Maximal Laziness

An Efficient Interpretation Technique for Purely Functional DSLs

Eelco Dolstra

Utrecht University
eelco@cs.uu.nl

Abstract

In lazy functional languages, any variable is evaluated at most once. This paper proposes the notion of *maximal laziness*, in which syntactically equal terms are evaluated at most once: if two terms e_1 and e_2 arising during the evaluation of a program have the same abstract syntax representation, then only one will be evaluated, while the other will reuse the former's evaluation result. Maximal laziness can be implemented easily in interpreters for purely functional languages based on term rewriting systems that have the property of *maximal sharing* — if two terms are equal, they have the same address. It makes it easier to write interpreters, as techniques such as closure updating, which would otherwise be required for efficiency, are not needed. Instead, a straight-forward translation of call-by-name semantic rules yields a call-by-need interpreter, reducing the gap between the language specification and its implementation. Moreover, maximal laziness obviates the need for optimisations such as memoisation and let-floating.

1 Introduction

In *lazy* functional languages such as Haskell [18], the value of a variable binding is computed only when it is needed, and then only once. For instance, in the Haskell function

```
f x y = if x == 0 then y else z + z where z = product [1..x]
```

the function argument y is only computed when $x = 0$, and the local variable z only and only once when $x \neq 0$. Laziness is a useful property because it allows the programmer to abstract over the ordering of computations, and enables the construction of infinite data structures and the definition of control structures within the language [12].

This paper proposes the notion of *maximal laziness*, in which any set of "equal" terms is evaluated at most once during the execution of a program. For instance, in a function such as

```
f n = fac n + fac n where fac n = product [1..n]
```

under a maximal laziness regime, the expression `fac n` will be computed only once, while the second occurrence of the expression will reuse the result from the first. To be precise, if two terms e_1 and e_2 arising *during the evaluation of a program* have the same abstract syntax representation, then only one will be evaluated, while the other will reuse the former's evaluation result. Hence, it's a rather stronger property than static common subexpression elimination. For instance, in the program

```
f n = fac n + fac 10 where fac n = product [1..n]
```

if `f` is called with argument `10`, `fac 10` will be computed only once.

Maximal laziness is an expensive property to implement in a general purpose, compiled language. Indeed, in compiled code there is generally no notion of the abstract syntax tree of a value — certainly not one that relates in a meaningful way to the abstract syntax of the language. However, for domain specific languages (DSLs), one typically does not want to implement a full compiler but rather an *interpreter* that performs sufficiently well without too much implementation effort. As a motivating example of such a DSL, this paper uses the *Nix expression language* (described in Section 2), a purely functional language used by the Nix software deployment system [7,5] to specify how to build and compose software packages.

In interpreters for functional languages based on term rewriting, maximal laziness is much easier to achieve. In a term rewriting approach, the abstract syntax term representing the program is rewritten according to the semantic rules of the language until a normal form — the evaluation result — is reached. In fact, maximal laziness comes naturally when one implements the interpreter in a term rewriting system that has the property of *maximal sharing*, such as ASF+SDF [23] or the Stratego/XT program transformation system [24], both of which rely on the ATerm library [20] to implement maximal sharing of terms. In such systems, if two terms are syntactically equal, then they occupy the same location in memory — i.e., any term is stored only once (a technique known as *hash-consing* in Lisp). This makes it easy and cheap to add a simple memoisation to the term rewriting code to map abstract syntax trees to their normal forms, thus "caching" evaluation results and achieving maximal laziness.

Maximal laziness has a number of advantages:

- The implementation of the language becomes simpler and stays closer to the specification of the semantics of the language. The semantics of a purely functional language can be specified conveniently as a set of rewrite rules over the terms of the language, e.g., β-reduction to execute function calls: $(\lambda x.e_1)e_2 \mapsto e_1[x \rightsquigarrow e_2]$. However, direct implementation of such "call-by-name" semantics generally gives extremely poor performance, as one gets a lot of *work duplication*: for instance, in the β-reduction rule, the computation of e_2 will be duplicated for every occurrence of x in e_1. To prevent this, an entirely different, call-by-need style of implementation is required. We cannot simply substitute variables; rather, they must be represented explicitly and updated when they have been computed (see, e.g., [6] for an attempt to do so in a rewriting formalism). But with maximal laziness, the naive implementation of the semantics has the required "updating" behaviour. For instance, in the case of β-reduction, multiple occurrences of e_2 will be evaluated at most once due to the memoisation of evaluation results. This is shown in detail in Section 3 for the case of the interpreter for the Nix language.

- Maximal laziness can give a nice performance improvement over a "traditional" implementation of sharing (Section 5), reducing the number of rewrite steps by 40% to as much as 280% for typical, large Nix expressions. So maximal laziness gives faster but simpler interpreters. Also, several years of experience with maximal laziness in the Nix expression evaluator has shown that memory use scales well despite the naive evaluation result caching that never discards any result.
- Maximal laziness obviates the need for optimisations such as *let-floating* [19] (also known as the full laziness transformation) and function memoisation in many cases (Section 4). However, function memoisation is tricky in the case of non-strict languages.

