

A Compiled Implementation of Normalization by Evaluation

Klaus Aehlig^{1*} and Florian Haftmann^{2**} and Tobias Nipkow²

¹ Department of Computer Science, Swansea University

² Institut für Informatik, Technische Universität München

Abstract. We present a novel compiled approach to Normalization by Evaluation (NBE) for ML-like languages. It supports efficient normalization of open λ -terms w.r.t. 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 normalization of terms w.r.t. user provided rewrite rules is one of the central tasks of any theorem prover. Several theorem provers (see §5) provide especially efficient normalizers which have been used to great effect [9,14] in carrying out massive computations during proofs. Existing implementations perform normalization of open terms either by compilation to an abstract machine or by Normalization by Evaluation, NBE for short. The idea of NBE is to carry out the computations by translating into some underlying functional 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 a decent functional language, while allowing the normalization of open λ -terms w.r.t. β -reduction and a set of (possibly higher-order) rewrite rules.
2. A formal model and correctness proof³ of our approach in Isabelle/HOL [15].

NBE is implemented and available at the user-level in Isabelle 2007, both to obtain the normal form t' of some given term t , and as a proof rule that yields the theorem $t = t'$.

Throughout the paper we refer to the underlying functional language as ML. This is only for brevity: any language in the ML family, including Haskell, is suitable. However, we assume that the language implementation provides its own evaluator at runtime, usually in the form of some compiler. The guiding

* Partially supported by grant EP/D03809X/1 of the British Engineering and Physical Sciences Research Council (EPSRC).

** Supported by DFG grant Ni 491/10-1

³ Available online at afp.sf.net

principle of our realization of NBE is to offload 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. The trusted basis of the theorem prover is not extended if the compiler used at runtime is the same as the one compiling the theorem prover.

2 Normalization by Evaluation in ML

Normalization by Evaluation uses the evaluation mechanism of an underlying metalanguage to normalize terms, typically of the λ -calculus. By means of an evaluation function $[\]_{\xi}$, or, alternatively by compiling and running the compiled code, terms are embedded into this metalanguage. 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” [5], serves to recover terms from the semantics. This process is also known as “type-directed partial evaluation” [7].

Normalization by Evaluation is best understood by assuming a semantics enjoying the following two properties.

- *Soundness*: if $r \rightarrow s$ then $[r]_{\xi} = [s]_{\xi}$, for any valuation ξ .
- *Reproduction*: if r is a term in normal form, then $\downarrow [r]_{\uparrow} = r$ with \uparrow a special valuation.

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 $\downarrow [r]_{\uparrow} = \downarrow [s]_{\uparrow} = s$.

We implement untyped normalization by evaluation [1] 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 int * Univ list
  | Clo of int * (Univ list -> Univ) * Univ list
```

Note how the constructors of the data type allow to distinguish between basic types and proper functions of implementation language. 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 what argument will come restricts the implementation to a particular typing discipline, whereas our untyped approach is flexible enough to work with any form of rewrite calculus.

The data type `Univ` represents the embedding of the syntax and the embedding of the function space. There is no constructor for application. The reason is that semantical values of the λ -calculus correspond to normal terms, whereas an

application of a function to some other value, in general, yields a redex. Therefore application is implemented by a function `apply: Univ -> Univ -> Univ` discussed below. The constructor `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 $C t_1 \dots t_k$ 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 $x t_1 \dots t_k$ with x a variable.

The constructor `Clo` represents partially applied functions. More precisely, “`Clo (n, f, [ak, . . . , a1])`” represents the $(n+k)$ -ary function f applied to a_1, \dots, a_k . This expression needs another n arguments 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 “`fn x => . . .`” function abstraction. 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 representation for infinite terms. However, this will not be a problem for our implementation, for several reasons. First of all, we only claim that terms are normalised correctly—this suffices for our procedure to be admissible in a theorem prover. During that normalisation process, only function that can be named by a (finite) term will occur as arguments to `Clo`. Moreover, only needing partial correctness, we will only ever be concerned with semantical values where our \downarrow -function terminates. But then, the fact that it did terminate, witnesses that the semantical value has a finite representation by one of our terms.

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 $C t_1 \dots t_k$ or $x t_1 \dots t_k$ 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 then 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 (1, f, xs)) x = f (x :: xs)
  | apply (Clo (n, f, xs)) x = Clo (n - 1, f, x :: xs)
  | 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 semantical universe `Univ` allows only normal values. Therefore, this call carries out all the normalization work.

As an example, consider a function `append` defined in some Isabelle/HOL theory `T` based on the type `list` defined in theory `List`

