Efficient Normalization by Evaluation

Mathieu Boespflug

École Polytechnique, INRIA

mboes@lix.polytechnique.fr

Abstract

Dependently typed theorem provers allow arbitrary terms in types. It is convenient to identify large classes of terms during type checking, hence many such systems provision some form of conversion rule. A standard algorithm for testing the convertibility of two types consists in normalizing them, then testing for syntactic equality of the normal forms. Normalization by evaluation is a standard technique enabling the use of existing compilers and runtimes for functional languages to implement normalizers, without peaking under the hood, for a fast yet cheap system in terms of implementation effort.

Our focus is on performance of untyped normalization by evaluation. We demonstrate that with the aid of a standard optimization for higher order programs (namely uncurrying) and the reuse of pattern matching facilities of the evaluator for datatypes, we may obtain a normalizer that evaluates non-functional values about as fast as the underlying evaluator, but as an added benefit can also fully normalize functional values — or to put it another way, partially evaluates functions efficiently.

1. Introduction

The objective here is to achieve efficient strong reduction (or full normalization) of terms in the λ-calculus. By *strong* reduction we mean the β-reduction of all redexes in a term, including inside functional values. By efficient we mean speedy execution on stock hardware.

Most implementations of the λ-calculus, such as those underpinning many functional languages, only implement *weak* reduction (also called *evaluation*). That is, reduction never occurs inside function bodies until these functions are applied to actual arguments. But looking further afield to other places where the λ-calculus is to be found, weak reduction is not always enough.

Dependently typed theories underlie many proof assistants such as Agda, Coq, or Epigram. Because types may contain arbitrary terms in these type theories, comparing two types must be done modulo β-equivalence. This is typically captured by the following conversion rule:

$$\frac{\Gamma \vdash a : \tau \qquad \tau \equiv \tau' : s}{\Gamma \vdash a : \tau'}$$

It is therefore the case that type checking (or equivalently proof checking) in such systems incurs the need to carry out arbitrary β-reductions. Efficient (full) normalization is particularly important when checking types entails a large amount of computation, as can often be the case, notably in proofs by reflection. Grégoire and Mahboubi (2005) and Gonthier (2007) provide ideal examples of such proofs. Other heavy users of normalization include partial evaluation, since specializing a function to statically known arguments amounts to fully normalizing this partially applied function.

Of late, functional languages have seen their influence considerably increase and their scope of application in the industry and in academia reach previously unforeseen niches. An enabling ingredient to this success has been the availability of efficient evaluation mechanisms for programs written in these languages, contending even with lower level imperative languages for the performance crown. A particularly elegant idea, normalization by evaluation (NbE), proposes to exploit off-the-shelf evaluators to implement normalization, rather than rolling out a custom built normalizer from scratch (Altenkirch et al.; Berger et al. 1998, 2003; Coquand 1994; Coquand and Dybjer 1997; Danvy 1996; Filinski and Rohde 2004). All the better for speedy execution on stock hardware: some evaluators for functional languages have benefited from dozens of man years spent pouring over complex optimizations and tweaking the execution paths on a multitude of computer architectures.

Unfortunately, all flavors of NbE proposed so far have, to the best of our knowledge, achieved one or the other of the following two goals, but never both:

1. generalize to well typed terms in arbitrarily complex type systems.
2. Avoid making the cost of each reduction significantly higher than that of the underlying evaluator.

Starting from a normalizing interpreter for the λ-calculus with constants, we iteratively improve the performance of the evaluator through equational reasoning and the introduction of higher order abstract syntax (HOAS), ultimately deriving a form of normalization by evaluation. In contrast to usual approaches to NbE, where the normalization is type driven, and along the same lines as Aehlig et al. (2008) and Filinski and Rohde (2004), we shunt the first problem by deriving an *untyped* variant of NbE that finds the normal form of all λ-terms if there is one (Section 2). We then show how to improve on this naive implementation to the point where the time cost of β-reduction is typically within a few percentage points of that of the underlying evaluator. We demonstrate this using a few benchmarks whose results we discuss in Section 4.

Our main contribution is an efficient yet lightweight method for normalizing arbitrary λ-terms, further reaffirming that beyond the theoretical interest in NbE, it is also a realistic execution technique whose performance is on par with the best (albeit weak) reduction devices available.

