Unlimp
Uniqueness as a Leitmotiv for Implementation

Stefan Kahrs[*]

University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ
email: smk@dcs.ed.ac.uk

Abstract. When evaluation in functional programming languages is explained using λ-calculus and/or term rewriting systems, expressions and function definitions are often defined as terms, that is as *trees*. Similarly, the collection of all terms is defined as a *forest*, that is a directed, acyclic graph where every vertex has at most one incoming edge. Concrete implementations usually drop the last restriction (and sometimes acyclicity as well), i.e. many terms can share a common subterm, meaning that different paths of subterm edges reach the same vertex in the graph.
Any vertex in such a graph represents a term. A term is represented uniquely in such a graph if there are no two different vertices representing it. Such a representation can be established by using *hash-consing* for the creation of heap objects. We investigate the consequences of adopting uniqueness in this sense as a leitmotiv for implementation (called Unlimp), i.e. not *allowing* any two different vertices in a graph to represent the same term.

1 Introduction

The definition of most programming languages is or can be based on some notion of term, e.g. the abstract syntax of the language. It is convenient to express properties of such terms as properties of tree-like objects, similarly as it is convenient to represent (in an implementation) a collection of terms as a directed acyclic graph, allowing the violation of the property that each vertex has an *indegree* of at most 1, i.e. that each vertex has at most one incoming edge.

If the language satisfies *referential transparency* for those terms, i.e. if the meaning of (closed) terms is context-independent and if this meaning is expressible as a term, one can moreover exploit the internal representation and destructively replace subgraphs by their results, even if their indegree is greater than 1.

Such graph reduction is the standard technique for implementing lazy languages, see [15, 4], because under lazy evaluation unevaluated subterms naturally occur. For implementing (general) term rewriting systems, graph reduction may lose confluence and weak normalisation, see [8], but rewriting systems in programming languages normally satisfy further properties that make graph reduction a correct implementation.

[*] The research reported here was partially supported by SERC grant GR/E 78463.

While acyclic graphs seem to be a natural choice for the internal representation of terms (cyclic graphs are not easily handled by a reference-counting garbage collector), one might also look at the extreme cases of this representation. There are two of particular interest: (i) the indegree of every vertex is at most 1 (trees and forests); (ii) the function that maps vertices to the terms they represent is injective, i.e. each represented term is represented uniquely.

The disadvantage of proper trees (i) is obviously the waste of space, but it also has advantages: memory management becomes easy, and sharing analysis [22] comes for free. For example, concatenation of two lists xs and ys in a graph representation usually works by copying xs and drawing an edge from the last vertex in the copy to ys, i.e. ys may become a shared object. In representation (i) however, it is known that xs and ys are uniquely used for the concatenation, hence it is not only possible to avoid copying ys, but also to avoid copying xs, using LISP's NCONC for list concatenation.

At first glance, the advantage of unique representation (ii) seems to be compactness, a further gain of space. But actually, it is more the uniqueness of representation itself, i.e. the property that a term can be uniquely identified by the root vertex of its representation, that turns out to be the major plus. The disadvantage is the effort needed to preserve this uniqueness under rewriting.

In this paper, we study this representation, how to get it and how to exploit it under the slogan "Uniqueness as a Leitmotiv for Implementation", short: Unlimp.

The examples are written in Haskell and in SML. Readers not familiar with these languages may consult [10] and [14].

2 Preliminaries

Instead of considering ordinary directed graphs, we deal with directed hypergraphs, i.e. we have directed hyperedges instead of ordinary edges. A hyperedge has in general more than one target (and also more than one source), we adopt a formal definition from [9]:

Definition 1. A *hypergraph* $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ over Σ consists of a finite set V_G of *vertices*, a finite set E_G of *hyperedges*, two mappings $s_G : E_G \to V_G^*$ and $t_G : E_G \to V_G^*$, assigning a string of *source vertices* and a string of *target vertices* to each hyperedge, and two mappings $l_G : V_G \to \mathrm{S}$ and $m_G : E_G \to \mathrm{OP}$, labelling vertices with sorts, and hyperedges with operation symbols.

