we carry out an eta expansion: the closure denotes a function that needs another argument, which we give it in the form of a fresh variable x. This application to the fresh x is performed via the function apply discussed above. In particular, this application might trigger a redex and therefore cause more computation to be carried out. For example, as normal form of "append Nil" we obtain—without adding any further equations!—the correct function "λu. u".

When reading the term equations, beware that on the right-hand side we find two kinds of applications, both represented by juxtaposition: that of the term language and that of the implementation language ML. Our formal model will clearly distinguish the two.

It should be noted that the definition of `term` follows the structure of normal forms. Expressions of the shape $x\, t_1 \dots t_k$ and $\lambda x.t$ are normal if $t, t_1, \dots, t_k$ are. This idea can be used to formally show that our algorithm only outputs normal terms (see Section 3.6).

Compared to the expressiveness of the underlying term language in Isabelle, our universal data type is quite simple. This is due to the fact that we consider an untyped term-rewriting mechanism. This simplicity, however, comes at a price: we have to translate back and forth between a typed and an untyped world. Forgetting the types to get to the untyped rewrite structure is, essentially, an easy task, even though some care has to be taken to ensure that the more advanced Isabelle features like type classes and overloading are compiled away correctly and the term to be normalised obeys the standard Hindley-Milner type discipline. More details of this transformation into standard typing discipline are described in Section 4.

From terms following this standard typing discipline the types are thrown away and the untyped normal form is computed, using the mechanism described earlier. Afterwards, the full type annotations are reconstructed. To this end, the types of all free variables have been stored before normalisation; the most general types of the constants can be uniquely rediscovered from their names. The type of the whole expression is kept as well, given that the Isabelle object language enjoys subject reduction. Standard type inference will obtain the most general type annotations for all sub-terms such that all these constraints are met. Since we are in a simply-typed setting without a let-construct, type inference is a linear-time problem.

In most cases, these type reconstructions are unique, as follows from the structure of normal terms in the simply-typed lambda calculus. However, in the presence of polymorphic constants, the most general type could be more general than intended. For example, let `f` be a polymorphic constant of type "`('a => 'a) => bool`", say without any rewrite rule. Then the untyped normal form of "`f (λu::bool. u)`" would be "`f (λu. u)`" with most general type annotations "`f (λu::'a. u)`". To avoid such widening of types only those equations will be considered as being proved by normalisation where the typing of the result is completely determined, i.e. those equations where the most general type for the result does not introduce any new type variables. It should be noted that this, in particular, is always the case if an expression evaluates to `True`.

3 Model and verification

This section models the previous section in Isabelle/HOL and proves partial correctness of the ML level with respect to rewriting on the term level. In other words, we will show that, if NBE returns an output t' to an input t, then $t = t'$ could have also be obtained by term rewriting with equations that are consequences of the theory. Moreover, t' will be in β-normal form and none of the term rewriting rules will be applicable.

We do not attempt to handle questions of termination or uniqueness of normal forms. This would hardly be possible anyway, as arbitrary proven equations may be added as rewrite rules. Given this modest goal of only showing soundness, which however is enough to ensure conservativity of our extension of the theorem prover, we over-approximate the operational semantics of ML. That is, every reduction ML can make is also a possible reduction our model of ML can make. Conversely, our ML model is non-deterministic with respect to both the choice among the applicable clauses of a compiled function and the order in which to evaluate functions and arguments—any evaluation strategy is fine, even non-left-linear equations are permitted in function definitions. This over-approximation shows that partial correctness of our implementation is quite independent of details of the implementation language. In particular, we could have chosen any functional language, including lazy ones like Haskell.

In Section 2, it was explained that Normalisation by Evaluation is best understood in terms of "soundness of the semantics" (i.e. the semantics identifies enough terms) and "reproduction" (i.e. normal terms can be read off from the semantics). For showing partial correctness, however, the task is slightly different. First of all, we cannot really guarantee that our semantics identifies enough terms; there might be equalities that hold in the Isabelle theory under consideration that are not expressed as rewrite rules. Fortunately, this is not a problem. A failure of this property can only lead to two terms that are equal in the theory, but still have different normal forms. The lack of these properties requires us to show a slightly stronger form of the reproduction property. We need to show for an *arbitrary* term r that $\downarrow [r]_{\uparrow}$ is, if defined, a term that our theory equates with r. To show this property, we give an operational semantics of our implementation language ML and assign each computation state during the reduction of terms a "denoted term". Then we just have to show that each step our ML model makes either does not change the denoted term, or transforms it to a term of which our theory shows that it is equal.

3.1 Basic notation

HOL conforms largely to everyday mathematical notation. This section introduces some non-standard notation and a few basic data types with their primitive operations.

The types of truth values and natural numbers are called *bool* and *nat*. The space of total functions is denoted by $\Rightarrow$. The notation $t :: \tau$ means that term t has type τ.

Sets over type α, type α *set*, follow the usual mathematical convention.

Lists over type α, type α *list*, come with the empty list [], the infix constructor $\cdot$, the infix @ that appends two lists, and the standard functions *map* and *rev*.

3.2 Terms

We model bound variables by de Bruijn indices (Bruijn, 1972) and assume familiarity with this device, and in particular the usual lifting and substitution operations. Below we will not spell those out in detail but merely describe them informally—the details are straightforward. Because variables are de Bruijn indices, i.e. natural numbers, the types *vname* and *ml-vname* used below are merely abbreviations for *nat*. Type *cname* on the other hand is an arbitrary type of constant names,for example strings.

ML terms are modelled as a recursive data type:

$$
\begin{aligned}
\mathit{ml} = {} & C_{ML}\ \mathit{cname} \\
\mid {} & V_{ML}\ \mathit{ml\text{-}vname} \\
\mid {} & A_{ML}\ \mathit{ml}\ (\mathit{ml\ list}) \\
\mid {} & \mathit{Lam}_{ML}\ \mathit{ml} \\
\mid {} & C_U\ \mathit{cname}\ (\mathit{ml\ list}) \\
\mid {} & V_U\ \mathit{vname}\ (\mathit{ml\ list}) \\
\mid {} & \mathit{Clo}\ \mathit{ml}\ (\mathit{ml\ list})\ \mathit{nat} \\
\mid {} & \mathit{apply}\ \mathit{ml}\ \mathit{ml}
\end{aligned}
$$

The default type of variables u and v shall be *ml*.