2. Untyped NbE

2.1 The framework

Consider normalization of the pure λ-calculus with constants. By iteratively and exhaustively applying the β-rule one can of course find the normal form of some arbitrary term. This is a directed notion of normalization. But an alternative view of normalization is to consider normalization as a term equivalence relation. Then, the normal form of a term is just a representative of the equational theory formed by the reflexive, transitive and symmetric closure of the β-reduction relation. A normalization function finds the normal

$$
\begin{array}{llcll}
 & \mathsf{Var} & \ni & x, y, z & \\
 & \mathsf{Term} & \ni & t & ::= x \mid \lambda.\ t \mid t\ t \\
\mathsf{Term} \supset & \mathsf{Term}_{\mathsf{N}} & \ni & t_e & ::= x \mid t_e\ t \\
\mathsf{Term} \supset & \mathsf{Term}_{\mathsf{NF}} & \ni & t_n & ::= t_a \mid \lambda.\ t_n \\
\mathsf{Term} \supset & \mathsf{Term}_{\mathsf{A}} & \ni & t_a & ::= x \mid t_a\ t_n
\end{array}
$$

Figure 1. Grammar and subgrammars of terms. Variables are encoded using de Bruijn levels.

form t' of a term t with t and t' equivalent. This is a reduction-free view of normalization (Filinski and Rohde 2004).

The normalization function does not have to be β-reduction based. Suppose we can construct a denotational model of the λ-calculus with the following two properties:

1. if $t_1 \leftrightarrow_{\beta\eta} t_2$ then $[\![t_1]\!] = [\![t_2]\!]$ (soundness);
2. if t_1 is in normal form then a term t_2 can be extracted from a denotation $[\![t_1]\!]$, such that $t_1 \leftrightarrow_{\alpha} t_2$ (reproduction).

Then a normalization function taking as input a closed term t can be given as

$$\Downarrow t \ = \downarrow ([\![t]\!]\ \emptyset),$$

where $\downarrow$ is the extraction function, which we will call *reification*. For any t_1 in normal form, by soundness of the model $\downarrow ([\![t_1]\!]\ \emptyset) = \downarrow ([\![t_2]\!]\ \emptyset)$ for all t_2 such that $t_1 \leftrightarrow_{\beta\eta} t_2$. Since by reproduction $\downarrow ([\![t_1]\!]\ \emptyset) \leftrightarrow_{\alpha} t_1$, we have $\Downarrow t_1 \leftrightarrow_{\beta\eta} t_2$ as expected.

2.2 Towards reduction-free normalization

Consider the following representation of the syntax[1] using de Bruijn levels. The grammar for the syntax is given by the Term production in Figure 1.

$$\mathbf{data}\ \mathit{Term} = \mathit{Var}\ \mathit{Int} \mid \mathit{App}\ \mathit{Term}\ \mathit{Term} \mid \mathit{Abs}\ \mathit{Term}$$

A normal order normalization is usually implemented along the lines of[2]

$$
\begin{array}{l}
\mathit{norm}_1 :: \mathit{Term} \to \mathit{Term} \\
\mathit{norm}_1\ (\mathit{App}\ (\mathit{Abs}\ x\ t_1)\ t_2) = \mathit{norm}_1\ (\mathit{subst}\ x\ t_2\ t_1) \\
\mathit{norm}_1\ (\mathit{App}\ t_1\ t_2) = \\
\quad \mathbf{case}\ \mathit{norm}_1\ t_1\ \mathbf{of} \\
\qquad \mathit{Abs}\ x\ t_1' \to \mathit{norm}_1\ (\mathit{App}\ (\mathit{Abs}\ x\ t_1')\ t_2) \\
\qquad t_1' \to \mathit{App}\ t_1'\ (\mathit{norm}_1\ t_2) \\
\mathit{norm}_1\ (\mathit{Abs}\ x\ t) = \mathit{Abs}\ x\ (\mathit{norm}_1\ t) \\
\mathit{norm}_1\ t = t
\end{array}
$$

We can aim for a much simpler implementation by using higher order abstract syntax (HOAS) (Pfenning and Elliot 1988), whereby binders of the term language are represented as functions in the metalanguage. This allows us to dispense with managing scopes, variables and capture avoiding substitutions ourselves. That work is offloaded to a contraption capable of doing it far more efficiently and correctly than we are: the metalanguage runtime. Moving to HOAS requires a few tweaks on the Term datatype:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Term} = \mathit{Const}\ \mathit{String} \mid \mathit{Abs}\ (\mathit{Term} \to \mathit{Term}) \\
\quad \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{array}
$$