The definition was originally intended for first-order signatures, $\Sigma = (\mathrm{S}, \mathrm{OP})$, but it can also be used in the higher-order case, simply by allowing non-elementary sorts in S, adding apply-symbols to OP, etc.

We also use some other standard graph-theoretic notions: $\mathsf{indegree}_G(v)$ denotes the sum (over all $e \in E_G$) of the number of occurrences of v in $t_G(e)$, analogously $\mathsf{outdegree}_G(v)$ for the $s_G(e)$. The subterm relation $\to_G$ (on vertices) is defined as follows:

$$v \to_G v' \quad \Leftrightarrow \quad \exists e \in E_G \ \exists a, a', b, b' \in V_G^* : s_G(e) = a \cdot v \cdot b \ \wedge \ t_G(e) = a' \cdot v' \cdot b'$$

A hypergraph is *acyclic* if the transitive closure of $\rightarrow_G$ is irreflexive. Notice that irreflexivity of $\rightarrow_G$ also implies that it is strongly normalising, as we have assumed a *finite* set of vertices.

Definition 2. A hypergraph G is a *jungle*, iff (i) it is acyclic, (ii) $\mathsf{outdegree}_G(v) = 1$ for all $v \in V_G$ and (iii) for all $e \in E_G, m_G(e) = f : s_1 \cdots s_n \rightarrow s$ implies $l_G^*(s_G(e)) = s$ and $l_G^*(t_G(e)) = s_1 \cdots s_n$.

l_G^* is the homomorphic extension of l_G to strings of vertices.

Acyclicity is useful for maintaining uniqueness of representation (see below), the restriction for the outdegree is motivated by the analogy between addresses and vertices, and the third condition is well-typedness.

The reason for this choice for the representation of terms is the very close correspondence between a jungle, vertices and hyperedges on the one hand, and a heap, addresses (pointers) and storage cells on the other, i.e. hypergraphs model implementations more faithfully than ordinary directed graphs. Therefore we will also freely intermix these notions, depending on whether it is in a particular case more intuitive or useful to talk about, say, a pointer rather than a vertex.

Given a jungle G, we define a function $\mathsf{term}_G : V_G \rightarrow \mathsf{Ter}(\Sigma)$ that assigns any vertex (address) the term it represents:

$$\mathsf{term}_G(v) \; = \; m_G(e) \cdot \mathsf{term}_G^*(t_G(e)) \textit{ where } s_G(e) = v \ .$$

The $*$ again denotes homomorphic extension to strings. Note that e is unique because of the outdegree restriction.

A jungle is a called *fully collapsed*, if term_G is injective. Because of this one-to-one correspondence we choose fully collapsed jungles to represent terms in Unlimp.

For any jungle there exists a (unique) fully collapsed jungle that represents the same set of terms. For this and some other results about representing terms by jungles and term rewriting by graph grammars, see [6, 9].

3 Hash Consing

Suppose we already have a fully collapsed jungle; then we have the problem of preserving this property each time we change the jungle, that is when we:

- create new objects
- evaluate an object
- delete an object

An object is anything represented by a vertex in a jungle. For the implementation of a functional programming language, objects could be values (elementary and composed values, functions), expressions yet to be evaluated, even non-closed expressions and type expressions (in an interpreter or compiler), etc.

How do we *create* a composed value $\mathsf{op}(x_1, ..., x_n)$ in a fully collapsed jungle (for some n-ary operation op)? Since we need a vertex that represents this value, there are two cases: either there is already a vertex v in V_G with $\mathsf{term}_G(v) = \mathsf{op}(x_1, ..., x_n)$, then we have to "find" it; or, if there is not, then we have to create a new cell (add a vertex and a hyperedge) and make sure that future searches can find it.

Typically, each x_i is already represented in the jungle, i.e. there is a vertex v_i in V_G with $\mathsf{term}_G(v_i) = x_i$. Because of Unlimp v_i is uniquely determined, as two vertices represent the same term iff they are identical. So any other hyperedge (in the jungle) pointing at a vertex that represents the same term as x_i, actually points at v_i.