The constructors come in three groups:

- The λ-calculus underlying ML is represented by C_{ML}, V_{ML}, A_{ML} and Lam_{ML}. Note that application A_{ML} applies an ML terms to a list of ML terms to cover both ordinary application (via singleton lists) and to model the fact that our compiled functions take lists as arguments. Constructor Lam_{ML} binds V_{ML}.
- Terms of the data type `Univ` (Section 2) are encoded by the constructors C_U, V_U and *Clo*. Note that the first argument of *Clo* is not of function type because type *ml* represents ML terms, not values.
- Constructor *apply* represents the ML function `apply` (Section 2).

Note that this does not model all of ML but just the fraction we need to express computations on elements of type `Univ`, i.e. encoded terms. The fact that the constructors of `Univ` and *apply* are also constructors of *ml* rather than encoded via C_{ML} is an optimisation; among other things, it directly expresses their type.

Capture-avoiding substitution $\mathit{subst}_{ML}\ \sigma\ u$, where $\sigma :: \mathit{nat} \Rightarrow \mathit{ml}$, replaces $V_{ML}\ i$ by $\sigma\ i$ in u. Notation $u[v/i]$ is a special case of $\mathit{subst}_{ML}\ \sigma\ u$ where σ replaces V_{ML} i by v and decreases all ML variables $> i$ by 1. Lifting the free ML variables $\geqslant$ i is written $\mathit{lift}_{ML}\ i\ v$. Predicate closed_{ML} checks if an ML value has no free ML variables ($\geqslant$ a given de Bruijn index).

The term language of the logical level is an ordinary λ-calculus, again modeled as a recursive data type:

$$\mathit{tm} = C\ \mathit{cname} \mid V\ \mathit{vname} \mid \mathit{tm} \bullet \mathit{tm} \mid \Lambda\ \mathit{tm} \mid \mathit{term}\ \mathit{ml}$$

The default type of variables r, s and t shall be *tm*.

This is the standard formalisation of λ-terms (using de Bruijn), but augmented with *term*. It models the function `term` from Section 2. The subset of terms not containing *term* is called *pure*.

We abbreviate $(\ldots(t \bullet t_1) \bullet \ldots) \bullet t_n$ by $t \bullet\bullet [t_1,\ldots,t_n]$. We have the usual lifting and substitution functions for term variables. Capture-avoiding substitution $\mathit{subst}\ \sigma\ s$, where $\sigma :: \mathit{nat} \Rightarrow \mathit{tm}$, replaces $V\ i$ by $\sigma\ i$ in s and is only defined for pure terms. The special form $s[t/i]$ is defined in analogy with $u[v/i]$ above, only for term variables. Lifting the free term variables $\geqslant i$ is written $\mathit{lift}\ i$ and applies both to terms (where V is lifted) and ML values (where V_U is lifted).

In order to relate the encoding of terms in ML back to terms we define an auxiliary function $\mathit{kernel} :: \mathit{ml} \Rightarrow \mathit{tm}$ that maps closed ML terms to λ-terms. For succinctness *kernel* is written as a postfix !; *map kernel vs* is abbreviated to *vs*!. Note that postfix binds tighter than prefix, i.e. $f\ v\,!$ is $f\ (v\,!)$.

$$
\begin{aligned}
&(C_{ML}\ nm)! = C\ nm\\
&(A_{ML}\ v\ vs)! = v\,! \bullet\bullet (rev\ vs)!\\
&(Lam_{ML}\ v)! = \Lambda\ ((lift\ 0\ v)[V_U\ 0\ []/0])!\\
&(C_U\ nm\ vs)! = C\ nm \bullet\bullet (rev\ vs)!\\
&(V_U\ x\ vs)! = V\ x \bullet\bullet (rev\ vs)!\\
&(Clo\ f\ vs\ n)! = f\,! \bullet\bullet (rev\ vs)!\\
&(apply\ v\ w)! = v\,! \bullet w\,!
\end{aligned}
$$

The argument lists *vs* need to be reversed because, as explained in Section 2, the representation of terms on the ML level reverses argument lists to allow `apply` to add arguments to the front of the list.

The case of Lam_{ML} warrants a bit of explanation. We lift all *syntactical* variables and substitute $V_U\ 0\ []$ for the 0'th *ML*-variable. So, the lifting and the substitution actually do *not* interfere. Applying the kernel to the obtained expression will eventually transform $V_U\ 0$ into $V\ 0$. The latter is bound by the *Lam* while all other syntactical variables are shifted out of the way.

The kernel of a *tm*, also written $t\,!$, replaces all sub-terms *term v* of t by $v\,!$.

Note that ! is not structurally recursive in the Lam_{ML} case. Hence, it is not obvious to Isabelle that ! is total, in contrast to all of our other functions. To allow its definition (Krauss, 2006) we have shown that the (suitably defined) size of the argument decreases in each recursive call of !. In the Lam_{ML} case this is justified by proving that both lifting and substitution of $V_U\ i\ []$ for $V_{ML}\ i$ do not change the size of an ML term.

3.3 Reduction

We introduce two reduction relations: → on pure terms, the usual λ-calculus reductions, and ⇒ on ML terms, which models evaluation in functional languages.

3.3.1 Reduction of pure terms

The reduction relation → on pure terms is defined by β-reduction: $\Lambda\ t \bullet s \rightarrow t[s/0]$, η-expansion: $t \rightarrow \Lambda\ (lift\ 0\ t \bullet V\ 0)$, rewriting:

$$\frac{(nm,\ ts,\ t) \in R}{C\ nm \bullet\bullet map\ (subst\ \sigma)\ ts \rightarrow subst\ \sigma\ t}$$

and context rules:

$$\frac{t \rightarrow t'}{\Lambda\ t \rightarrow \Lambda\ t'} \qquad \frac{s \rightarrow s'}{s \bullet t \rightarrow s' \bullet t} \qquad \frac{t \rightarrow t'}{s \bullet t \rightarrow s \bullet t'}$$

Note that R :: (*cname* × *tm list* × *tm*) *set* is a global constant that models a (fixed) set of rewrite rules. A triple $(f,\ ts,\ t)$ models the rewrite rule $C\ f \bullet\bullet ts \rightarrow t$.