Syntax variables are represented by metalanguage variables. We can therefore dispense with the Var constructor and introduce in its place the Const constructor, which stands in lieu of uninterpreted constants — or equivalently, free variables. For example, the term using named variables $\underline{(\lambda x.\ (\lambda y.\ y\ x))\ z}$ parses to the expression

$$\mathit{App}\ (\mathit{Abs}\ (\lambda x \to \mathit{Abs}\ (\lambda y \to \mathit{App}\ y\ x)))\ (\mathit{Const}\ \texttt{"0"})$$

The datatype Term represents the universe of all λ-terms, normalization of which is achieved by the following code:

$$
\begin{array}{l}
\mathit{norm}_2\ n\ (\mathit{App}\ (\mathit{Abs}\ t_1)\ t_2) = \mathit{norm}_2\ n\ (t_1\ t_2) \\
\mathit{norm}_2\ n\ (\mathit{App}\ t_1\ t_2) = \\
\quad \mathbf{case}\ \mathit{norm}_2\ n\ t_1\ \mathbf{of} \\
\qquad \mathit{Abs}\ t_1' \to \mathit{norm}_2\ n\ (t_1'\ t_2) \\
\qquad t_1' \to t_1'\ \underline{@}\ (\mathit{norm}_2\ n\ t_2) \\
\mathit{norm}_2\ n\ (\mathit{Abs}\ t) = \\
\quad \underline{\lambda.}\ (\mathit{norm}_2\ (n+1)\ (t\ (\mathit{Const}\ (\mathit{show}\ n)))) \\
\mathit{norm}_2\ n\ (\mathit{Const}\ c) = \underline{c}
\end{array}
$$

One can see here how the problem with shifting bindings to the metalanguage is that we can no longer descend under abstractions; they have become black boxes. But descending under abstractions is needed to normalize, so let us deconstruct these abstractions, thus turning the variable bound by some abstraction free. Remember that we already have a way to represent free variables, using Const. So normalizing an abstraction simply requires applying the abstraction to a fresh[3] (unbound) variable and normalizing the result.

After deconstructing and normalizing under the abstraction comes the time to reconstruct this abstraction. Rather than reconstructing an opaque metalanguage term, we can simply reify the abstraction into a term of the syntax. Our normalization function is no longer an endomorphism on Term: its result is a syntactic term in normal form.

The next step is to split out of norm_2 the code dealing with applications into an *app* function. By appeal to the semantics of the metalanguage, we can offload yet more work to the metalanguage runtime. Insofar as evaluation order of the normalizer and metalanguage correspond, all App nodes can be removed from terms and replaced with calls to the *app* function. The App constructor is still needed, but only to represent neutral terms (i.e. $\mathsf{Term}_{\mathsf{N}}$ of Figure 1). The previous example then becomes

$$\mathit{app}\ (\mathit{Abs}\ (\lambda x \to \mathit{Abs}\ (\lambda y \to \mathit{app}\ y\ x)))\ (\mathit{Const}\ \texttt{"0"})$$

This leads to the final definition of our normalizer:

$$
\begin{array}{l}
\mathit{app}\ (\mathit{Abs}\ t_1)\ t_2 = t_1\ t_2 \\
\mathit{app}\ t_1\ t_2 = \mathit{App}\ t_1\ t_2 \\
\mathit{norm}\ n\ (\mathit{App}\ t_1\ t_2) = (\mathit{norm}\ n\ t_1)\ \underline{@}\ (\mathit{norm}\ n\ t_2) \\
\mathit{norm}\ n\ (\mathit{Abs}\ t) = \underline{\lambda.}\ (\mathit{norm}\ (n+1)\ (t\ (\mathit{Const}\ (\mathit{show}\ n)))) \\
\mathit{norm}\ n\ (\mathit{Const}\ c) = \underline{c}
\end{array}
$$

After this final step, notice that all forms in the syntax are now interpreted directly with their corresponding (tagged) forms in the metalanguage, as shown in Figure 2. *norm* matches the specification of a reification function. Indeed, parsing a term to the metalanguage, then unparsing the resulting construct with *norm*, is an untyped, reduction-free, normalization by evaluation function, in the sense of Section 2.

A full implementation of the normalizer is detailed in Appendix **??**. The metalanguage type system ensures that the terms produced are in normal form, if any.

3. Optimizations

In this section we will focus on offloading yet more work to the metalanguage runtime by exploiting intrinsic features of most

[1] For notational clarity, we will underline in what follows the syntax of terms and denote the implementation language (or *metalanguage*) using the more convenient Haskell syntax.

[2] The definition of subst is elided for conciseness.