To search for a cell $\mathsf{op}(x_1, ..., x_n)$ we could scan the entire heap, but that would be horribly inefficient. Since such a cell is uniquely determined by the vertices v_i and the operation op, these could give us a hint *where* we have to search. In other words, we can compute a search key from them, a value which is (almost) unique for the cell to be constructed. Because of the "almost" we still have to search, but our search space is very restricted.

Such a method is known as "hash consing", see [21, 17], because CONS is the only cell-constructing operation in LISP. It does not only apply for creating composite *values*, but for any kind of composite heap object, for example terms representing the application of a function to some other terms, also type expressions, etc. For the implementation of op, it is only a minor difference whether the cell to be created is supposed to be a value or some other heap object. We can even apply this method for creating λ-abstractions: If we lambda-lift [15, 4] all nested abstractions (such that any abstraction becomes a closed term) and then rename its variables[2] to x_0, x_1, etc., then α-congruence becomes trivial (pointer comparison).

In implementations of strict languages, one usually tries to avoid to create heap objects whenever possible, i.e. a term like `length [6,3,4]` would never exist as a vertex in the jungle. For reasons which become apparent later, we do not follow this line.

The author experimented with several ways to organize the heap. The method finally chosen (surely not the best one) is a combination of digital search trees and double hashing (see [5, 18] or some other standard book on data structures and algorithms): the search tree has hash tables as its leaves; searching an entry is done by using the leading bits of the key to branch in the tree, and finally the remaining bits are used for double hashing at a leaf.

4 Reduction

The second way to change the hypergraph is to *evaluate* a term.

Evaluation usually refers to the evaluation in the language itself, but we may apply it to a more general setting: the evaluation result of a type expression is the type expression one gets after substituting all type synonyms (e.g. in Haskell), the evaluation result of function definition is the code the compilation produces for it.

In a strict language implementation, an evaluation simply creates some new objects and makes some other objects (probably) obsolete. For Unlimp this view is harmless, because it reduces the problem of keeping the uniqueness to the previous one, to the creation of new objects.

[2] SML and Haskell have a generalised λ-abstraction (patterns instead of variables) that allows to abstract more than one variable in a single abstraction.

In an ordinary implementation of a lazy language the following happens: if a term t is evaluated to u, then the subhypergraph reachable from the vertex t is deleted and replaced by the subhypergraph reachable from u. This method is usually called *graph reduction* [4].

In Unlimp, this would not work well. Firstly, we might lose the injectivity of term_G, because if t is a subterm of some $C[t]$ then the vertex representing $C[t]$ now represents $C[u]$, but this term might already be represented by another vertex. For the implementation the situation is worse, because even if the jungle remains fully collapsed, the term $C[u]$ would be located at the wrong place in the search tree in our sketched implementation method. One could repair this mess by relocating all those terms in the search tree that have t as a direct[3] subterm, but then all (direct) superterms of t have to be found. Moreover, this method can introduce cycles[4], i.e. it may destroy our jungle structure. Allowing cycles would lead to (some restrictions for garbage collection and) the Unlimp problem for cycles, i.e. having a unique representation for any infinite term that can be expressed by a cyclic graph.

For these reasons, we do not replace t by u. On the other hand, we do not want to evaluate t a second time, staying as lazy as possible. Therefore, we draw an additional edge from t to u, a result edge. The hypergraph containing all the edges (hyperedges and result edges) may now be cyclic, but the result edges form a kind of second layer for the graph and both layers are in themselves acyclic.

A very simple example:

```
cycle xs = xs ++ cycle xs
```

A one-step evaluation for `cycle xs` (for an arbitrary `xs`) leads to the hypergraph in figure 1.

The three small circles are the vertices, the marked ellipses together with all incoming and outgoing arrows the hyperedges. The dotted arrow from the left to the right circle is a result edge. Looking just at the ordinary hyperedges, the picture says: "the result of `cycle xs` is `xs++cycle xs`", but thinking of the result edge as an indirection pointer, we have the full result `xs++xs++xs++...`. These result edges are not only a natural way to perform lazy evaluation; they are also useful for certain debugging tasks, like tracing a function, because the unevaluated expression and the evaluation result are available at the same time.