```
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)`” was proved. Compiling these equations together with associativity of `append` yields the following ML code.

```

fun T_append [v_cs, Nbe.Const ("T.append", [v_bs, v_as])] =
  T_append [T_append [v_cs, v_bs], v_as]
| T_append [v_bs, Nbe.Const ("List.Cons", [v_as, v_a])] =
  Nbe.Const ("List.Cons", [T_append [v_bs, v_as], v_a])
| T_append [v_bs, Nbe.Const ("List.Nil", [])] =
  v_bs
| T_append [v_a, v_b] =
  Nbe.Const ("T.append", [v_a, v_b])

```

The second and third clause of the function definition are in one-to-one correspondence with the definition of the function `append` in the theory. 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 “ $f [a_n, \dots, a_1]$ ”.

The last clause is a *default clause* fulfilling the need that the ML pattern matching be exhaustive. 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.

Continuing the example, now assume that we want to normalise the expression “`append [a,b] [c]`”. Then the following compiled version of this expression would be evaluated to obtain an element of `Univ`.

```

(Nbe.apply
  (Nbe.apply
    (Clo (2,T_append, []))
    (Nbe.Const ("List.cons",
      [(Nbe.Const ("List.cons",
        [(Nbe.Const ("List.nil", [])),
          (Nbe.free "b")]))],
      (Nbe.free "a")))))

```

```
(Nbe.Const ("List.cons", [(Nbe.Const ("List.nil", [])),
                          (Nbe.free "c")]))))
```

As discussed, values of type `Univ` represent normal terms. Therefore we can easily implement the \downarrow -function, which will be called `term` in our implementation. The function `term` returns a normal term representing a given element of `Univ`. For values of the form “`Const name [vn, . . . , v1]`” we take the constant C named by the string, recursively apply `term` to v_1, \dots, v_n , obtaining t_1, \dots, t_n , and build the application $C t_1 \dots t_n$. Here, again, we keep in mind that arguments are in reverse order in the implementation. The definition in the case of a variable is similar. In the case $v = \text{“Clo ...”}$ of a closure we just carry out an eta expansion: the value denotes a function that needs at least another argument, so we can always write it as $\lambda x. \text{term}(v x)$, with x a fresh syntactical variable. Naturally, this application of v to the fresh variable x is done 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 “ $\lambda u. u$ ”.

Immediately from the definition we note that `term` can only output normal terms. Indeed, the `Const` construct is used only for constructors or functions where the arguments are of such a shape that no redex can occur. Expressions of the shape $x t_1 \dots t_k$ and $\lambda x. t$ are always normal if t, t_1, \dots, t_k are; the latter we can assume by induction hypothesis. Note that we have shown the normality of the output essentially by considering ways to combine terms that preserve the normality. In fact, the normalisation property of normalisation by evaluation can be shown entirely by considering an appropriate typing discipline [8].

Compared to the expressivity of the underlying term language in Isabelle, our universal datatype 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 §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 normalization; 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.

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 “ $(\text{'a} \Rightarrow \text{'a}) \Rightarrow \text{bool}$ ”, say without any rewrite rule. Then the untyped normal form of “ $f (\lambda u : \text{bool}. u)$ ” would be “ $f (\lambda u. u)$ ” with most general type annotations “ $f (\lambda u : \text{'a}. u)$ ”. To avoid such widening of types only those equations will be considered as being proved by normalization 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 w.r.t. 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 been obtained by term rewriting with equations that are consequences of the theory.

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 w.r.t. 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 the introduction it was mentioned that Normalization by Evaluation is best understood in terms of the mentioned properties “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. Then, the lack of this properties requires us to show a slightly stronger form of the reproduction property. We need to for *arbitrary* terms r that $\downarrow [r]_{\uparrow}$ is, if defined, a term that our theory equates with r . To show this property, we give a model of our implementation language and assign each internal state a “denoted term”; having this term denotation at hand we just have to show that each step our

machine model makes either doesn't 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 [6] 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 modeled as a recursive **datatype**:

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

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 value to a list of ML values 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} .
- Values of the datatype **Univ** (§2) are encoded by the constructors C_U, V_U and Clo .
- Constructor *apply* represents the ML function **apply** (§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.