[3] In practise one can opt for one of a variety of strategies for freshness. For simplicity, in this paper we get away with a simple integer counter by using de Bruijn levels in the term syntax.

$$\llbracket \underline{x} \rrbracket\ n = \hat{x} \qquad \text{if } x < n$$
$$\llbracket \underline{x} \rrbracket\ n = Const\ \underline{x} \qquad \text{otherwise}$$
$$\llbracket \underline{\lambda.\ t} \rrbracket\ n = Abs\ (\lambda \hat{n} \to \llbracket t \rrbracket\ (n+1))$$
$$\llbracket \underline{t_1\ t_2} \rrbracket\ n = app\ (\llbracket t_1 \rrbracket\ n)\ (\llbracket t_2 \rrbracket\ n)$$

Figure 2. Translation of the syntax into the metalanguage. $\hat{\cdot}$ maps naturals to variable names.

higher order programming languages that go beyond the pure λ-calculus. One such feature is the uncurrying of function applications, the other is pattern matching on algebraic datatypes.

3.1 Minimizing closures

Functional values in functional programming languages are typically represented as *closures*, a pairing of code and an environment assigning values to all free variables appearing in the code. Consider a church encoding of lists and a right fold in a syntax where functions can be applied to multiple arguments in one go.

$$nil \equiv \underline{\lambda fg.\ f}$$
$$cons \equiv \underline{\lambda htfg.\ g\ h\ (t\ f\ g)}$$
$$map \equiv \underline{\lambda fl.\ l\ nil\ (\lambda ht.\ cons\ (f\ h)\ t)}$$

Y is the usual call-by-name fixed-point combinator. The notation $\lambda x_1 \ldots x_n.\ []$ is syntactic sugar for $(\lambda x_1.\ \ldots (\lambda x_n.\ []) \ldots)$. That is, the higher-order functions above take multiple arguments, but are encoded in terms of unary functions that return functions. This encoding is called *currying*.

Note however that currying has a cost. Applying a function to multiple arguments entails the creation of many short-lived intermediate closures, one for each function returned as a result of the application to one argument. In general, one will need to allocate (and then deallocate soon thereafter) $n-1$ closures during the consecutive application of a function to n arguments. For instance,

$$\begin{aligned}&\llbracket map\ id\ nil \rrbracket\\ &\quad = app\ (app\ map\ id)\ nil\\ &\quad = app\ (app\ (Abs\ (\lambda f \to Abs\ (\lambda l \to ...)))\ id)\ nil\\ &\quad \to_\beta\ app\ (Abs\ (\lambda l \to ...))\ nil\\ &\quad \to_\beta\ nil\end{aligned}$$

Here, map is applied to two arguments, therefore one intermediate Abs structure is constructed. But an alternative encoding of n-ary functions could avoid this.

The literature abounds with various encodings of n-ary functions (i.e. calling conventions) targeted by compilers to avoid costly closure allocation. Simon Peyton Jones Peyton-Jones (1992) proposes the Push/Enter and Eval/Apply dichotomy to describe them. We pick the Eval/Apply model here for its very cheap implementation cost and good performance in the common case (Marlow and Peyton-Jones 2006). That is, assuming a syntax where consecutive λ's have been folded into multiple argument abstractions, we can forgo many Abs constructions by means of a family ap_n of application operators and the addition of a number of Abs_n constructors, as shown in Figure 3. Note that most functions appearing in terms of the syntax will typically have low arity, so that one could reap most of the benefit of this approach even if bounding the number of ap_n operators and Abs_n constructors to a small number such as 4 or 5. Though uncommon, applications of functions with higher arity is still possible, but at a slight performance cost due to extra closure construction.

Parsing the above terms to the metalanguage now gives:

1. $ap_n\ (Abs_m\ f)\ t_1\ \ldots\ t_n = Abs_{m-n}\ (f\ t_1\ \ldots\ t_n)$
2. $ap_n\ (Abs_m\ f)\ t_1\ \ldots\ t_n = f\ t_1\ \ldots\ t_n$
3. $ap_n\ (Abs_m\ f)\ t_1\ \ldots\ t_n = ap_{n-m}\ (f\ t_1\ \ldots\ t_m)\ t_{m+1}\ \ldots\ t_n$

where conditions on (1) are if $n < m$, on (2) if $n = m$, on (3) if $n > m$.