To allow these result edges in our hypergraph model, we add another component, a set of result edges to it:

Definition 3. A *result jungle* $J = (G, R_J, s_J, t_J)$ consists of a jungle G, a set of result edges R_J and two mappings $s_J, t_J : R_J \to V_G$, such that s_J is injective and $\forall e \in R_J :\ l_G(s_J(e)) = l_G(t_J(e))$.

Furthermore we call a result jungle *loopless* if the subterm relation $\to_H$ of any hypergraph $H = (V_G, R_J, s_J, t_J, \_, \_)$ is strongly normalising.

Result edges of the above form are simply partial, sort-preserving functions on vertices, but the above encoding within the hypergraph world preserves the close

[3] Example: t is a direct subterm of $f(t)$, but not of $f(f(t))$.

[4] According to [8], theorem 5.5, jungle reduction cannot introduce cycles, due to the realisation of rewrite steps chosen there. We do not consider this here, because it violates the Unlimp principle in a different way.

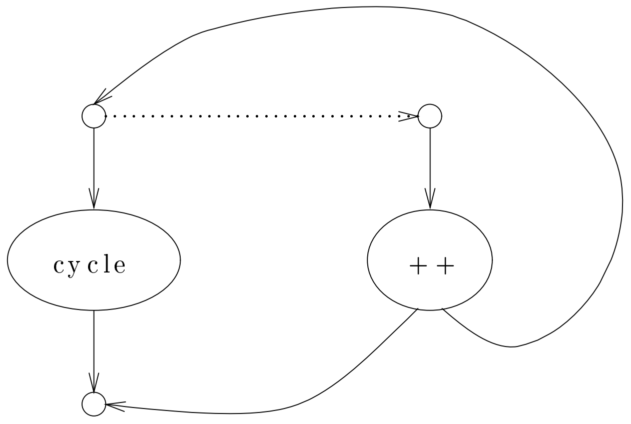

Fig. 1. Term with Result Edge

correspondence to implementations. For example, the storage cell corresponding to a result edge could be located at the address corresponding to its source vertex, either before or after the storage cell corresponding to the "ordinary" hyperedge going from there.

For loopless result jungles, we have another mapping term_J from vertices to terms, but the terms might be of infinite size, see [12]:

$$\begin{aligned}
&\mathsf{term}_J : V_G \to \mathsf{Ter}^\infty(\Sigma)\\
&\mathsf{term}_J(s_J(r)) = \mathsf{term}_J(t_J(r))\\
&\mathsf{term}_J(s_G(e)) = m_G(e) \cdot \mathsf{term}^*_J(t_G(e)),\ \text{if } \neg\exists r \in R_J :\ s_J(r) = s_G(e)
\end{aligned}$$

For any vertex v, $\mathsf{term}_G(v)$ is the (finite) term before evaluation and $\mathsf{term}_J(v)$ the (possibly infinite) term after it. The result jungle J has to be loopless to make the above definition of term_J well-defined. In figure 1, we have $\mathsf{term}_G(v) =$ `cycle xs` where v is top left vertex and `xs` the term represented by the bottom vertex, and $\mathsf{term}_J(v) =$ `xs'++xs'++...` where `xs'` is the term_J-value of the bottom vertex.

The definition of result jungles still allows chains of result edges. We can propagate results by the following relation $\trianglerighteq$ between result jungles:

$$\begin{aligned}
&(G, R_J, s_J, t_J) \trianglerighteq (G, R_J, s_J, t'_J) \text{ iff}\\
&\forall r \in R_J :\ t_J(r) = t'_J(r) \vee \exists r' \in R_J :\ s_J(r') = t_J(r) \wedge t_J(r') = t'_J(r)
\end{aligned}$$

For loopless result jungles, the irreflexive part $\triangleright$ of $\trianglerighteq$ is strongly normalising. Its (unique) normal forms are the result jungles with fully propagated result edges. The relation $\trianglerighteq$ does not change term_J, i.e. if $J \trianglerighteq J'$, $J = (G, R_J, s_J, t_J)$, then $\forall v \in V_G :\ \mathsf{term}_J(v) = \mathsf{term}_{J'}(v)$.