3.3.2 Reduction of ML terms

Just like $\rightarrow$ depends on R, $\Rightarrow$ depends on a compiled version of the rules, called $compR :: (cname \times ml\ list \times ml)\ set$. A triple (f, vs, v) represents the ML equation with left-hand side $A_{ML}\ (C_{ML}\, f)\ vs$ and right-hand side v. The definition of $compR$ in terms of our compiler is given further below.

First we have β-reduction $A_{ML}\ (Lam_{ML}\ u)\ [v] \Rightarrow u[v/0]$ and invocation of a compiled function. The latter is modelled by two inference rules. One describes the reductions induced by closed instances of compiled rewrite rules:

$$\frac{(nm, vs, v) \in compR \qquad \forall i.\ closed_{ML}\ 0\ (\sigma\ i)}{A_{ML}\ (C_{ML}\ nm)\ (map\ (subst_{ML}\ \sigma)\ vs) \Rightarrow subst_{ML}\ \sigma\ v}$$

The other one describes the default clause in our implementation:

$$\frac{\begin{array}{c}\forall i.\ closed_{ML}\ 0\ (\sigma\ i) \qquad vs = map\ V_{ML}\ [0..{<}arity\ nm] \\ vs' = map\ (subst_{ML}\ \sigma)\ vs \qquad no\text{-}match\text{-}compR\ nm\ vs'\end{array}}{A_{ML}\ (C_{ML}\ nm)\ vs' \Rightarrow subst_{ML}\ \sigma\ (C_U\ nm\ vs)}$$

This rule requires some explanation. Function *arity* is a global table mapping each constant to the number of arguments it expects. Notation $[0..{<}k]$ is short for $[0,\ldots,k-1]$. The default rule only applies if none of the other rules for the compiled function match. This is formalised via an abbreviation

$$\begin{array}{l} no\text{-}match\text{-}compR\ cs\ vs \equiv \\ \forall (nm', ps, v){\in}compR.\ cs = nm' \longrightarrow no\text{-}match_{ML}\ ps\ vs \end{array}$$

and a recursive function

$$\begin{array}{l} no\text{-}match_{ML}\ ps\ os \longleftrightarrow \\ (\exists\, i{<}min\ |os|\ |ps|. \\ \quad \exists\, nm\ nm'\ vs\ vs'. \\ \qquad (rev\ ps)_{[i]} = C_U\ nm\ vs\ \wedge \\ \qquad (rev\ os)_{[i]} = C_U\ nm'\ vs' \wedge (nm = nm' \longrightarrow no\text{-}match_{ML}\ vs\ vs')) \end{array}$$

Notation $xs_{[i]}$ represents the ith element of list xs, and $|xs|$ is the length of xs. Function $no\text{-}match_{ML}$ checks if there is a witness for non-matching, i.e. a clash of two distinct constructors (nm and nm').

The reduction rules for *apply* realize the defining equations for `apply` in Section 2:

$$\begin{array}{l} apply\ (Clo\ f\ vs\ (Suc\ n))\ v \Rightarrow Clo\ f\ (v{\cdot}vs)\ n \quad \text{(if } 0 < n\text{)} \\ apply\ (Clo\ f\ vs\ (Suc\ 0))\ v \Rightarrow A_{ML}\, f\ (v{\cdot}vs) \\ apply\ (C_U\ nm\ vs)\ v \Rightarrow C_U\ nm\ (v{\cdot}vs) \\ apply\ (V_U\ x\ vs)\ v \Rightarrow V_U\ x\ (v{\cdot}vs) \end{array}$$

Note that this is not a definition of *apply*, which is a constructor, but of $\Rightarrow$. Similarly for *term* below.

Finally we have all the context rules (not shown). They say that reduction can occur anywhere, except under a Lam_{ML}. Note that we do not fix lazy or eager evaluation but allow any strategy. Thus we cover different target languages. The price we pay is that we can only show partial correctness because some evaluation strategies may fail to terminate.

3.3.3 Reduction of function term

These reduction rules realize the description of `term` in Section 2:

$$\begin{aligned}
&term\ (C_U\ nm\ vs) \Rightarrow C\ nm \bullet\bullet\ map\ term\ (rev\ vs)\\
&term\ (V_U\ x\ vs) \Rightarrow V\ x \bullet\bullet\ map\ term\ (rev\ vs)\\
&term\ (Clo\ vf\ vs\ n) \Rightarrow \Lambda\ (term\ (apply\ (lift\ 0\ (Clo\ vf\ vs\ n))\ (V_U\ 0\ [])))
\end{aligned}$$

The last clause formalises η-expansion. By lifting, 0 becomes a fresh variable which the closure object is applied to and which is bound by the new Λ.

In addition, we can reduce anywhere in a *tm*:

$$\frac{t \Rightarrow t'}{\Lambda\ t \Rightarrow \Lambda\ t'} \qquad \frac{s \Rightarrow s'}{s \bullet t \Rightarrow s' \bullet t} \qquad \frac{t \Rightarrow t'}{s \bullet t \Rightarrow s \bullet t'} \qquad \frac{v \Rightarrow v'}{term\ v \Rightarrow term\ v'}$$

It should be noted that the reduction $\Rightarrow$ just defined is on type *tm*, whereas the reduction $\Rightarrow$ defined earlier was on type *ml*. We intentionally overloaded $\Rightarrow$ because the one on *tm* also models execution on the ML level, but returns a term.

3.4 Compilation

This section describes our compiler that takes a λ-calculus term and produces an ML term. Its type is $tm \Rightarrow (nat \Rightarrow ml) \Rightarrow ml$ and it is defined for pure terms only:

$$\begin{aligned}
&compile\ (V\ x)\ \sigma = \sigma\ x\\
&compile\ (C\ nm)\ \sigma\\
&= (\textsf{if}\ 0 < arity\ nm\ \textsf{then}\ Clo\ (C_{ML}\ nm)\ []\ (arity\ nm)\ \textsf{else}\ A_{ML}\ (C_{ML}\ nm)\ [])\\
&compile\ (s \bullet t)\ \sigma = apply\ (compile\ s\ \sigma)\ (compile\ t\ \sigma)\\
&compile\ (\Lambda\ t)\ \sigma = Clo\ (Lam_{ML}\ (compile\ t\ (V_{ML}\ 0\ \#\#\ \sigma)))\ []\ 1
\end{aligned}$$

We explain the equations one by one.