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

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

$$tm = C \text{ cname} \mid V \text{ vname} \mid tm \cdot tm \mid \Lambda \text{ tm} \mid \text{term } ml$$

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

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

We abbreviate $(\dots(t \cdot t_1) \cdot \dots) \cdot t_n$ by $t \cdot\cdot [t_1, \dots, t_n]$. We have the usual lifting and substitution functions for term variables. Capture-avoiding substitution $subst \sigma s$, where $\sigma :: nat \Rightarrow tm$, replaces $V i$ by σi 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 $\geq i$ is written $lift i$ 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 $kernel :: ml \Rightarrow tm$ that maps closed ML terms to λ -terms. For succinctness $kernel$ is written as a postfix `!`; $map \text{ kernel } vs$ is abbreviated to $vs!$. Note that postfix binds tighter than prefix, i.e. $f \ v!$ is $f (v!)$.

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

The arguments lists vs need to be reversed because, as explained in §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 kernel of a tm , also written $t!$, replaces all subterms $term \ v$ 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 [13] 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: \rightarrow on pure terms, the usual λ -calculus reductions, and \Rightarrow on ML terms, which models evaluation in functional languages.

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

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

and context rules:

$$\frac{t \rightarrow t'}{\Lambda t \rightarrow \Lambda t'} \quad \frac{s \rightarrow s'}{s \cdot t \rightarrow s' \cdot t} \quad \frac{t \rightarrow t'}{s \cdot t \rightarrow s \cdot t'}$$

Note that $R :: (cname \times tm \text{ list} \times tm) \text{ set}$ is a global constant that models a (fixed) set of rewrite rules. The triple (f, ts, t) models the rewrite rule $C f \cdot ts \rightarrow t$.

Just like \rightarrow depends on R , \Rightarrow depends on a compiled version of the rules, called $compR :: (cname \times ml \text{ list} \times ml) \text{ set}$. 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.

The ML reduction rules come in three groups. First we have β -reduction $A_{ML} (Lam_{ML} u) [v] \Rightarrow u[v/0]$ and invocation of a compiled function:

$$\frac{(nm, vs, v) \in compR \quad \forall i. \text{closed}_{ML} 0 (\sigma i)}{A_{ML} (C_{ML} nm) (\text{map} (\text{subst}_{ML} \sigma) vs) \Rightarrow \text{subst}_{ML} \sigma v}$$

This is simply one reduction step on the level of ML terms.

Then we have the reduction rules for function *apply*:

$$\frac{0 < n}{\text{apply} (Clo f vs (Suc n)) v \Rightarrow Clo f (v \cdot vs) n}$$

$$\text{apply} (Clo f vs (Suc 0)) v \Rightarrow A_{ML} f (v \cdot vs)$$

$$\text{apply} (C_U nm vs) v \Rightarrow C_U nm (v \cdot vs)$$

$$\text{apply} (V_U x vs) v \Rightarrow V_U x (v \cdot vs)$$

which directly realize the defining equations for **apply** in §2.

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 λ -calculus terms may contain *term*, they too reduce via \Rightarrow . These reduction rules realize the description of **term** in §2:

$$\text{term} (C_U nm vs) \Rightarrow C nm \cdot \text{map term} (\text{rev vs})$$

$$\text{term} (V_U x vs) \Rightarrow V x \cdot \text{map term} (\text{rev vs})$$

$$\text{term} (Clo vf vs n) \Rightarrow \Lambda (\text{term} (\text{apply} (\text{lift } 0 (Clo vf vs n)) (V_U 0 [])))$$

The last clause formalizes 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 :

$$\frac{t \Rightarrow t'}{\Lambda t \Rightarrow \Lambda t'} \quad \frac{s \Rightarrow s'}{s \cdot t \Rightarrow s' \cdot t} \quad \frac{t \Rightarrow t'}{s \cdot t \Rightarrow s \cdot t'} \quad \frac{v \Rightarrow v'}{\text{term } v \Rightarrow \text{term } v'}$$

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:

$$\begin{aligned} \text{compile } (V x) \sigma &= \sigma x \\ \text{compile } (C nm) \sigma &= \text{Clo } (C_{ML} nm) [] (\text{arity } nm) \\ \text{compile } (s \cdot t) \sigma &= \text{apply } (\text{compile } s \sigma) (\text{compile } t \sigma) \\ \text{compile } (\Lambda t) \sigma &= \text{Clo } (\text{Lam}_{ML} (\text{compile } t (V_{ML} 0 \#\#\sigma))) [] 1 \end{aligned}$$

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*, where *arity* is a global table mapping each constant to the number of arguments it expects. 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 we only show partial correctness: even though the output would not be normal, it still would be a reduct of the input.
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 $\text{lift}_{ML} 0 (\sigma i)$. This is the de Bruijn way of moving under an abstraction.