5 Memoization by Memo Tables

Memoization (sometimes called function caching) is a method to store evaluation results such that they can be reused if the same evaluation is required again later.

The traditional approach [13] uses memo tables, i.e. hash tables that store pairs of argument and result for those functions that are supposed to be memoized. From the hypergraph point of view, this method corresponds to a slightly different encoding of result edges, see [7]: a result hyperedge has then n sources (the n arguments of the function), one label (the function symbol) and one target, the result. One disadvantage, which is immediately obvious from the encoding itself, is a certain waste of space: the information the hyperedge has to carry comprises the function symbol, all the argument vertices and the result vertex.

Hughes [11] generalised memoization appropriately for lazy evaluation, storing as argument the (pointer to the) unevaluated argument in such a table. Pugh and Teitelbaum [16] showed how to widen the application of memoization to incremental computations (like attribute grammars) by carefully selecting the representation of the involved data types.

Beside the mentioned disadvantage that is apparent just by looking at the hypergraph encoding, memo tables have certain other drawbacks:

– they may overflow,
– they may be nearly unused,
– entries have to be searched for,
– they are oriented towards a first-order programming style.

In other words: they need some administration.

The last point refers to the problem: Where do we store the result of $(f \circ g)(x)$? The natural solution "in the memo table of $\circ$" seems fairly unreasonable, because it then heavily depends on the programming style whether the memo table for a combinator like $\circ$ is nearly empty, for programs written in first-order style, or totally overcrowded, as would be typical for programs developed in the Bird-Meertens formalism [2].

Figure 1 suggests a natural place for the result edge; it is the storage cell of the (unevaluated) expression. This means to make storage cells of expressions bigger (space for an additional pointer), provided the expression can have a value. This proviso is simply the negation of "is in weak head normal form" (for the terminology, see [15]), and this property is known when the cell is created.

This approach has a neat side-effect. Suppose, we create a cell $\mathsf{op}(t_1, ..., t_n)$. If a cell of this form already exists somewhere in the heap, Unlimp guarantees (and forces) us to find it, and if it has already been evaluated before, we will moreover find the result in this very cell. A memo table approach requires some further search: look up the memo table for op and then search for the appropriate n-tuple.

Similarly, if we store the compiled code of a λ-abstraction as its result, then creating the same (an α-congruent) λ-abstraction would find this compilation result, avoiding a superfluous recompilation. One could even give a pattern a value: the code to match it; this would not only guarantee to avoid recompilation of patterns, it might also ease the task of creating a decision tree[1] for pattern matching compilation.

6 Dealing with Side-Effects

Occasionally, the effect of memoization is unwanted, particularly in the presence of side-effects of various kinds, e.g. assignment and I/O, or – in an interpreter – a change of the rule base. To allow this in an Unlimp framework, one has to distinguish between applicative expressions, expressions that may depend on the state, and expressions that may change the state.

Expressions that may *change* the state have to be re-evaluated each time their value is required, hence we do not need a result edge for them. Expressions that may *depend on* the state but do not change it (like access to variables) have to be re-evaluated each time the state changes, hence result edges have to be time-stamped, *time* being a kind of side-effect counter. If the considered side-effects include the change of the rule base in an interpreter, every expression is state dependent and so each result edge needs a time stamp.

The properties "may change state" and "may depend on state" can be seen as simple syntactic properties of certain elementary operations (e.g. assignment the former, variable access the latter). But it is not quite obvious how they should be inherited by other operations or composite expressions. An abstract interpretation would probably provide a good approximation to the required information. But even in the absence of such an analysis, one can do the following:

For each expression (except whnf's), space for a result edge *and* its time stamp has to be provided. There are global counters for state changes and state accesses. If the evaluation of an expression t to some result u increases neither of the counters, we can draw an unstamped result edge. If the state change counter was increased, we do not draw a result edge; if the state remains unchanged but was accessed, the result edge gets the actual "time" (state change counter) as a stamp.