2 Motivating example: Nix expressions

This section introduces a motivating example: a purely functional, domain specific language for which we want to implement an efficient interpreter without much effort. The DSL in question is the *Nix expression language*, which is used in the *Nix deployment system* [7,5] to specify how to build and compose software packages. A purely functional language is a good fit to the problem of specifying the building of software packages, because packages often need to be built in different versions or variants. For instance, different packages in the system may need to be built with different versions of the C compiler; and packages often have a great deal of variability in the functionality that can be compiled into them, such as whether to build Mozilla Firefox with support for Scalable Vector Graphics, which requires additional dependencies. Thus, it makes sense to describe packages as *functions* of their variability in terms of dependencies and optional features, so that such functions can be called any number of times with different arguments to create the desired instances of a package. The evaluation of a Nix expression yields a graph of build actions that must be performed to build a specific instance of a package with all its dependencies.

As an example, Figure 1 show a Nix expression that builds the GNU Hello package. First, it specifies at point $\boxed{1}$ a *function* named helloFun that builds the Hello package, given values that describe the dependencies required by that package, such as perl. Functions have the syntax *arg*: *body*. Thus, the body of helloFun is the call to the function stdenv.mkDerivation (at $\boxed{2}$).

The helper function stdenv.mkDerivation returns a special value called a *derivation*, which is simply the build graph for this particular instance of the Hello package. The arguments to mkDerivation specify the various inputs to the build, such as the package's name, its source code (src), and its dependencies (buildInputs). The source code is obtained by calling the function fetchurl, which specifies a derivation that downloads a file from the network. The evaluation of a Nix expression, at top-level, must yield one or more of these derivations, which are then used to perform the

```
helloFun =
  {stdenv, fetchurl, perl}: [1]

  stdenv.mkDerivation { [2]
    name = "hello-2.1.1";
    src = fetchurl {
      url = mirror://gnu/hello/hello-2.1.1.tar.gz;
      md5 = "70c9ccf9fac07f762c24f2df2290784d";
    };
    buildInputs = [perl];
  };

hello = helloFun { [3]
  inherit fetchurl stdenv perl;
};

stdenv = ...; perl = ...; fetchurl = ...;
```

Fig. 1. Nix expression for GNU Hello

required build actions imperatively. Thus, the declarative, purely functional Nix expression language is used to specify a set of imperative build steps.

Since helloFun is a function, to actually build an instance of the Hello package, we must call it. This is done at point $\boxed{3}$, and the resulting build graph is bound to the variable hello. The function is called with a set of arguments inherited from the surrounding lexical scope using the inherit keyword. (inherit x is merely syntactic sugar for an argument specification x = x;, i.e. the argument x is the expression x, where the latter x refers to the variable x in the surrounding scope.) Of course, the function can be called any number of times. For instance, if we had a value perl6 representing a different version of Perl, we could build Hello with it: helloWithPerl6 = helloFun {inherit fetchurl stdenv; perl = perl6;}. This is a lazy language: expressions are only evaluated, and the build graphs they represent only built, when they are actually needed.

It is not the purpose of this paper to give a full treatment of Nix or its expression language. (These can be found in [5] and in the Nix manual at `http://nix.cs.uu.nl/`.) Instead, the remainder of this section shows a part of the syntax and semantics of Nix expressions to illustrate, in Section 3, how such a semantics can be turned into an efficient interpreter using term rewriting and maximal laziness.

2.1 Syntax

The Nix expression language has several data types, such as strings, Booleans (with values true and false), lists (between square brackets), and attribute sets (between curly braces). The most important data type in the language is the *attribute set*, which is a set of name/value pairs, e.g., { x = "foo"; y = 123; }. Attribute names are identifiers, and attribute values are arbitrary expressions. The order of attributes is irrelevant, but any attribute name can occur only once in a set. Attributes can be selected using the . operator. E.g., { x = "foo"; y = 123; }.y evaluates to 123.

Recursive attribute sets allow attribute values to refer to each other. They are constructed using the rec keyword. Formally, each attribute in the set is added to the scope of the entire attribute set. Hence, rec { x = y; y = 123; }.x evaluates to 123. If rec were omitted, the identifier y in the definition of the attribute x would refer to some y bound in the surrounding scope. Recursive attribute sets introduce the possibility of recursion, including non-termination, e.g. `rec { x = x; }.x`. Recursion is used in the Nix language for many purposes, such as defining packages that are an input to themselves, e.g., the bootstrap process of compilers.

As we saw above, when defining an attribute set, attribute values can be *inherited* from the surrounding lexical scope or from other attribute sets. The expression `x: { inherit x; y = 123; }` defines a function that returns an attribute set with two attributes: `x` which is inherited from the function argument named `x`, and `y` which is declared normally. As the `inherit` construct is just syntactic sugar, the previous expression could also have been written as `x: { x = x; y = 123; }`. Note that the right-hand side of the attribute `x = x` refers to the function argument `x`, *not* to the attribute `x`. Thus, `x = x` is not a recursive definition.

The language has two types of functions. The first takes a single argument and has the form $x : e$. For instance, the (anonymous) identity function can be defined as `x: x`. Of course, this style of function is just a plain λ-abstraction from the λ-calculus. Though this style only allows functions with a single argument, since this is a functional language we can still define (in a sense) functions with multiple arguments, e.g., `x: y: x + y`, which is a function taking an argument `x` that returns *another function* that accepts an argument `y`.

The second style of function definition, and the one used in Figure 1, is more important in this language. It takes an attribute set and binds the attributes defined therein to local variables. Thus, `{x, y}: x + y` declares a function that accepts an attribute set with attributes `x` and `y` (and nothing else), and the expression `({x, y}: x + y) {y = "bar"; x = "foo";}` yields `"foobar"`.