Figure 3. A family of ap operators

$$\begin{aligned}&nil = Abs_2\ (\lambda f\ g \to f)\\ &cons = Abs_4\ (\lambda h\ t\ f\ g \to ap_2\ g\ h\ (ap_2\ t\ f\ g))\\ &map = Abs_2\ (\lambda f\ l \to\\ &\quad ap_2\ l\ nil\ (Abs_2\ (\lambda h\ t \to ap_2\ cons\ (ap_1\ f\ h)\ t)))\end{aligned}$$

For small n, n-ary functions in the syntax are encoded using n-ary functions in the metalanguage. Beyond economizing data structure allocations, this optimization permits us to reap the benefits of closure allocation strategies typically found in compilers to reduce the cost and frequency of extending closure environments. For example, many execution environments such as the OCaml interpreter can avoid any allocation of environments on the heap in the common case of n-ary functions applied to n arguments, instead pushing all arguments on the stack (Leroy 1990; Peyton-Jones 1987).

3.2 Specialized constructors

Representing all datatypes as functions via Church encodings induces needlessly many β-reductions and wastes opportunities for optimization. Haskell and many other statically typed functional programming languages feature algebraic datatypes and pattern matching facilities on these datatypes, enabling more natural and more efficient data manipulation. Compiling complex pattern matches to decision trees (Maranget 2008) or to backtracking automata (Le Fessant and Maranget 2001) can drastically reduce the amount of computation needed to access and manipulate algebraic structures.

With the current definition of $Term$, it is already possible to parse patterns in the syntax to case analysis constructs in the metalanguage, but currently a metalanguage representation of a pattern p_1 can become quite a bit larger than p_1. Assume for instance constants nil and $cons$, constructors of the list type, and take the definition of $append$ in the metalanguage:

$$\begin{aligned}&append = Abs_2\ (\lambda xs\ ys \to \mathbf{case}\ xs\ \mathbf{of}\\ &\quad Const\ \texttt{"nil"} \to ys\\ &\quad App\ (App\ (Const\ \texttt{"cons"})\ x)\ xs' \to\\ &\qquad ap_2\ (Const\ \texttt{"cons"})\ x\ (ap_2\ append\ xs'\ ys)\end{aligned}$$

Replacing the constructor names with integers rather than strings to avoid string comparison cost does spare some computation, but it is better to avoid the $Const$ constructor altogether. Rather than representing a datatype as an in-memory tree, with App constructors at branch nodes and $Const$ constructors at the leaves, each in its own memory cell, it is much more memory efficient to add all data constructors found in the syntax as additional constructors to the metalanguage interpretation, effectively flattening the representation in memory. That is, for constructors nil and $cons$, add

$$\mathbf{data}\ Term = ...\ |\ Nil\ |\ Cons\ Term\ Term$$

As shall be detailed in Section 4, a flatter structure means less indirection when performing pattern matches, hence better performance.

The downside of mirroring syntax level constructors as constructors in $Term$ is that doing so breaks modularity. Since the $Term$ datatype is the universe of all syntax terms, breaking up definitions in the syntax into modules requires that all constructors in all modules need to be coalesced into the term $Term$ datatype. Encoding modules in the syntax with modules in the metalanguage is useless, because introducing a new constructor means modifying $Term$, which in turn means recompiling all modules because they all depend on $Term$.

A solution to recover modularity is to hardcode a set of constructors in the $Term$ datatype, much as we hardcoded the set Abs_n of n-ary functions. This means that constructors with small arity in the source language can be represented using a single constructor in the metalanguage. Larger (less common) constructors in the source language can of course be represented as the composition of smaller constructors.

$$\mathbf{data}\ Term = ... \mid Const\ Int \mid Const\ Int\ Term$$
$$\mid ... \mid Const\ Int\ Term\ ...\ Term$$

In languages that feature first class arrays, in particular allowing pattern matching on arrays (such as OCaml), one could also replace the definition of $Const$ with

```
type term = ... | Const of name * term array
```

The effect of removing $Const$ is to build in a closed world assumption on constructors of the syntax. Some languages allow the definition of extensible datatypes, which we can use to break the closed world assumption. Recent versions of OCaml feature polymorphic variants and Standard ML's exn exception datatype is extensible. Terms applied to a constant would simply be accumulated in the array. The array size is known in advance because all constructors have a fixed number of fields.

In summary, the appropriate option will be contingent on the runtime environment chosen to execute the normalizer. As always, the objective here is to make do with existing runtime environments without modification, whilst observing that the penalty of this constraint can be made close to negligible — an observation substantiated in the following section.

4. Benchmarks

Our use of untyped NbE is as a cheap contraption to efficiently perform the conversion test in dependent type theories. In this section we examine the effect of various optimizations presented previously on a small set of benchmarks and compare them to earlier work on untyped NbE by Aehlig et al. (2008). In these benchmarks, the object language is Haskell. The interpretation stage of NbE then becomes a source-to-source transformation on programs, which we implement using Template Haskell. The transformed source is then compiled to native code by the GHC compiler.