In the hypergraph world, we can encode this as follows:

Definition 4. A *changeable jungle* $C = (J, T, p)$ consists of a result jungle J, a number $T \in \omega$ (the time), and a mapping $p : R_J \to \omega + 1$ (the time stamp), such that $\forall r \in R_J :\ p(r) = \omega \vee p(r) \leq T$.

The effect of the time stamp can be described by a forgetful map from changeable jungles to result jungles, which maps $((G, R_J, s_J, t_J), T, p)$ to (G, R'_J, s_J, t_J), R'_J being the set $\{r \in R_J |\, p(r) \geq T\}$, i.e. the map forgets the expired result edges. The stamp ω indicates that the result edge is not state-dependent.

Result propagation for changeable jungles is slightly trickier than for result jungles, because instead of simply redirecting the target of certain result edges, we may have to add further result edges:

$$\begin{array}{l}
((G, R_J, s_J, t_J), T, p) \quad \trianglerighteq \quad ((G, R_J \cup R'_J, s_J \cup s'_J, t'_J), T, p \cup p') \text{ iff} \\
\forall r \in R'_J :\ p'(r) = T \ \wedge\ \exists a, b \in R_J :\ p(a) = \omega \ \wedge\ p(b) = T \ \wedge \\
\qquad s'_J(r) = s_J(a) \ \wedge\ t'_J(r) = t_J(b) \ \wedge\ s_J(b) = t_J(a) \\
\forall r \in R_J :\ t_J(r) = t'_J(r) \quad \vee \quad p(r) = T \ \wedge \\
\qquad \exists r' \in R_J :\ p(r') \geq T \ \wedge\ s_J(r') = t_J(r) \ \wedge\ t_J(r') = t'_J(r)
\end{array}$$

What this rather lengthy formula is all about can be seen in figure 2.

If an unmarked result edge is followed by a marked one, we can propagate the result as shown by the dotted line, but it would be a pity to overwrite the unmarked

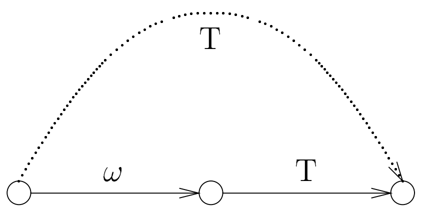

Fig. 2. Result Propagation for Changeable Jungles

edge: if there is a further change of state, the unmarked result is still valid while the marked result information expires. Therefore we have an additional set of result edges R'_J created in situations as above.

For the implementation this suggests the need to provide space for two result edges, a marked and an unmarked one, but this is not really necessary. If we restrict $\trianglerighteq$ to the case where R'_J is empty, then chains of (non-expired) result edges in the normal forms of $\triangleright$ have length at most 2.

Intuitively, this means the following. Suppose we have an evaluation sequence $t_0 \Rightarrow_1 t_1 \Rightarrow_2 \ldots \Rightarrow_n t_n$. We draw an unstamped result edge from t_0 to the last t_i, such that all the $\Rightarrow_j, j \leq i$ are applicative, and mark t_i as an *applicative normal form.* If $i < n$, we draw a stamped result edge from t_i to t_n, provided no $\Rightarrow_l, l \leq n$ changed the state.

Notice that this affects the notion of value: in addition to weak head normal forms there are now *applicative normal forms.*

The concept of a monolithic state is a bit strict, because it does not reflect locality of variables, e.g. (in SML):

```
fun fac n =
    let val p = ref (n,1) in
        while #1(!p) > 0
        do p := (#1(!p)-1, op * (!p));
        #2(!p)
    end;
```

The lifetime of the variable `p` does not exceed any call of `fac` and it is not accessible outside of `fac` – a dataflow analysis could easily detect this. We could exploit information of this kind for a more sophisticated concept of time and time stamp, but this goes beyond the scope of this paper.

7 Compilation

One subtask of compiling a function definition in a language that supports pattern matching is the management of a symbol table for the pattern variables. It assigns to each variable name a relative address (relative to the stack) and can furthermore be used to detect free variables, anonymous variables and non-linear patterns. Non-linear patterns are forbidden in most languages (not all), but even when they are allowed, the second occurrence of a variable in a pattern has to be treated differently from its first occurrence.