2.2 Semantics

The operational semantics of the language is specified using semantic rules of the form $e_1 \mapsto e_2$ that transform expression e_1 into e_2. Rules may only be applied to closed terms, i.e., terms that have no free variables. Thus it is not allowed to arbitrarily apply rules to subterms.

An expression e_1 is said to *evaluate* to e_2, notation $e_1 \overset{*}{\mapsto} e_2$, if there exists a sequence of zero or more applications of semantic rules to e_1 that transform it into e_2 such that no rule is applicable to e_2. Thus e_2 is the *normal form* of e_1. Since rules are only allowed to be applied to an expression at top level (i.e., not to subexpressions), a normal form corresponds to the notion of a *weak head normal form* (WHNF) [16, Section 11.3.1]. Weak head normal form differs from the notion of *head normal form* in that right-hand sides of functions need not be normalised. A nice property of this style of evaluation is that there can be no *name capture* [3], which simplifies the evaluation machinery. Not all expressions have a normal form. For instance, the expression `(rec {x = x;}).x` does not terminate. But if evaluation does terminate, there must be a single normal form. This confluence property [2] follows from the fact that at most one rule applies to any expression.

The semantic rules are stated below in the following general form $\textsc{Rule} : \frac{condition}{e \mapsto e'}$. That is, we can conclude that e evaluates to e' if the proposition *condition* holds. If there are no conditions, the rule is simply written as $\textsc{Rule} : e \mapsto e'$.

As an example of a simple rewrite rule, consider conditionals, $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. Conditional expressions first evaluate the condition expression. It must evaluate to a Boolean. (Evaluation fails if it is not, but for simplicity I will not consider type errors here.) The conditional then evaluates to one of its alternatives.

$$\textsc{IfThen} : \frac{e_1 \overset{*}{\mapsto} \mathsf{true}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_2} \quad \textsc{IfElse} : \frac{e_1 \overset{*}{\mapsto} \mathsf{false}}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mapsto e_3}$$

The SELECT rule implements attribute selection. This rule governs *successful* selection, i.e., it applies only if the given attribute name exists in the attribute set.

$$\textsc{Select} : \frac{e \overset{*}{\mapsto} \{as\} \wedge \langle n = e' \rangle \in as}{e.n \mapsto e'}$$

Here as are the elements of an attribute set, and $\langle n = e \rangle \in as$ denotes that the attribute set as has an attribute named n with value e. Note that there is no rule for failure. If attribute n is not in as, evaluation fails and a nice error message is printed in the actual implementation.

For the remaining rules below, we need a notion of substitution of expressions for variables in other expressions. The substitution function $\mathsf{subst}(subs, e)$ (not shown here) performs a set of substitutions $subs$ in the expression e. The set $subs$ consists of substitutions of the form $x \rightsquigarrow e$ that replace a variable x with an expression e. subst replaces all *free variables* for which there is a substitution. A variable is free in a subexpression if it is not *bound* by any of its enclosing expressions. Variables are bound in functions and in recursive attribute sets. In recursive attribute sets, only the *recursive* attributes (as_1) bind variables; the non-recursive attributes (as_2) do not. It is assumed that the expressions in $subs$ contain no free variables, so subst does not have to perform renaming to prevent name capture.

A recursive attribute set is desugared to a normal attribute set by replacing all occurrences of references to the attributes with the recursive attribute set. For instance, if $e = \mathsf{rec}\ \{\mathsf{x = f\ x\ y;\ y = x;}\}$, then e is desugared to $\{\mathsf{x = f\ (e.x)\ (e.y);\ y = e.x;}\}$, or in full, $\{\mathsf{x = f\ ((rec\ \{x = f\ x\ y;\ y = x;\}).x)\ ((rec\ \{x = f\ x\ y;\ y = x;\}).y);\ y = (rec\ \{x = f\ x\ y;\ y = x;\}).x;}\}$. This desugaring is implemented by the REC rule:

$$\textsc{Rec} : \mathsf{rec}\ \{as\} \mapsto \{\mathsf{subst}(subs, \{as\})\}$$

where $subs = \{n \rightsquigarrow (\mathsf{rec}\ \{as\}).\mathsf{n} \mid n \in \mathsf{names}(as)\}$ and $\mathsf{names}(as)$ is the set of attribute names occurring in the left hand side of a set of attributes as. As we shall see in Section 3, due to maximal sharing, this substitution does not lead to a potential explosion in the size of expressions.

Function calls to single-argument functions (i.e., lambdas) are just plain β-reduction in the λ-calculus [3].

$$\beta\text{-}\textsc{Reduce} : \frac{e_1 \overset{*}{\mapsto} x\!:\ e_3}{e_1\ e_2 \mapsto \mathsf{subst}(\{x \rightsquigarrow e_2\}, e_3)}$$

(As argued in Section 3, expression e_2 can contain no free variables. Therefore, there is no danger of name capture in subst.) Calls to multi-argument functions,

```
Expr eval(Expr e)
{
  Expr e1, e2, e3;
  if (matchIf(e, e1, e2, e3) && evalBool(e1))
    return eval(e2);
  ATerm x;
  if (matchCall(e, e1, e2) &&
      matchFunction1(eval(e1), x, e3)) {
    ATermMap subs; subs.set(x, e2);
    return eval(subst(subs), e3);
  }
  ... more rules ...
}
```