We compare 6 flavors of NbE:

ahn This is untyped NbE as described in Aehlig et al. (2008). All functions are interpreted as unary functions. All function arguments are packed into lists that the function pattern matches over to extract individual arguments.

singlearity This interpretation takes every function to a unary closure. Functions taking multiple arguments are curried and are represented using multiple embedded closures.

evalapply The optimization described in 3.1.

constructors Every constructor appearing in terms of the object language become additional constructors $Term$, as in 3.2.

ucea Combination of "evalapply" and "constructors".

whnf The identify interpretation, where terms of the object language are interpreted as themselves.

We run the following benchmarks for each of the flavors:

append Concatenation of two large lists of integers of size 50,000.

even Test whether an input list is even or odd. Lists are represented using a Church encoding, so that no pattern matching occurs in this benchmark. It is meant to test performance of applications.

sort Sorting of large lists of integers encoded using constructors. This benchmark is meant to be rather more sensitive to pattern matching performance. The implementation is mergesort found in the base package of the Haskell libraries.

exp3-8 A tiny benchmark appearing in the nofib suite: taking 3 to the power of 8, in Peano arithmetic.

queens Enumerate the solutions to this classic constraint satisfaction problem: find a way to place 10 queens on a 10x10 chess board such that no two queens are on the same column or row.

The results are shown in Figure 4 and Table 1. Note immediately how the vast majority of the performance benefits comes from interpreting constructors as constructors; this greatly reduces the size of the patterns to match and help allocate fewer objects on the heap. An overview of the heap usage and garbage collection on each of the above benchmarks shows that using constructors typically halves total heap allocation during the lifetime of the program and eases the pressure on the garbage collector somewhat.

Currying functions, rather than grouping the arguments into lists that are frequently deconstructed and reconstructed, affords a gain in most benchmarks. The eval/apply optimization allows a further halving of execution time on benchmarks with functions with high arity, such as queens and its heavy use of $foldr$.

The main observation, however, is that untyped normalization by evaluation with the addition of the eval/apply optimization and the use of metalanguage constructors is hardly any slower on these benchmarks than the execution of these benchmarks by evaluation alone. In pathological cases where none of the execution time is spent in pattern matching, such as the "even" benchmark, we observe a penalty of about 20%. However, pattern matching or garbage collection and heap allocation dominates the runtime of many functional programs. In such cases the extra cost of tagging closures is often negligible.

5. Related Work

Our work is a continuation of many other contributions regarding normalization by evaluation and its applications. Whilst many treatments of NbE do discuss computational efficiency, few quantify empirically performance on select benchmarks. Aehlig et al. (2008) is one work on which we build upon, being closely related both in its attention to the performance side of the coin and in the essence of their scheme. They too map terms of the object language to tagged equivalents in the metalanguage by embedding functions, free variables and constants into a datatype. Our approach differs from theirs in that we treat functions of arbitrary arity uniformly by currying. In their approach functions of the object language are mapped to single arity functions within the metalanguage, encapsulating all arguments of the functions inside lists. The body of the functions then pattern match on the input list to extract arguments. Whilst appealing in its simplicity, their approach suffers performance-wise from allocating many lists during function application time that are then immediately deconstructed. In addition, encapsulating arguments inside lists breaks the optimization described in Section 3.1. For simplicity, constructors in the object language are not translated to constructors in the metalanguage but rather represented with a special constructor for constants. Lindley (2005) also considers untyped normalization by evaluation in a performance sensitive context, giving a quantitative analysis of the performance of a number of algorithms and variants compared to reduction based approaches. Optimizations for higher order programs and data constructors are not considered, however.

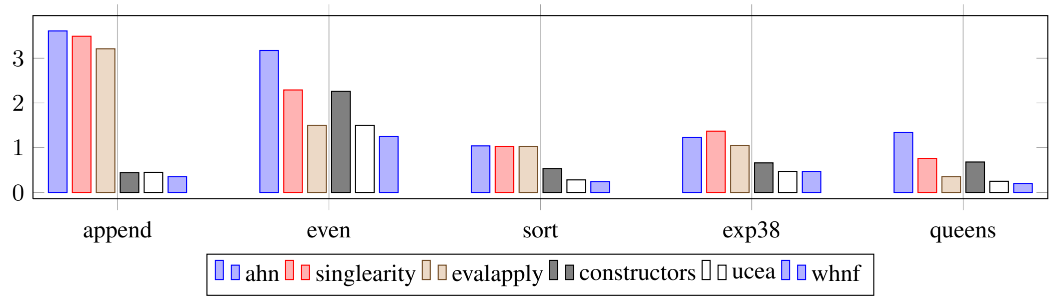

Figure 4. Visual representation of the data in Table 1.