Under Unlimp, we can generalise the symbol table easily by treating not just variables, but arbitrary non-ground expressions. *Easily*, because comparing complex expressions is here not more difficult or expensive than comparing variables, since it is just the comparison of addresses.

The generalisation to non-ground expressions (`nge`) works as follows:

– An `nge` is allocated space on the stack, if and only of it occurs more than once in the left-hand or right-hand side of the definition.
– If an `nge` occurs a second time, we do not count its subterms as second occurrences.

Variables are also `nge`'s, and in this special case the first point is the detection of anonymous variables, because variables occurring only once do not need to be stored on the stack. For composite expressions it is a common subexpression elimination, because we put them onto the stack if they occur more than once, which corresponds to the introduction of a let-expression. An example (in Haskell), taken from [10]:

```
dropWhile p []  = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs
```

For the first rule, there are 3 `nge`'s, but none of them requires space on the stack, `p` occurs only once and is hence anonymous. In the second rule, we have 8 `nge`'s and 5 of them are allocated space on the stack, see table 1.

Table 1. Generalised Symbol Table

nge	occurrences
`dropWhile p (x:xs)`	1
`dropWhile p`	2
`p`	2
`x:xs`	2
`x`	2
`xs`	2
`p x`	1
`dropWhile p xs`	1

In this example, each `nge` which is to be stored on the stack is a subexpression of the left-hand side of the rule. Hence, when an expression matches the left-hand side, each `nge` to be stored on the stack is a subterm of this expression and can be stored during the matching process. One can argue about `nge`'s like `dropWhile p`; it depends on other implementation details (representation of function application) whether they should count or not.

Some care is necessary to treat conditional expressions properly, e.g. common subexpressions of the then- and else-parts of a conditional expression are not really common. It is harmless to put them onto the stack, but harmful to expect them to be there.

8 Garbage Collection

... is a weak point of Unlimp.

The problem is that there is very little *proper* garbage. Deallocating an unreferenced cell would also throw away its result edge and hence a bit of useful information, so that only unreferenced weak head normal forms (have no result edge) and former K-redexes[5] (result edge remains NIL under lazy evaluation) are proper garbage. Unfortunately, almost no weak head normal form will be unreferenced, at least there is the result edge from some (perhaps unreferenced) vertex, and K-redexes are more the exception than the rule. Only in the presence of side-effects can we expect some unreferenced weak head normal forms, because the time stamps of the result edges pointing to them may have expired.

For this reason, a garbage collector would need to collect improper garbage, which is against the spirit of Unlimp, of course. Each unreferenced vertex is (im)proper garbage. Even vertices only referenced by result edges could be treated as improper garbage, but this would require some additional administration, e.g. the garbage collection has to be treated as a global side-effect.

9 Programming Style

Working with an Unlimp implementation can influence programming style. First let us look at a similar influence of lazy evaluation.

Lazy and strict evaluation do not have the same computational power (in a *practical* sense), because lazy evaluation can deal with (conceptually) infinite objects, whereas strict evaluation cannot. Thus, when the natural solution of a problem requires the intermediate creation of an object of infinite size, solving the problem with a strict language means looking for a less natural way.

But such an influence on programming style is also present when there is no such principal difference in computational power, because for certain programming styles, strict evaluation is very inefficient. Typical for this are backtracking algorithms, see [20]; one example is the following simplified version (in Haskell) of the pairing algorithm used for Swiss System chess tournaments:

```
type Entry a = (a,[a])
type Pairing a = [(Entry a,Entry a)]
pairing :: (Eq a) => [Entry a] -> Pairing a
pairing table = if allpairs==[] then error "no pairing"
                else head allpairs
                where allpairs = fullpairs table
```

[5] In λ-calculus, $(\lambda x.t)u$ is a K-redex if x is not free in t.

```
fullpairs :: (Eq a) => [Entry a] -> [Pairing a]
fullpairs [] = [[]]
fullpairs (x:xs) = [ (x,y):zs | y <- xs, condition x y,
                                zs <- fullpairs (xs\\[y]) ]
condition :: (Eq a) => Entry a -> Entry a -> Bool
condition (x,xs)(y,ys) = notElem x ys
```

The function `pairing` is applied to the actual table of the players (which is supposed to be a list of even length) and produces a list of pairs (the pairing for the next round), such that each pair fulfills the `condition`. Moreover, the table leader should play (if possible) against the second, the third against the fourth, etc. The entries in the table consist of the player and his or her opponents so far, which is sufficient for the condition "haven't already played against each other".