Fig. 2. Implementation of some of the semantic rules in C++

```
Expr eval(Expr $e$) :
  if cache[$e$] $\neq \epsilon$ :
    return cache[$e$]
  else :
    $e' \leftarrow$ realEval($e$)
    cache[$e$] $\leftarrow e'$
    return $e'$
```

Fig. 3. Evaluation caching (pseudo-code)

i.e., functions that match an attribute set, are a bit more complicated:

$$\beta\text{-Reduce'} : \frac{e_1 \overset{*}{\mapsto} \{fs\}\text{: } e_3 \;\wedge\; e_2 \overset{*}{\mapsto} \{as\} \;\wedge\; \mathsf{names}(as) = fs}{e_1\ e_2 \mapsto \mathsf{subst}(\{n \rightsquigarrow e \mid \langle n = e\rangle \in as\}, e_3)}$$

(fs (for "formals") is the set of names of arguments of a multi-argument function.) Note that a multi-argument function call is strict in its argument—the attribute set—but not in the values of the attributes.

3 Implementation

Using term rewriting, it is straight-forward to turn the semantic rules from the previous section into a concrete interpreter for the language. However, without maximal laziness, such an interpreter would not perform well. This section shows how we can obtain an efficient interpreter from a straight-forward translation of the semantic rules using term rewriting and maximal laziness.

3.1 Evaluation through rewriting

A typical way to derive an interpreter from rewrite rules is to select some abstract syntax representation for terms, and then to translate the rewrite rules into whatever meta-language the interpreter is implemented in. The Nix expression evaluator uses *ATerms* (for Annotated Terms) [20] to represent terms. The ATerm library is a C library that allows the efficient creation and manipulation of term data structures in C. An ATerm t is a the application of an n-ary constructor to n subterms, denoted $C(t_1, \ldots, t_n)$; or a list of n terms $[t_1, \ldots t_n]$; or some terminal term such as an integer or string (actually a nullary constructor). For example, the expression (x: x) 123 is represented as the ATerm Call(Function1("x",Var("x")),Int(123)).

Both the ASF+SDF Meta-Environment [23] and Stratego/XT [24] program transformation systems use ATerms for representing abstract syntax trees, and can be used to manipulate them conveniently. For example, in Stratego/XT the IfThen rule could be implemented as

```
eval: If(e1, e2, e3) -> e2 where <eval> e1 => Bool(True)
```

which is an almost literal translation of that rule. However, the Nix expression evaluator is written in C++, thus the translation of the rules is a bit more verbose. Figure 2 shows the outline of the function eval that implements the Nix expression evaluator, with the code corresponding to the IfThen and β-Reduce rules. It takes a pointer e to the ATerm representing the term to be evaluated, and returns a pointer to the ATerm representing the resulting normal form. Helper functions such as matchIf are used to recognise and build ATerms. The elided helper function evalBool(e) calls eval(e) and returns true if the resulting term is Bool(True).

The evaluator in Figure 2 is extremely slow. This is a result of a lack of *sharing* in the evaluation of variables. For instance, the code for the β-Reduce rule simply replaces every occurrence of Var(x) in the body of the function with the term representing the argument value. Thus, if x occurs n times in the body of the function, it is possible for e_2 (the argument) to be evaluated n times. Indeed, if x is passed as an argument to other functions, it may be duplicated even further, quickly leading to an exponential running time.

The typical solution to this explosion is to arrange for sharing of variable evaluation. For instance, β-Reduce could be defined as follows:

$$\beta\text{-}\textsc{Reduce} : \frac{e_1 \overset{*}{\mapsto} x\!: \ e_3}{e_1 \ e_2 \mapsto \mathsf{let} \ x = e_2 \ \mathsf{in} \ e_3}$$

where we give let a special "destructive update" semantics so that the evaluation result of x is written back into the right-hand side of the let-binding. Of course, to give a semantics to let, we need to maintain some kind of environment, which makes the semantic rules rather more complicated [6]. At runtime, there is the same problem: in an interpreter, we need to keep an environment of the bound variables that are in scope (which is much more work than the simple call to subst in Figure 2), while in compiled code, x would be a pointer that points to a piece of memory containing code and environment pointers (the *closure* or *thunk* [17]), which after evaluation is overwritten with the actual result.

3.2 Maximal sharing with ATerms

A very nice property of the ATerm library, which will be critical in solving the performance problems described above, is its *maximal sharing*: if two terms are syntactically equal, then they occupy the same location in memory. This means that a shallow pointer equality test is sufficient to perform a deep syntactic equality test. Maximal sharing is implemented through a hash table. Whenever a new term is created through the ATerm API (using functions such as ATmakeAppl), the term to be created is looked up in the hash table. If the term already exists, the address of the term obtained from the hash table is returned. Otherwise, the term is allocated, initialised, added to the hash table, and returned. A garbage collector takes care of freeing terms that are no longer referenced.

Maximal sharing makes term creation slower, due to the hash table check. However, this is offset by the fact that memory use is reduced and the overhead of allocating a term that already exists is removed. More importantly, *testing for equality between terms is very cheap*, namely a pointer equality test. This makes the implementation of operations such as substitutions and memoisation very cheap. Empirical results on the efficiency of maximal sharing of ATerms are given in [20,21].