Note that our actual compiler avoids building intermediate closures that are directly applied to an argument.

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:

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

We can now define the set of compiled rewrite rules $compR$ as the union of the compilation of R and the default rules (§2) for each defined function symbol

$$\begin{aligned}
compR = & \\
& (\lambda(nm, ts, t). (nm, map\ comp-open\ (rev\ ts),\ comp-open\ t)) \text{ ‘ } R \cup \\
& (\lambda(nm, ts, t). let\ vs = map\ V_{ML}\ [0..<arity\ nm]\ in\ (nm, vs, C_U\ nm\ vs)) \text{ ‘ } R
\end{aligned}$$

where $f \text{ ‘ } M$ is the image of a set under a function and $[m..<n]$ is the list $[m, \dots, n-1]$. Since compilation moves from the term to the ML level, we need to reverse argument lists. On the left-hand sides of each compiled rule this is done explicitly, on the right-hand side it happens implicitly by the interaction of *apply* with closures. For the default rewrite rules no reversal is necessary.

We can model the compiled rewrite rule 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 in which order the clauses are tried. If the default rule is chosen, no reduction occurs, which is correct, too. Of course the actual implementation puts the default clause last. The implementation also ensures that in all clauses $f p_1 \dots p_n = t$ for some function f , n is the same: additional parameters can always be added by extensionality.

3.5 Verification

The main theorem is partial correctness of compiled evaluation at the ML level w.r.t. term reduction:

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

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

Lemma 2. *If pure t and $\forall i. \sigma\ i = V_U\ i\ []$ then $(compile\ t\ \sigma)! = t$.*

Lemma 3. *If pure t and $\forall i. closed_{ML}\ n\ (\sigma\ i)$ then $closed_{ML}\ n\ (compile\ t\ \sigma)$.*

yield $(term\ (comp-fixed\ t))! = t$ and $closed_{ML}\ 0\ (term\ (comp-fixed\ t))$. Then

Theorem 4. *If $t \Rightarrow^* t'$ and $closed_{ML}\ 0\ t$ then $t! \rightarrow^* t'! \wedge closed_{ML}\ 0\ t'$.*

yields the desired result $t \rightarrow^* t'$ (because $pure\ t' \implies t'! = t'$). Theorem 4 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 5. *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 6. *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 7. *If $(nm, vs, v) \in compR$ and $\forall i. closed_{ML} 0 (\sigma i)$ then $C nm \cdot (map (subst_{ML} \sigma) (rev vs))! \rightarrow^* (subst_{ML} \sigma v)!$.*

That is, taking the kernel of a compiled and instantiated rewrite rule yields a rewrite on the λ -term level. The conclusion is expressed with \rightarrow^* rather than \rightarrow because the rule in $compR$ may also be a default rule, in which case both sides become identical.

The proof of Theorem 7 requires one nontrivial inductive lemma:

Lemma 8. *If pure t and $\forall i. closed_{ML} 0 (\sigma i)$ then $(subst_{ML} \sigma (comp-open t))! = subst (kernel \circ \sigma) t$.*

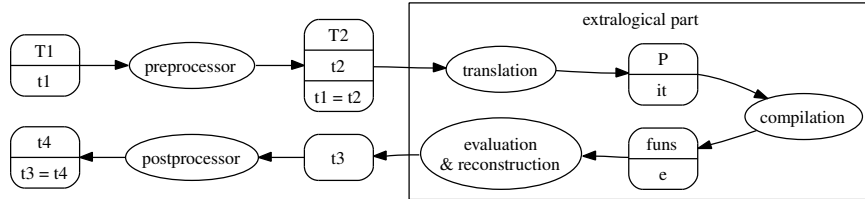
In the proof of Theorem 7 this lemma is applied to vs and v , which are the output of $comp-open$ by definition of $compR$. Hence we need that all rules in R are pure:

$$(nm, ts, t) \in R \implies (\forall t \in set\ ts. pure\ t) \wedge pure\ t$$

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.

4 Realization in Isabelle

The implementation of our NBE approach in Isabelle/HOL is based on a generic code generator framework [12]. The following diagram and description explains how this is connected to the rest of Isabelle:



1. The input is an Isabelle term t_1 to be normalized w.r.t. a set of equational theorems T_1 (and β -reduction). Until evaluation both t_1 and T_1 are processed in parallel.
2. The framework allows one to configure arbitrary logical transformations on input t_1 (and T_1) and output t_3 (pre- and postprocessing). 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 [12]. 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 §2.
4. P is compiled (via *comp-open*, see §3.4) to a series of SML function definitions **fun**s 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 §2) on the result yields t_3 .