1. In the variable case we look the result up in the additional argument σ. This is necessary to distinguish two situations. On the one hand, the compiler is called to compile terms to be reduced. Free variables in those terms must be translated to V_U variables, their embedding in type `Univ`. Function *term* reverses this translation at the end of ML execution. On the other hand, the compiler is also called to compile rewrite rules (R) to ML (*compR*). In this case, free variables must be translated to ML variables which are instantiated by pattern matching when that ML code is executed.
2. A constant becomes a closure with an empty argument list. The counter of missing arguments is set to *arity nm*. Note that our implementation takes care to create only closures with a non-zero counter—otherwise *apply* never fires. This does not show up in our verification because, if *apply* never fires, our computation gets stuck and we will not produce an output, hence, in particular, no unsound or non-normal one.

3. Term application becomes *apply*.
4. Term abstraction becomes a closure containing the translated ML function waiting for a single argument. The construction $V_{ML}\ 0\ \#\#\ \sigma$ is a new substitution that maps 0 to $V_{ML}\ 0$ and $i{+}1$ to $\mathit{lift}_{ML}\ 0\ (\sigma\ i)$. This is the de Bruijn way of moving under an abstraction.

As explained above, the compiler serves two purposes: compiling terms to be executed (where the free variables are fixed) and compiling rules (where the free variables are considered open). These two instances are given separate names:

$$\mathit{comp\text{-}open}\ t = \mathit{compile}\ t\ V_{ML} \qquad \mathit{comp\text{-}fixed}\ t = \mathit{compile}\ t\ (\lambda i.\ V_U\ i\ [])$$

We can now define the set of compiled rewrite rules *compR* as the compilation of *R*.

$$\mathit{compR} = (\lambda(\mathit{nm}, \mathit{ts}, t).\ (\mathit{nm}, \mathit{map\ comp\text{-}pat}\ (\mathit{rev\ ts}), \mathit{comp\text{-}open}\ t))\ `\ R$$

where $f\ `\ M$ is the image of a set under a function. Since compilation moves from the term to the ML level, we need to reverse argument lists. On the left-hand side of each compiled rule this is done explicitly, on the right-hand side it happens implicitly by the interaction of *apply* with closures.

The function *comp-pat* recursively folds $C\ \mathit{nm} \bullet\bullet\ \mathit{xs}$ to $C_U\ \mathit{nm}\ (\mathit{rev\ xs})$. In other words, iterated applications are folded together to application lists, which do not exist at the term level, and arguments are reversed.

We can model the compiled rewrite rules as a set (rather than a list) because the original rewrite rules are already a set and impose no order. For partial correctness it is irrelevant as to which order the clauses are tried in. But for normalisation (Section 3.6), we need to ensure that the default rule is only applied if none of the rewrite rules is applicable. This is the reason why it is modelled separately (in Section 3.3), and not as part of the compiled rules.

3.5 Soundness

The main theorem is partial correctness of compiled evaluation at the ML level with respect to term reduction:

Theorem 1
If $\mathit{pure}\ t$, $\mathit{term}\ (\mathit{comp\text{-}fixed}\ t) \Rightarrow^* t'$ and $\mathit{pure}\ t'$ then $t \rightarrow^* t'$.

Let us examine the key steps in the proof. The two inductive lemmas

Lemma 1
If $\mathit{pure}\ t$ and $\forall i.\ \sigma\ i = V_U\ i\ []$ then $(\mathit{compile}\ t\ \sigma)! = t$.

Lemma 2
If $\mathit{pure}\ t$ and $\forall i.\ \mathit{closed}_{ML}\ n\ (\sigma\ i)$ then $\mathit{closed}_{ML}\ n\ (\mathit{compile}\ t\ \sigma)$.

yield $(\mathit{term}\ (\mathit{comp\text{-}fixed}\ t))! = t$ and $\mathit{closed}_{ML}\ 0\ (\mathit{term}\ (\mathit{comp\text{-}fixed}\ t))$. Then

Theorem 2
If $t \Rightarrow^* t'$ and $\mathit{closed}_{ML}\ 0\ t$ then $t! \rightarrow^* t'! \wedge \mathit{closed}_{ML}\ 0\ t'$.

yields the desired result $t \rightarrow^* t'$ (because $pure\ t' \Longrightarrow t'! = t'$). Theorem 2 is proved by induction on $\Rightarrow^*$ followed by induction on $\Rightarrow$. The inner induction, in the *term* case, requires the same theorem, but now on the ML level:

Theorem 3
If $v \Rightarrow v'$ and $closed_{ML}\ 0\ v$ then $v! \rightarrow^* v'! \wedge closed_{ML}\ 0\ v'$.

This is proved by induction on the reduction $\Rightarrow$ on ML terms. There are two nontrivial cases: β-reduction and application of a compiled rewrite rule. The former requires a delicate and involved lemma about the interaction of the kernel and substitution which is proved by induction on u (and whose proof requires an auxiliary notion of substitution):

Theorem 4
If $closed_{ML}\ 0\ v$ and $closed_{ML}\ (Suc\ 0)\ u$ then $(u[v/0])! = ((lift\ 0\ u)[V_U\ 0\ []/0])!\ [v!/0]$.

The application of a compiled rewrite rule is justified by

Theorem 5
If $(nm, vs, v) \in compR$ and $\forall i.\ closed_{ML}\ 0\ (\sigma\ i)$ then $C\ nm \bullet\bullet (map\ (subst_{ML}\ \sigma)\ (rev\ vs))! \rightarrow^* (subst_{ML}\ \sigma\ v)!$.

That is, taking the kernel of a compiled and instantiated rewrite rule yields a rewrite on the λ-term level.