3.3 Maximal laziness

Maximal sharing is extremely useful in the implementation of a Nix expression interpreter since it allows easy *caching* of evaluation results, which speeds up expression evaluation by removing unnecessary evaluation of identical terms. The interpreter maintains a hash lookup table $\mathsf{cache} : \mathsf{ATerm} \to \mathsf{ATerm}$ that maps ATerms representing Nix expressions to their normal forms. Figure 3 shows pseudo-code for the caching evaluation function eval, which "wraps" the real eval function from Figure 2 (now renamed to $\mathsf{realEval}$) in a memoisation layer. It is assumed that $\mathsf{realEval}$ calls back into eval to evaluate subterms (i.e., every time a rule uses the relation $\overset{*}{\mapsto}$ in a condition). Thus we obtain the desired caching. The special value ϵ denotes that no mapping exists in the cache for the expression. Note that thanks to maximal sharing, the lookup $\mathsf{cache}[e]$ is very cheap: it is a lookup of a pointer in a hash table.

Since any syntactically equal term is now evaluated at most once, the interpreter in Figure 3 is maximally lazy. So does this solve the performance problem with the "naive" implementation of rules such as β-REDUCE? Intuitively this seems to be the case, because multiple occurrences of x will be replaced by the same argument term e_2, and due to the memoisation in Figure 3, repeated encounters of e_2 will reuse the normal form of e_2 computed on the first encounter.

However, there is a catch: what if later substitutions in subexpressions of the body of the function cause the copies of e_2 to change in different ways? Consider the function call (x: (rec {y = "foo"; z = x;}.z) + (rec {y = "bar"; z = x;}.z)) y, which would reduce to (rec {y = "foo"; z = y;}.z) + (rec {y = "bar"; z = y;}.z). Here, the two occurrences of the variable x in the original expression evaluate to different results ("foo" and "bar" respectively). This is of course the result of unhygienic substitution: the free variable y in the argument becomes bound after β-reduction. However, it is easy to see that this situation can never occur because all top-level terms are always closed. (This fact is proven in [5, Section 4.4] and follows from the observation that all rules produce closed terms when applied to closed terms.)

Thus, a straight-forward, substitution-based reduction scheme such as the naive implementation of β-REDUCE has *at least* as much sharing as a more difficult implementation based on closure updating. More importantly, this property comes at almost no additional cost, as Section 5 shows.

4 Optimisations

4.1 Optimising substitution

While the memoisation of term evaluation prevents unnecessary recomputation, there is still a problem with substitution-based semantic rules such as β-REDUCE. Consider the expression $(x : \ y : \ e_1) \ e_2 \ e_3$, where e_2 is a large expression. With normal substitution, we first replace all occurrences of x in e_1 with e_2. Then, we replace all occurrences of y in the resulting term with e_3. This substitution also descends into the e_2 replacements of x, even though those subterms are closed. Since e_2 is large, this is inefficient. A naive implementation of subst that recurses over the

```
Expr eval(Expr $e$) :
    if cache[$e$] $\neq \epsilon$ :
        if cache[$e$] = blackhole : Abort.
        return cache[$e$]
    else :
        cache[$e$] $\leftarrow$ blackhole
        $e'$ $\leftarrow$ realEval($e$)
        cache[$e$] $\leftarrow$ $e'$
        return $e'$
```

Fig. 4. Evaluation caching with blackholing

```
                    (rec {f = x: f x;}).f 10
(Rec)        ↦     {f = x: (rec {f = x: f x;}).f x;}.f 10
(Select)     ↦     (x: (rec {f = x: f x;}).f x) 10
(β-Reduce)   ↦     (rec {f = x: f x;}).f 10
```

Fig. 5. Detecting infinite recursion

structure of the term, may thus perform a lot of redundant work by substituting repeatedly in syntactically equal subterms. It is important to recognise that under maximal sharing, a term should be treated as a graph rather than a tree. Thus, one optimisation for subst is to memoise it (taking into account the fact that when substitutions are removed from the mapping subs in some of the recursive cases, a new memoisation table must be used for the recursive call).

There is, however, a much simpler and efficient solution that uses the fact that all substitution terms are closed. The optimisation is that we can *mark* replacement terms to indicate to the substitution function that it need not descend into such subterms. Since substitution terms are always closed, we can adapt substitution function subst as follows:

$$\mathsf{subst}(subs, x) = \begin{cases} \mathsf{closed}(e) & \text{if } (x \rightsquigarrow \mathsf{closed}(e)) \in subs \\ \mathsf{closed}(e) & \text{if } (x \rightsquigarrow e) \in subs \\ x & \text{otherwise} \end{cases}$$

That is, replacement terms e are placed inside a wrapper $\mathsf{closed}(e)$. (The first case merely prevents repeated wrapping in closed nodes, e.g., $\mathsf{closed}(\mathsf{closed}(e))$, which reduces the effectiveness of caching.) The wrapper denotes that e is a closed subterm under which no substitution is necessary, since it has no free variables. To actually make use of this optimisation, we also add a case to subst to stop at closed terms, namely $\mathsf{subst}(subs, \mathsf{closed}(e)) = \mathsf{closed}(e)$. Of course, during evaluation we must get rid of closed eventually. That's easily implemented through a rule Closed: $\mathsf{closed}(e) \mapsto e$, as a closed term is semantically equivalent to the term that it wraps. Since reduction only takes place at top-level, the closed wrapper is only discarded when the term actually needs to be evaluated.