We conducted a number of timing measurements to determine the relative performance of NBE w.r.t. two other normalization mechanisms available in Isabelle:

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

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 datatype for each Isabelle datatype, rather than operating on a universal datatype. For details see [12].

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: *eras* computes the first 100 prime numbers using the Sieve of Eratosthenes in a symbolic and naive implementation; *graph* computes the strongly connected components of a graph represented as a finite set of pairs; *sort* sorts a list of strings by insertion sort:⁴

	<i>eras</i>		<i>graph</i>		<i>sort</i>	
<i>simp</i>	4304	1384%	222717	11404%	1451169	22488%
<i>nbe_s</i>	339	109%	3312	170%	11174	173%
<i>nbe</i>	311	100%	1953	100%	6453	100%
<i>eval</i>	48	15%	292	15%	393	6%

Unsurprisingly, *nbe* turns out to be faster than *simp* and slower than *eval*. However the relative differences increase from left to right. In this order also the use of pattern matching in the examples increases. This shows the superiority

⁴ Absolute figures are in milliseconds using Isabelle 2007 with PolyML 5.1 on a Linux 2.6 AMD 1 GHz machine

of native pattern matching as exploited by *eval* over the pattern matching via strings in some universal datatype as required by *nbe*, which is in turn superior to pattern matching programmed in SML as in *simp*. This relevance of pattern matching motivated us to use integers (not strings) to identify constant names in patterns. Indeed, if we use an implementation using strings for constant names (*nbe_s*), there is a considerable loss of efficiency.

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 comparably small performance penalty *nbe* can lift all these restrictions, while still outperforming the simplifier by 1–2 orders of magnitude.

5 Related Work

The work probably most closely related to ours is that of Berger, Eberl, and Schwichtenberg [3,4] 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 §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 `T_append`, with pattern matching directly inherited from the implementation language.

The unavailability of the shape of a semantical 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.

Following Aehlig and Joachimski [1], our proof shows again that correctness of NBE is completely independent of any type system. In particular, no new version of NBE has to be invented each and every time it is applied to some term system with a different typing discipline. There simply is no need for logical relations in the proof.

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

- Barras [2] extends the HOL [10] 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 [11] present and verify a modification of the abstract machine underlying 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 [17].

6 Future Work

A small extension of the formalization is the straightforward proof normality of the output (see §2). More interesting are extensions of the class of permitted rewrite rules:

- Currently the implementation inherits ML’s restriction to left-linear rules. It can be lifted to allow repeated variables on the left-hand side roughly as follows: make all variables distinct on the left-hand side but check for equality on the right-hand side. The details are more involved.
- More adventurous generalizations include ordered rewriting (where a rewrite rule only fires if certain ordering constraints are met) and conditional rewriting. The former should be easy to add, the latter would require a nontrivial generalization of the underlying code generator framework.

It would also be interesting to model λ -terms by different means than de Bruijn indices. Particularly prominent is the nominal approach [16] and its realisation by Urban [18] in Isabelle. As about one third of our proofs are primarily concerned with de Bruijn indices, it would be an interesting comparison to redo the verification in the nominal setup. Our preference for de Bruijn terms is due to the fact that the current implementation of nominal data types in Isabelle does not support nested data types, where recursion is through some other data type like *list*, which occurs in our model of ML terms.

References

1. Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14(4):587–611, August 2004.
2. Bruno Barras. Programming and computing in HOL. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, Lecture Notes in Computer Science, pages 17–37. Springer Verlag, 2000.
3. U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183:19–42, 2003.

4. Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer Verlag, 1998.
5. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 203–211, 1991.
6. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
7. Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the Twenty-Third Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL '96)*, 1996.
8. Olivier Danvy, Morten Rhiger, and Christopher H. Rose. Normalisation by evaluation with typed syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.
9. Georges Gonthier. A computer-checked proof of the four-colour theorem. <http://research.microsoft.com/~gonthier/4colproof.pdf>.
10. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
11. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
12. Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*. Department of Computer Science, University of Kaiserslautern, 2007.
13. Alexander Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer Verlag, 2006.
14. Tobias Nipkow, Gertrud Bauer, and Paula Schultze. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 21–35. Springer Verlag, 2006.
15. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.
16. Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
17. Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, pages 13 – 39. Kluwer, 1998.
18. Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *Automated Deduction — CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag, 2005.