The proof of Theorem 5 requires one nontrivial inductive lemma:

Lemma 3
If $pure\ t$ and $\forall i.\ closed_{ML}\ 0\ (\sigma\ i)$ then $(subst_{ML}\ \sigma\ (comp\text{-}open\ t))! = subst\ (kernel \circ \sigma)\ t$.

In the proof of Theorem 5 this lemma is applied to *vs* and *v*, which are the output of *comp-open* by definition of *compR*. Hence, we need that all rules in *R* are pure:

$$(nm, ts, t) \in R \Longrightarrow (\forall t{\in}set\ ts.\ pure\ t) \wedge pure\ t$$

This is an axiom because *R* is otherwise arbitrary. It is trivially satisfied by our implementation because the inclusion of *term* as a constructor of λ-terms is an artefact of our model.

3.6 Normalisation

We show that whenever our normalisation routine outputs a term, it is indeed a normal one. In other words, we show the following theorem:

Theorem 6
If $pure\ t$ and $term\ (comp\text{-}fixed\ t) \Rightarrow^* t'$ and $pure\ t'$ then $normal\ t'$.

Here the predicate *normal* is defined inductively in the usual way.

$$\frac{\forall t{\in}set\ ts.\ normal\ t}{normal\ (V\ x \bullet\bullet ts)} \qquad \frac{normal\ t}{normal\ (\Lambda\ t)}$$

$$\frac{\forall\, t{\in}set\ ts.\ normal\ t \qquad \forall\, \sigma.\ \forall\, (nm',\, ls,\, r){\in}R.\ \neg\ (nm\ =\ nm' \wedge\ take\ |ls|\ ts\ =\ map\ (subst\ \sigma)\ ls)}{normal\ (C\ nm\ {\bullet\bullet}\ ts)}$$

The main idea to prove normalisation is to notice that function *term* is defined such that it can output only normal terms. The variable rule says *term* $(V_U\ x\ vs)$ $\Rightarrow$ $V\ x\ {\bullet\bullet}\ map\ term\ (rev\ vs)$ and if *ts* are normal, then so is $V\ x\ {\bullet\bullet}\ ts$. Similarly, the lambda rule says $term\ (Clo\ vf\ vs\ n) \Rightarrow \Lambda\ (term\ (apply\ (lift\ 0\ (Clo\ vf\ vs\ n))\ (V_U\ 0\ [])))$ and if t is normal, so is $\Lambda\ t$. The only non-trivial case is that of a constructor $C_U\ nm$ of the universal data type. In the rule $term\ (C_U\ nm\ vs) \Rightarrow C\ nm\ {\bullet\bullet}\ map\ term\ (rev\ vs)$ we need to know what the arguments *vs* look like. To do so, we introduce a predicate $C\text{-}normal_{ML}$ stating that $C_U\ nm\ vs$ is only formed if no rule for *nm* is applicable. In other words, the main rule of the recursive predicate $C\text{-}normal_{ML}$ is the following:

$$C\text{-}normal_{ML}\ (C_U\ nm\ vs) \longleftrightarrow (\forall\, v{\in}set\ vs.\ C\text{-}normal_{ML}\ v) \wedge no\text{-}match\text{-}compR\ nm\ vs$$

In all other cases $C\text{-}normal_{ML}$ is defined homomorphically.

Since our notion of non-matching is a positive one—i.e., we require a different constructor to be exhibited—adding new arguments to a failing match will preserve this property.

If $no\text{-}match\text{-}compR\ nm\ vs$ then $no\text{-}match\text{-}compR\ nm\ (v{\cdot}vs)$.

Similarly, as reduction will never change outermost C_U constructors, non-matching is preserved under reductions.

If $no\text{-}match_{ML}\ ps\ vs$ and $vs \Rightarrow vs'$ then $no\text{-}match_{ML}\ ps\ vs'$.

Given these two observations, it is easy to show that $C\text{-}normal_{ML}$ is established by compilation (because there are simply no constructors C_U in the output of *compile*) and is invariant under reduction:

Lemma 4
If $pure\ t$ and $\forall\, i.\ C\text{-}normal_{ML}\ (\sigma\ i)$ then $C\text{-}normal_{ML}\ (compile\ t\ \sigma)$.

Lemma 5
If $v \Rightarrow v'$ and $C\text{-}normal_{ML}\ v$ then $C\text{-}normal_{ML}\ v'$.

Now we define a corresponding notion of constructor normality at the term level. However, at the term level the failure of the match is not immediately visible, but only after *term* is evaluated further. We therefore introduce a predicate *C-normal* that anticipates *term* being unfolded further. The only non-homomorphic clause in the inductive definition of *C-normal* is the one for the constructor *C*.

$$\frac{\forall\, t{\in}set\ ts.\ C\text{-}normal\ t \qquad no\text{-}match\text{-}R\ nm\ (map\ dterm\ ts)}{C\text{-}normal\ (C\ nm\ {\bullet\bullet}\ ts)}$$

The actual anticipation is done by the function *dterm* which is homomorphic except

$$dterm\ (term\ v) = dterm_{ML}\ v$$

where dterm_{ML} is defined to be

$$\mathit{dterm}_{ML}\ (C_U\ \mathit{nm}\ \mathit{vs}) = C\ \mathit{nm} \bullet\bullet\ \mathit{map}\ \mathit{dterm}_{ML}\ (\mathit{rev}\ \mathit{vs})$$

and $V\ 0$ in all other cases. For pure terms, *dterm* behaves like the identity.

By means of *dterm*, our notion of non-matching at the ML-level implies non-matching at the term level:

Lemma 6
If $\mathit{no\text{-}match}_{ML}\ \mathit{ps}\ \mathit{vs}$ then $\mathit{no\text{-}match}\ (\mathit{map}\ \mathit{dterm}_{ML}\ (\mathit{rev}\ \mathit{ps}))\ (\mathit{map}\ \mathit{dterm}_{ML}\ (\mathit{rev}\ \mathit{vs}))$.

The predicate *C-normal* is preserved under reduction, and so is the skeleton of the outermost constructors.