flavor	append	%	even	%	sort	%	exp3-8	%	queens	%
ahn	3.61	1031	3.17	253	1.04	433	1.23	261	1.34	670
evalapply	3.21	917	1.50	120	1.03	429	1.05	223	0.35	175
singlearity	3.49	997	2.29	183	1.03	429	1.37	191	0.76	380
constructors	0.44	125	2.26	180	0.53	220	0.66	140	0.68	340
ucea	0.45	128	1.50	120	0.28	116	0.47	100	0.25	120
whnf	0.35	100	1.25	100	0.24	100	0.47	100	0.20	100

Table 1. Absolute execution times (seconds) and relative to execution time of whnf.

Filinski and Rohde (2004) propose a similar algorithm for untyped normalization by evaluation. Whilst Aehlig et al. (2008) prove only partial correctness, namely that if their algorithm returns a term then that term is in normal form and convertible to the input (*soundness* and *standardization* properties), Filinski and Rohde (2004) further prove *completeness*. However, the focus there is on a precise semantic study, rather than an evaluation of performance.

Of particular note in the work of Aehlig et al. (2008) is their generalization of NbE to the symbolic normalization of terms with regards to arbitrary user-provided rewrite rules. For conciseness, we do not discuss this matter further in this paper, but their translation of rewrite rules as pattern matching functions in the metalanguage can readily be adapted to the normalization scheme presented here. This generalization is not required for the conversion test in the Calculus of Inductive Constructions used by Coq for instance, but it is useful for reduction in Isabelle/HOL and for the conversion test in formalisms such as $\lambda\Pi$-modulo (Cousineau and Dowek 2007). Blanqui et al. (2007) independently propose a similar translation of rewrite rules into OCaml though in the context of finding canonical forms for non-free algebraic datatypes rather than applied to normalization.

A variety of virtual machines have been proposed for normalization. Notably, Crégut (2007) proves correct a normalizer for the λ-calculus. The code can be executed by expansion to Motorola 68000 assembly code, resulting in an efficient but more heavyweight (in the sense of implementation effort) and less portable execution model compared to NbE based approaches. The machine of Grégoire and Leroy (2002) which Coq sometimes uses for the conversion test should also be mentioned here. Theirs is a modified and formalized version of a bytecode interpreter for OCaml (the ZAM), to do normalization via reduction to weak head normal form along with a *readback* phase to restart weak reduction under binders. Whilst offering striking similarities to NbE, including in its reuse of existing evaluators, one important difference lies in the fact that the implementation of the underlying evaluator needs to be modified, whereas the objective of NbE, here and elsewhere, is to get away without looking under the hood. As a side effect, NbE affords more freedom of choice regarding which evaluator to choose, allowing for instance to trade off minimizing the trusted base for better performance.

The principal extension made to the ZAM to normalize Coq terms is the introduction of *accumulators*, which represent applications of free variables to a number of terms. Embedding this construct within the virtual machine avoids having to do case analysis at every application to discriminate between function applications and applications of neutral terms. We show that with the simple optimization of Section 3.1, the overhead of this case analysis is very small in practise.

These approaches can be seen as complementary to the one exposed here in that these normalizers are abstract machines whose correctness is more readily established, hence avoiding extending the trusted base of a theorem prover with code as large as that of a full scale compiler and the associated runtime environment for the chosen metalanguage. They may also reduce the cost of compilation, which for small terms can far exceed the time needed to normalize them.

6. Conclusion

Just as moving from a naive interpreter to an optimizing compiler can mean moving from the intractable to the feasible for the evaluation of programs, so too does compiling the costly components of the type checking problem in dependent type theories may reap enormous benefits. Others have shown how it is possible to bring to bear the power of existing compiler technology in proof assistants with little implementation effort. We have shown that to get excellent performance rivalling that of stock runtime systems for popular programming languages, the implementation effort is nearly trivial: parse the object language and pretty print it to tagged terms in the form of a functional program. We can have our cake and eat it too.

A limitation of normalization by evaluation is that terms are always evaluated to weak head normal forms before normalizing under binders. When strongly normalizing a term, this may not be the best strategy: in fact (Lévy 1978) has shown that this could lead to redundant copying of exponentially many λ-terms, which an optimal strategy might avoid. But seeking the optimal strategy may introduce far too much overhead to be viable in practice. As in Grégoire and Leroy (2002), the approach presented here seeks to minimize the cost of each reduction, at some expense on the total number of reductions performed. It would be interesting however, to allow for short-circuiting of normalization when reduction so far has yielded enough information to decide the convertibility of two terms, whilst retaining the conceptual and implementation simplicity of normalization by evaluation.