4.2 Blackholing

Figure 4 shows a simple modification of the eval function in Figure 3 that, in addition to maximal laziness, implements a trick known as *blackholing* [17] that allows detection of certain simple kinds of infinite recursion. When we evaluate an expression e, we store in the cache a preliminary "fake" normal form blackhole. If, during the evaluation of e, we need to evaluate e *again*, the cache will contain blackhole as the normal form for e. Due to the determinism and purity of the language, this necessarily indicates an infinite loop, since if we start evaluating e again, we will eventually encounter it another time, and so on.

Note that blackholing as implemented here differs from conventional blackholing, which overwrites a value being evaluated with a black hole. This allows discovery of self-referential values, e.g., x = ... x ...;. But it does *not* detect infinite recursions like in the expression (rec {f = x: f x;}).f 10, since every recursive call to f creates a *new* value of x, and so blackholing will not catch the infinite recursion. In contrast, our blackholing *does* detect it, since it is keyed on maximally shared ATerms that represent *syntactically equal* expressions. Figure 5 shows the evaluation of this expression. Note that the final expression is equal to the first (which is blackholed at this time), and so an infinite recursion is signalled.

4.3 Optimisations

Because of maximal laziness, we get some optimisations that are conventionally applied to purely functional languages for free. For instance, the *full laziness transformation* [19] makes code more efficient by moving subexpressions outward as far as possible, e.g. let {f x = let {y = fac 100} in x + y} in f 1 + f 2, which computes fac 100 twice, can be transformed into let {y = fac 100; f x = x + y} in f 1 + f 2, which computes it only once in a conventional lazy implementation. With maximal laziness, this transformation is unnecessary: repeated occurrences of the same subexpression across multiple calls to a function will be computed only once.

Another, usually explicit optimisation in purely functional programs is to memoise specific functions [4]. Intuitively, one would expect that memoisation of the language evaluation function (Figure 3) also memoises functions in the language. This is not necessarily the case, however, in the presence of non-strict arguments. For instance, consider the Fibonacci function:

```
fib = n: if n == 0 then 0 else
         if n == 1 then 1 else fib (n-1) + fib (n-2);
```

Without memoisation, this function is very inefficient. But maximal laziness won't memoise it for us in a non-strict language. This is because the arguments won't be evaluated terms 1, 2, etc., but unevaluated expressions such as ((9-1)-1)-1. Some memoisation will occur, but not enough to make the function run in $O(n)$ time.

Note that the function fib *is* in fact strict in its argument; the conditional if n == 0 ... forces evaluation of the argument. But by the time we evaluate the argument, we are already in the evaluation of the function, and it's too late.

The Nix expression evaluator implements a technique called *function short-circuiting* that cuts off evaluation of a function when the normal form of its argument becomes known and the function has been called before with an argument with the same normal form. It does so by keeping track of which function calls are currently being evaluated. In eval, after realEval returns with a normal form e' for some expression e, we check if we are currently evaluating some function Call(f, e) and cache[Call(f, e')] $\neq \epsilon$. If so, we unwind the stack to the eval call for Call(f, e) (by throwing an exception), and return cache[Call(f, e')]. Also, memoised Calls in cache must be stored with their normalised argument. That is, when eval has computed that Call(f, e) $\overset{*}{\mapsto} e'$, and cache[e] $\neq \epsilon$ (i.e., the function has evaluated its argument), it sets cache[Call(f, cache[e])] to e'.

5 Evaluation

To see how the variants of maximal laziness perform compared to no sharing and to a conventional implementation based on closure updating, Table 1 shows the execution times of several variants of the Nix expression evaluator on a number of Nix expressions. The execution times are in seconds. The tests were performed on an Linux-based Athlon 64 X2 3800+ with 1 GiB of memory. Entries marked "-" in the table mean that the test did not finish in a reasonable amount of time because of the exponential explosion due to the lack of sharing or in substitutions. Table 2 shows the number of calls to eval for each of the Nix expressions in Table 1 for some of the variants, along with the number of cache hits (the number of times that a call to eval could be satisfied from the cache).

The tested Nix expressions are: *1)* The function fib (Section 4.3) with $n = 25$. *2)* A variant of fib in which the recursive call reads strict fib (n-1) + strict fib (n-2) where strict is a built-in function that reduces its second argument to normal form before applying the first to it. *3)* While fib is a toy problem for which the Nix expression DSL isn't even intended, the remaining tests are realistic. The third test computes the derivation graph of the gcc attribute in the Nix Packages collection, a set of Nix expressions for over a thousand Unix software packages. *4)* The evaluation performed by the Nix command nix-env -qa '*' –drv-path –out-path (which shows all packages defined by an expression) applied to the Nix Packages collection, which involves computing the derivation graphs of all packages, and causes the evaluation of 743 source files containing 22191 lines of code. *5)* The computation of the derivation graph for the installation CD of NixOS, a Linux distribution based on Nix, which involves the evaluation of 162 source files containing 13503 lines of code.

The variants of the Nix expression evaluator (available at https://svn.cs.uu.nl:12443/repos/trace/nix/branches/sharing-hackery/) are as follows.