Lemma 7
If $t \Rightarrow t'$ and C-normal t then
$\mathit{C\text{-}normal}\ t' \wedge$
$(\mathit{dterm}\ t = C\ \mathit{nm} \bullet\bullet\ \mathit{ts} \longrightarrow$
$\mathit{dterm}\ t' = C\ \mathit{nm} \bullet\bullet\ \mathit{map}\ \mathit{dterm}\ (C_U\text{-}\mathit{args}\ t') \wedge$
$|C_U\text{-}\mathit{args}\ t| = |C_U\text{-}\mathit{args}\ t'| \wedge$
$(\forall i{<}|C_U\text{-}\mathit{args}\ t|.\ (C_U\text{-}\mathit{args}\ t)_{[i]} \Rightarrow^* (C_U\text{-}\mathit{args}\ t')_{[i]}))$

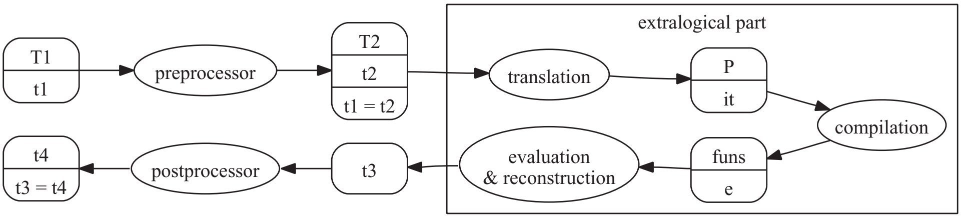
Here C_U-*args* extracts the arguments of a constructor term:

$$\begin{aligned}&C_U\text{-}\mathit{args}\ (s \bullet t) = C_U\text{-}\mathit{args}\ s\ @\ [t]\\ &C_U\text{-}\mathit{args}\ (\mathit{term}\ (C_U\ \mathit{nm}\ \mathit{vs})) = \mathit{map}\ \mathit{term}\ (\mathit{rev}\ \mathit{vs})\\ &C_U\text{-}\mathit{args}\ \_ = []\end{aligned}$$

Now we can prove Theorem 6. Let us examine the only critical reduction $\mathit{term}\ (C_U\ \mathit{nm}\ \mathit{vs}) \Rightarrow C\ \mathit{nm} \bullet\bullet\ \mathit{map}\ \mathit{term}\ (\mathit{rev}\ \mathit{vs})$. Because all ML-values during reduction are $\mathit{C\text{-}normal}_{ML}$, the subterm $C_U\ \mathit{nm}\ \mathit{vs}$ is $\mathit{C\text{-}normal}_{ML}$. By definition, this implies *no-match-compR nm vs*. By Lemma 6 we obtain the corresponding non-matchings at the term level after the application of *dterm*, hence the righ-hand side is *C-normal.* Lemma 7 asserts that the constructor structure is preserved under reduction. But if a reduct happens to be pure, the detour via *dterm* is no longer needed, as it has no effect. Hence, we have constructive non-matching at the term level outright. Theorem 6 follows.

4 Realisation in Isabelle

The implementation of our NBE approach in Isabelle/HOL is based on a generic code generator framework (Haftmann & Nipkow, 2010). The following diagram and description explains how this is connected to the rest of Isabelle. The box labelled "extralogical part" comprises the extension of Isabelle's trusted kernel.



1. The input is an Isabelle term t_1 to be normalised with respect to a set of equational theorems T_1 (and β-reduction). Note that while our formalisation has no notion of order between equations, the implementation necessarily has since it uses ML for evaluation; hence it allows the user to specify an order in which overlapping equations shall be applied.
2. The framework allows one to configure arbitrary logical transformations on input t_1 (and T_1) and output t_3 (pre- and post-processing). This is for the user's convenience and strictly on the level of theorems: both transformations yield equational theorems $t_1 = t_2$ and $t_3 = t_4$; together with the equation $t_2 = t_3$ stemming from the actual evaluation (this is where we have to trust the evaluator!), the desired $t_1 = t_4$ is obtained by transitivity and returned to the user.
3. The main task of the framework is to transform a set of equational theorems T_2 into a program P (and t_2 into *it*) in an abstract intermediate language capturing the essence of languages like SML or Haskell with an equational semantics. The intermediate term language is practically the same as the Isabelle term language, and the equational semantics is preserved in the translation. The key changes are the replacement of an unordered *set* of equational theorems by a structured presentation with explicit dependencies, and, most importantly, the removal of overloading and the dictionary translation of type classes. For details see Haftmann & Nipkow 2010. Inputs to NBE are in this intermediate language. Having compiled away type classes and overloading, NBE operates on terms following the Hindley–Milner type discipline, as assumed in Section 2.
4. P is compiled (via *comp-open*, see Section 3.4) to a series of SML function definitions `funs` and *it* (via *comp-fixed*) to an SML term `e`. Then `term (let funs in e end)` is given to the SML compiler, causing the evaluation of `e` and the translation of the result back into an Isabelle term; type reconstruction (see Section 2) on the result yields t_3.

We conducted a number of timing measurements to determine the relative performance of NBE with respect to two other normalisation mechanisms available in Isabelle:

eval, the ground evaluator which compiles terms and theorems directly to SML, without support for open terms. It uses the same code generator framework but defines a native SML data type for each Isabelle data type, rather than operating on a universal data type. For details see Haftmann & Nipkow 2010.

simp, the symbolic simplifier which operates on the level of Isabelle terms and theorems and produces a theorem purely by inference.

Our setup for this experiment ensures that all three evaluators use the same equational theorems and the same reduction strategy.

We measured the performance of three different programs: *ack* computes the Ackermann function of 3 and 12 (in successor representation); *rotate* rotates a list of 10 integers $5 * 10^7$ times; *sort* sorts the list of integers from 10,000 down to 0 into ascending order by insertion sort.

	ack		*rotate*		*sort*	
	absolute	relative	absolute	relative	absolute	relative
eval	5	.07	17	.07	5	.05
nbe	70	1	250	1	110	1
simp	3163	45	$3.2 * 10^6$	128	$14 * 10^6$	1287

Absolute figures are in seconds using Isabelle 2011 with PolyML 5.3.0 on a MacBook Pro with a 2.66 GHz Intel processor.