The normalization algorithm presented here has been implemented in a prototype proof checker for the $\lambda\Pi$-calculus modulo called *Europa*, but transferring this technology to full-fledged proof assistants would be of benefit. Further, effort is underway to establish full correctness. One possibility consists in demonstrating a computational adequacy result between the algorithm presented here and that studied in (Filinski and Rohde 2004), thus transferring the soundness, standardization and completeness results.

Acknowledgments

The author would like to thank Gilles Dowek for helpful comments on prior versions of this paper. Many thanks to Klaus Aehlig, Olivier Danvy, Benjamin Grégoire and Hugo Herbelin for fruitful discussions on normalization by evaluation, and to the anonymous reviewers for their comments.

A. Full implementation of a normalizer

$normalize$ maps terms to normal forms. The NF datatype represents terms in normal form. This datatype cannot be made a subtype of $Term$ in Haskell so the inj function is needed to build an element of $Term$ from one of NF. The definition of app is as in Section 2.2.

$$\begin{array}{l}
\mathbf{data}\ Term = Const\ Int \\
\quad \mid Abs\ (Term \to Term) \\
\quad \mid App\ Term\ Term \\
\mathbf{data}\ NF = N\ Neutral \\
\quad \mid NAbs\ (NF \to NF) \\
\mathbf{data}\ Neutral = NConst\ Int \\
\quad \mid NApp\ Neutral\ NF \\
inj :: NF \to Term \\
inj\ (N\ (NConst\ x)) = Const\ x \\
inj\ (N\ (NApp\ x\ y)) = App\ (inj\ (N\ x))\ (inj\ y) \\
inj\ (NAbs\ f) = Abs\ (\lambda x \to inj\ (f\ (normalize\ x))) \\
normalize :: Term \to NF \\
normalize\ (Const\ x) = N\ (NConst\ x) \\
normalize\ (Abs\ f) = NAbs\ (\lambda x \to normalize\ (f\ (inj\ x))) \\
normalize\ (App\ t_1\ t_2) = \mathbf{case}\ normalize\ t_1\ \mathbf{of} \\
\quad N\ t \to N\ (NApp\ t\ (normalize\ t_2)) \\
app\ (Abs\ t1)\ t2 = t1\ t2 \\
app\ t1\ t2 = App\ t1\ t2
\end{array}$$

References

K. Aehlig, F. Haftmann, and T. Nipkow. A Compiled Implementation of Normalization by Evaluation. *Theorem Proving in Higher Order Logics (TPHOLs 2008), Lecture Notes in Computer Science. Springer-Verlag*, 2008.

T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210.

U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. *Prospects for Hardware Foundations*, pages 117–137, 1998.

U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183(1):19–42, 2003.

F. Blanqui, T. Hardin, and P. Weis. On the Implementation of Construction Functions for Non-free Concrete Data Types. *Lecture Notes In Computer Science*, 4421:95, 2007.

C. Coquand. From Semantics to Rules: a Machine Assisted Analysis. *Lecture Notes In Computer Science*, pages 91–91, 1994.

T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(01): 75–94, 1997.

D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. *Lecture Notes in Computer Science*, 4583:102–117, 2007.

P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007.

O. Danvy. Type-directed partial evaluation. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–257, 1996.

A. Filinski and H.K. Rohde. A denotational account of untyped normalization by evaluation. 2004.

G. Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. ISBN 978-3-540-87826-1.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 235–246, 2002.

B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in coq. *Lecture notes in computer science*, 3603:98, 2005.

F. Le Fessant and L. Maranget. Optimizing pattern matching. *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, 2001.

X. Leroy. The ZINC experiment: an economical implementation of the ML language. 1990.

J.J. Lévy. *Réductions correctes et optimales dans le Lambda-Calcul*. Université Paris 7, 1978.

S. Lindley. Normalisation by evaluation in the compilation of typed functional programming languages, 2005.

L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46. ACM New York, NY, USA, 2008.

S. Marlow and S. Peyton-Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice/Hall International, 1987.

S. Peyton-Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5. *Journal of Functional Programming*, 2(2):127–202, 1992.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI '88*, pages 199–208, Atlanta, Georgia, United States, 1988. doi: 10.1145/53990.54010. URL `http://portal.acm.org/citation.cfm?doid=53990.54010`.