- *No sharing*: the naive term-rewriting based interpreter from Section 2. Unsurprisingly, it performs very poorly.
- *Traditional sharing*: an implementation that updates variable bindings after they have been evaluated. This is the sharing model in most implementations of functional languages. This implementation is a modification of the existing (maximally lazy) Nix expression evaluator, made for comparison purposes. It therefore does use the ATerm library, and maximal sharing to store terms efficiently.
- *Maximal laziness*: the *no sharing* variant with memoisation as in Figure 3. Table 2 shows that the number of rewrite steps is substantially smaller than with traditional sharing, and the number of cache hits is substantial. (The number of rewrite steps is the same as for the *maximal laziness with both* variant in Table 2.) However, term blow-up due to substitutions causes it to perform poorly on large terms: it is outperformed by traditional sharing on the GCC test, and doesn't finish in a reasonable time frame on the nix-env and NixOS tests.
- *Maximal laziness with substitution memoisation* is the previous variant with memoisation around the subst function (see Section 4.1). It helps performance a bit,

	fib 25	fib 25 (strict)	GCC	nix-env	NixOS
No sharing	-	28.866	151.633	-	-
Traditional sharing	32.576	33.178	0.788	45.304	10.752
Maximal laziness	23.904	0.018	2.778	-	-
Maximal laziness + substitution memoisation	15.253	0.023	1.025	-	-
Maximal laziness + closed term optimisation	6.558	0.018	0.212	2.750	0.988
Maximal laziness + both	6.184	0.022	0.195	2.752	0.939
Maximal laziness + short-circuiting	0.022	0.022	0.197	2.820	1.181

Table 1
CPU times in seconds for sharing variants

	fib 25		fib 25 (strict)		GCC		nix-env -qa		NixOS	
	Steps	Hits	Steps	Hits	Steps	Hits	Steps	Hits	Steps	Hits
No sharing	-	n/a	6809K	n/a	4421K	n/a	-	n/a	-	n/a
Traditional sharing	5838K	n/a	6809K	n/a	6516	n/a	561K	n/a	975K	n/a
Maximal laziness + both	3820K	1850K	705	342	4538	2254	368K	225K	253K	111K
Maximal laziness + short-circuiting	675	292	682	319	4538	2254	367K	225K	253K	111K

Table 2
Rewrite steps and cache hits for sharing variants

but not enough to save it.

- *Maximal laziness with closed term optimisation* wraps substituted terms in closed nodes as described in Section 4.1. This very simple change alone makes maximal laziness fast enough: operations such as nix-env -qa now run in a few seconds, which is actually faster than similar operations in other package management tools that do not have package descriptions in a full-fledged programming language. We can thus conclude that the closed term optimisation is essential to make maximal laziness feasible. This is the variant that the production version of Nix uses.
- *Maximal laziness with both* combines substitution memoisation and closed term optimisation. It does not give an appreciable improvement over the latter.
- *Maximal laziness with short-circuiting* adds the short-circuiting technique described in Section 4.3. It does indeed succeed in turning the non-strict fib function automatically into a memoised function that runs in $O(n)$ time. However, it doesn't do much for Nix expressions in the real world.

So what is the cost in terms of memory use of maximal laziness? The nix-env -qa test, which represents the largest computation occuring in practice, takes around 21 MiB, a fairly modest amount of memory on current systems. On the other hand, the atypical non-strict fib 25 test on the interpreter with maximal laziness and the closed term optimisation (the one with 3820K reduction steps) takes around 170 MiB as a result of a lack of identical subterms (which short-circuiting solves).

The main lesson of this evaluation is that maximal laziness only works well with the closed term optimisation, which is fortunately trivial to implement. However, cache pruning becomes necessary to control memory consumption when evaluating programs with little sharing, which is not the case for the Nix DSL.

6 Related work

Maximal sharing, the technique upon which maximal laziness is implemented, goes back a long way. It is known as hash-consing in Lisp [1,8], where its utility is limited by the impurity of Lisp [10]. Type-safe hash-consing in OCaml that ensures that programmers cannot make unshared terms is discussed in [9], which uses operations on λ-terms (similar to the term rewriting in Section 2) as an example.

The Nix expression evaluator is built on top of the ATerm library [20,22], which is used in numerous term rewriting systems such as Stratego/XT [24] and in particular the ASF+SDF Meta-Environment [23], for which the ATerm library was originally developed. The evaluator would certainly have been easier to implement in Stratego than in C++, but this was not done as 1) C++ is a more suitable language for general systems programming, and 2) being a deployment tool, Nix should have as few dependencies as possible to ensure portability and ease of installation. The ASF+SDF compiler relies heavily on maximal sharing for performance; see [21] for a in-depth discussion. The compiler can be instructed to generate memoisation around explicitly specified ASF+SDF functions. The authors note that "memoization may easily become counterproductive if the memoized functions are not called with the same arguments sufficiently often, and finding the right subset of functions to memoize may require considerable experimentation and insight." In this paper, we have suggested memoising *everything*, which is not a feasible strategy for general purpose languages, but, as we have seen, may simplify the implementation of DSLs while providing sufficient performance. Unlimp [13] also appears to memoise arbitrary term evaluation in the context of a purely functional language, but does not discuss experience with non-trivial programs.

The "purity" of purely functional languages naturally suggests the use of maximal sharing, since, contrary to impure languages, one can unconditionally memoise functions to obtain optimised versions. This relationship is explored in depth in [11], which describes a language that has maximal sharing as a part of its runtime system to ensure that all data is maximally shared, and shows that maximal sharing is not an expensive feature (a fact also borne out by the experiences with ASF+SDF and Stratego/XT). It also discusses the sharing of *computations* (as opposed to data) through memoisation using maximal sharing — precisely the subject of this paper. However, as in [4], memoisation is not automatically applied to *all* computations.