Since all examples should be runnable with each of the evaluation mechanisms, they only involve ground terms (necessary for *eval* — ground terms do not speed up *nbe* or *simp*). In order to obtain reliable figures, we had to use inputs that trigger long computations. However, *simp* does not cope with such large examples (it actually runs out of space on *sort*). As a result, the *simp* times for *sort* and *rotate* are estimates extrapolating the running times on smaller inputs.

Unsurprisingly, *nbe* turns out to be faster than *simp* and slower than *eval*. There is a trade-off between performance and expressiveness. While *eval* is fast, it can evaluate only closed terms. Furthermore, if the result of *eval* is to be "read back" as an Isabelle term, it must only contain constructors and no function values. Finally, *eval* cannot cope with additional rewrite rules like associativity. With a performance penalty of a factor of 10–20, *nbe* can lift all these restrictions, while still outperforming the simplifier by a factor of 50 or much more.

In the above examples, compilation time is about 0.1 second or less for *eval* and *nbe*; *simp* requires no compilation at all, the necessary data structures are maintained incrementally. For terms involving many data type and function definitions, compilation time can obviously increase. For typical examples it is still in the range of 1 second. One atypical example (Nipkow *et al.*, 2006) produces 1200 lines ML for *eval*; generating and compiling it takes about 1 second. But generating and compiling the corresponding more complicated *nbe* code takes about 40 seconds. When we investigated this, we found that 90% of the time was spent in the ML compiler (PolyML): one particular function definition yielded very complicated ML patterns that took 30 seconds to compile.

5 Extensions

5.1 Non-left-linear rules

In our modelling of the translation, pattern matching was quite liberal. In particular, non-left-linear equations were admissible. However, an actual programming language like ML does not allow such patterns. Nevertheless, they can be desirable in proven rewrite rules. Examples include the defining equation (x = x) = True for equality, and x - x = 0.

Fortunately, equality can be approximated from below. That is, we can define a function same on our universal type such that, if same u v evaluates to True then u and v definitely denote the same value; moreover, the definition is such that in enough cases equality is actually realised and the function is still efficient. We use the following definition of same:

```
fun same (Const (k, xs), Const (l, ys)) =
      k = l andalso sames (xs, ys)
  | same (Var (k, xs), Var (l, ys)) =
      k = l andalso sames (xs, ys)
  | same _ = false
and sames ([], []) = true
  | sames (x :: xs, y :: ys) = same (x, y) andalso sames (xs, ys)
  | sames _ = false;
```

Using this function same, non-left-linear rules can be added by using different variables for each occurrence of a variable used multiple times on the left-hand side, and requiring these variables to be the same. In Haskell, we could write such an equation simply by conditional pattern matching.

```
minus [x, y] | same x y  = Const "0" []
...
```

Since ML does not natively support guarded pattern matching, we implement it by a sequence of functions, each handling one pattern and calling the next function, if the pattern matching fails. With this technique, the above conditional pattern matching is implemented as

```
fun minus_1 [x, y] =
     if same x y then Const ("0", []) else minus_2 [x, y]
  | minus_1 args  = minus_2 args
fun minus_2 ...
```

This extension adds flexibility to the allowed rewrite rules. However, the under-approximation of same comes at the price that the default rule is used even in cases where a non-left-linear rule might be applicable but this is not detected as there are lambda-abstractions in the non-left-linear positions. Hence, non-left-linear rules mean that we can no longer guarantee normality of the result computed by our implementation.

Correctness of the result still holds: in our model of ML, we allow non-left-linear function definitions. To prove that this carries over to real ML, one would need to prove that the above schema for translating non-left-linear functions into left-linear ones with `same` is correct: any reduction of the translated function is justified by a reduction with the original function.

5.2 Case combinators

A second extension of the formalised behaviour is needed, as the call-by-value semantics of ML sometimes leads to undesirable evaluation behaviour. In particular, a naive implementation of the "if" construct as ternary function would not work in recursive definitions like

```
fib n = if (n <= 2) 1 (fib (n - 1) + fib (n - 2))
```

The problem is, that in a call-by-value language, the else-branch would be evaluated unconditionally, leading to non-termination. For this reason, languages like ML treat "if" as a special syntactic construct, not as an ordinary function. To be able to use normalisation by evaluation for functions like `fib` we need to treat the "if" construct special as well.

For this, we equip our code generation with an explicit notion of "case combinators". A constant C can act as a case combinator if equations of the following form are proven.

$$\begin{array}{rcl} C\ w_1 \cdots\ w_n\ (c_1\ \overline{x_1}) & = & w_1\ \overline{x_1} \\ & \vdots & \\ C\ w_1 \cdots\ w_n\ (c_n\ \overline{x_n}) & = & w_n\ \overline{x_n} \end{array}$$

These equations are proved automatically for standard constructs like "if" and cases on data types. They justify the following translation of $C\ w_1 \cdots\ w_n\ t$ into ML:

```
  case t of Const (c_1, ts) => foldl apply w_1 ts
          | ...
          | Const (c_n, ts) => foldl apply w_n ts
          | _  => C_next_case
```

Here `C_next_case` simulates the failure of the pattern matching for this equation: it calls the next defining equation of the surrounding pattern matching, possibly the default equation. An alternative would have been to leave `t` unchanged on failure, but practical experience has shown that propagating a case match failure to a match failure of the whole equation yields evaluation results which are easier to interpret for the human user. This propagation is admissible since it does not violate partial correctness.

For our example function `fib` the code in Figure 1 is generated. Note that the recursive case is only evaluated once we have established that $n \leqslant 2$ evaluates to `False`, i.e. only once we have positive knowledge that this case is to be used.

Correctness of this extension with case combinators is easy because we only generate case constructs in ML for functions where the user has provided equational theorems that prove that the function acts like a case combinator. As pointed out

```
fun fib [n] = case less_eq [n, Const ("2", [])]
 of Const ("True", []) => Const ("1", [])
  | Const ("False", []) =>
      apply
       (apply plus
        (fib [apply (apply minus n) (Const ("1", []))]))
        (fib [apply (apply minus n) (Const ("2", []))])
  | _ => fib_2 [n]
fun fib_2 [n] = Const("fib", [n])
```

Fig. 1. Translation of fib.

above, normality of the result is no longer guaranteed because of readability reasons: equations where the right-hand side starts with a `case t of` are only applied if the `t` starts with a constructor, i.e. the case combinator can be contracted.

6 Related work

The work probably most closely related to ours is that of Berger, Eberl, and Schwichtenberg (Berger *et al.* 1998, 2003) who also integrated NBE into a proof assistant. However, their approach is based on a type-indexed semantics with constructors coinciding with those of the object language. Besides the administrative hassle, the commitment to a particular type system in the object language, and unneeded and unwanted η-expansions, the main disadvantage of this choice is that functions, like the append function in our example in Section 2, cannot serve the role as additional constructors. Note that in our example, this usage of an append constructor made it possible to effortlessly incorporate associativity into the definition of the function append, with pattern matching directly inherited from the implementation language.

The unavailability of the shape of a semantic object, unless it is built from a canonical constructor of some ground type, made it necessary in the approach by Berger *et al.* to revert to the term representation. This led to the artificial (at least from a user's point of view) and somewhat obscure distinction between so-called "computational rules" and "proper rewrite rules" where only the former are handled by NBE. The latter are carried out at a symbolic level (using pattern matching on the term representation). This mixture of computations on the term representation and in the implementation language requires a continuous changing between both representations. In fact, one full evaluation and reification is performed for each single usage of a rewrite rule.

Two other theorem proving systems provide specialised efficient normalisers for open λ-terms. Both of them are based on *abstract machines* and are therefore complementary to our compiled approach:

- Barras (Barras, 2000) extends the HOL (Gordon & Melham, 1993) system with an abstract reduction machine for efficient rewriting. It is as general as our approach and even goes through the inference kernel. For efficiency reasons, HOL's term language was extended with explicit substitutions.

- Grégoire and Leroy (Grégoire & Leroy, 2002) present and verify a modification of the abstract machine underling OCaml. This modified abstract machine has become part of Coq's trusted proof kernel. The main difference is that they cannot deal with additional rewrite rules like associativity.

Compiled approaches to rewriting of first-order terms can also be found in other theorem provers, e.g. KIV (Reif *et al.*, 1998).

Boesplfug (Boespflug, 2010) discusses variations of untyped normalisation by evaluation. The main emphasis is on comparing performance of various optimisations of the generated code. In particular, it is pointed out that performance can be increased considerably by using native ML-constructors for the object language constructors rather than representing all of them via a single `Const` as we do.

References

Aehlig, K. & Joachimski, F. (2004) Operational aspects of untyped normalization by evaluation. *Math. Struct. Comput. Sci.* **14**(4), 587–611.

Aehlig, K., Haftmann, F. & Nipkow, T. (2008) A compiled implementation of normalization by evaluation. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, Mohamed, O.A., Muñoz, C., & Tahar, S. (eds), *Lecture Notes in Computer Science,* vol. 5170. Springer Verlag, pp. 39–54.

Barras, B. (2000) Programming and computing in HOL. In *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, Aagaard, M. & Harrison, J. (eds), *Lecture Notes in Computer Science,* vol. 1869. Springer Verlag, pp. 17–37.

Berger, U., Eberl, M. & Schwichtenberg, H. (2003) Term rewriting for normalization by evaluation. *Inf. Comput.* **183**, 19–42.

Berger, U. & Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed λ–calculus. In *Proceedings of The Sixth IEEE Symposium on Logic in Computer Science, LICS 1991*, Vemuri, R. (ed), pp. 203–211.

Berger, U., Eberl, M. & Schwichtenberg, H. (1998) Normalization by evaluation. In *Prospects for Hardware Foundations*, Möller, B. & Tucker, J.V. (eds), *Lecture Notes in Computer Science,* vol. 1546. Springer Verlag, pp. 117–137.

Boespflug, M. (2010) Conversion by evaluation. In *Proceedings of the Twelfth International Symposium on Pracical Aspects of Declarative Languages (PADL '10)*, Carro, M. & Peña, R. (eds), *Lecture Notes in Computer Science,* vol. 5937. Springer Verlag, pp. 58–72.

Bruijn, N. G. de. (1972) Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Math.* **34**, 381–392.

Danvy, O. (1996) Type-directed partial evaluation. In *Proceedings of the Twenty-third ACM Symposium on Priciples of Programming Languages (POPL 1996)*, pp. 242–257.

Gonthier, G. (2008) Formal proof—the four-color theorem. *Not. AMS.* **55**, 1382–1393.

Gordon, M. J. C. & Melham, T. F. (eds). (1993) *Introduction to HOL: A Theorem-Proving Environment for Higher Order Logic*. Cambridge University Press.

Grégoire, B. & Leroy, X. (2002) A compiled implementation of strong reduction. In *International Conference on Functional Programming (ICFP 2002)*. ACM Press, pp. 235–246.

Haftmann, F. & Nipkow, T. (2010) Code generation via higher-order rewrite systems. In *Functional and Logic Programming, FLOPS 2010*, Blume, M., Kobayashi, N. & Vidal, G. (eds), *Lecture Notes in Computer Science,* vol. 6009. Springer Verlag, pp. 103–117.

Krauss, A. (2006) Partial recursive functions in higher-order logic. In *Automated Reasoning (IJCAR 2006)*, Furbach, U. & Shankar, N. (eds), *Lecture Notes in Computer Science,* vol. 4130. Springer Verlag, pp. 589–603.

Nipkow, T., Paulson, L., & Wenzel, M. (2002) *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, *Lecture Notes in Computer Science,* vol. 2283. Springer Verlag.

Nipkow, T., Bauer, G., & Schultz, P. (2006) Flyspeck I: Tame graphs. In *Automated Reasoning (IJCAR 2006)*, Furbach, U. & Shankar, N. (eds), *Lecture Notes in Computer Science,* vol. 4130. Springer Verlag, pp. 21–35.

Reif, W., Schellhorn, G., Stenzel, K., & Balser, M. (1998) Structured specifications and interactive proofs with KIV. In *Automated Deduction—A Basis for Applications*, Bibel, W., & Schmitt, P. (eds), *Systems and Implementation Techniques*, vol. II. Kluwer, pp. 13–39.