There is a great deal of theoretical work on optimal reduction strategies for the λ-calculus (in particular Lamping's work [14]; an overview is given in [15]). The main restriction in the Nix evaluator, compared to optimal reduction, is that it only shares closed terms (as these are the only terms that eval ever sees). Thus, in an expression such as $(f : f\ 1 + f\ 2)(x : e)$, the expression e is duplicated, with only subterms of e not containing x being shared between the calls. On the other hand, the evaluator does optimise evaluation for terms that are not initially shared but become syntactically equal after a number of reduction steps.

7 Conclusion

This paper has given a practical demonstration of the use of maximal sharing as an implementation technique for interpreters. It shows that maximal sharing is efficient enough to allow the evaluation of a practical purely functional DSL to be completely memoised, giving rise to the highly useful property of maximal laziness.

Acknowledgements. This paper builds heavily upon the ATerm library and its property of maximal sharing, developed by Mark van den Brand, Hayco de Jong, Paul Klint and Pieter Olivier. I would also like to thank the contributors to the Nix package manager, in particular Martin Bravenboer and Eelco Visser, as well as the anonymous reviewers for their comments.

References

[1] Allen, J., "Anatomy of LISP," McGraw-Hill, Inc., New York, NY, USA, 1978.

[2] Baader, F. and T. Nipkow, "Term rewriting and all that," Cambridge University Press, 1998.

[3] Barendregt, H., "The Lambda Calculus: Its Syntax and Semantics," Studies in Logic and the Foundations of Mathematics **II**, Elsevier Science Publishers, Amsterdam, 1984, second edition.

[4] Cook, B. and J. Launchbury, *Disposable memo functions (extended abstract)*, in: *ICFP '97: Proc. ACM SIGPLAN Intl. Conf. on Functional programming* (1997), p. 310.

[5] Dolstra, E., "The Purely Functional Software Deployment Model," Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands (2006).

[6] Dolstra, E. and E. Visser, *Building interpreters with rewriting strategies*, in: M. van den Brand and R. Lämmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, Electronic Notes in Theoretical Computer Science **65/3** (2002), pp. 57–76.

[7] Dolstra, E., E. Visser and M. de Jonge, *Imposing a memory management discipline on software deployment*, in: *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)* (2004), pp. 583–592.

[8] Ershov, A. P., *On programming of arithmetic operations*, Commun. ACM **1** (1958), pp. 3–6.

[9] Filliâtre, J.-C. and S. Conchon, *Type-safe modular hash-consing*, in: *ML '06: Proceedings of the 2006 workshop on ML* (2006), pp. 12–19.

[10] Goto, E., *Monocopy and associative algorithms in an extended Lisp*, Technical Report TR-74-03, University of Toyko (1974).

[11] Goubault, J., *Implementing functional languages with fast equality, sets and maps: an exercise in hash consing*, in: *Journes Francophones des Langages Applicatifs (JFLA'93)*, 1993, pp. 222–238.

[12] Hudak, P., *Conception, evolution, and application of functional programming languages*, ACM Computing Surveys **21** (1989), pp. 359–411.

[13] Kahrs, S., *Unlimp — Uniqueness as a Leitmotiv for implementation*, in: M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Intl. Symposium on Programming Language Implementation and Logic Programming PLILP '92* (1992), pp. 115–129.

[14] Lamping, J., *An algorithm for optimal lambda calculus reduction*, in: *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1990), pp. 16–30.

[15] Lawall, J. L. and H. G. Mairson, *Optimality and inefficiency: what isn't a cost model of the lambda calculus?*, SIGPLAN Notices **31** (1996), pp. 92–101.

[16] Peyton Jones, S., "The Implementation of Functional Programming Languages," Prentice Hall, 1987.

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

[18] Peyton Jones, S., editor, "Haskell 98 Language and Libraries: The Revised Report," Cambridge University Press, 2004.

[19] Peyton Jones, S., W. Partain and A. Santos, *Let-floating: moving bindings to give faster programs*, in: *ICFP '96: Proc. ACM SIGPLAN Intl. Conf. on Functional programming* (1996), pp. 1–12.

[20] van den Brand, M. G. J., H. A. de Jong, P. Klint and P. A. Olivier, *Efficient annotated terms*, Software—Practice and Experience **30** (2000), pp. 259–291.

[21] van den Brand, M. G. J., J. Heering, P. Klint and P. A. Olivier, *Compiling language definitions: the ASF+SDF compiler*, ACM Trans. Program. Lang. Syst. **24** (2002), pp. 334–368.

[22] van den Brand, M. G. J. and P. Klint, *ATerms for manipulation and exchange of structured data: It's all about sharing*, Inf. Softw. Technol. **49** (2007), pp. 55–64.

[23] van Deursen, A., J. Heering and P. Klint, editors, "Language Prototyping. An Algebraic Specification Approach," AMAST Series in Computing **5**, World Scientific, Singapore, 1996.

[24] Visser, E., *Program transformation with Stratego/XT: Strategies, tools, and systems in StrategoXT-0.9*, in: C. Lengauer, D. S. Batory, C. Consel and M. Odersky, editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science **3016**, Spinger-Verlag, 2004 pp. 216–238.