The above algorithm is expressed in terms of computing all possible pairings and then selecting the first one, which – because of the structure of the algorithm – tends to pairs first with second etc. This is fine for lazy evaluation, but under strict evaluation it is very inefficient, because the number of all possible pairings usually (depending on `condition`) grows very fast. Table 2 shows the number of reduction steps (successful rule applications) executed to evaluate `pairing tab`, for five different examples[6], depending on whether the evaluation strategy is strict or lazy and whether full memoization is used or not.

Table 2. Reduction Steps for a Backtracking Algorithm

strategy	`stab`	`mtab1`	`mtab2`	`ltab1`	`ltab2`
strict, nomemo	347	16,401	18,211	1,776,421	1,865,213
strict, memo	184	2,133	2,068	84,117	84,361
lazy, nomemo	110	134	474	238	2,276
lazy, memo	88	128	240	230	540

Clearly, strict evaluation is inappropriate for this program. Although the algorithm is correct for strict evaluation too, a programmer using a strict language is encouraged to solve the problem on a lower level, e.g. by making the backtracking strategy explicit.

The impact of memoization on the program is characteristic: the "better" the algorithm is, the less is the effect of memoization. It cannot turn a horribly slow program into a fast one, but it can reduce the horror drastically. The drastic improvement under strict evaluation, and the slight but significant improvement for the heavy backtrackers (`mtab2` and `ltab2`) under lazy evaluation are rather surprising, as the `pairing` program does not appear to be a prime candidate for memoization.

Full memoization can work together with lazy as well as strict evaluation, but it does not affect the computational power of either strategy. Therefore, there is no

[6] The chosen examples were lists of length 6 after 2 rounds (`stab`), of length 10 after 3 rounds (`mtab1` and `mtab2`), and of length 14 after 4 rounds (`ltab1` and `ltab2`). The examples `mtab2` and `ltab2` were chosen to require a lot of backtracking, in contrast to `mtab1` and `ltab1`.

principle need to change the programming style when memoization is absent, but we do have similar kinds of unpleasant encouragement to solve problems at a lower level.

Some further examples (and references) can be found in [11]. We do not have to look for examples that are contrived to support this argument – the following piece of program (in SML) to compute the nth prime number was taken from [19]:

```
fun prime n =
    let fun next(k,i) =
            if n<=i then k
            else if divides(prime i,k) then next(k+1,0)
            else next(k,i+1)
    in
        if n=0 then 2
        else next(prime(n-1)+1,0)
    end
```

It was considered there to be "rather inefficient". In a traditional implementation it is indeed, but under Unlimp it turns out to be fairly reasonable, because memoizing `prime` makes the algorithm behave like a (rather naïve) variation of the sieve of Eratosthenes.

10 Speed-Up in the Small

Most examples people mention when they promulgate memoization are like the naïve version of the Fibonacci function or the above version of `prime` - without memoization terribly inefficient and - since they are naïve - only naïve people would write the function this way, unless it is known that the implementation supports memoization[7].

But memoization also has great effects in the small, as in the `pairing` program. Sometimes they appear very unexpectedly, like the following one:

As their favoured benchmark test for functional programs, Jörn von Holten and Richard Seifert at the University of Bremen took arithmetic on natural numbers represented as successor terms. To make the task hard, the following version of arithmetic was used:

```
data Nat    = Z | S Nat
add Z     x = x
add (S x) y = S (add x y)
mul Z     x = Z
mul (S x) y = add (mul x y) y
pow x     Z = S Z
pow x (S y) = mul (pow x y) x
```

This version is supposed to make arithmetic expensive, because (minor reason) `add` is not tail recursive and (major reason) the right-hand sides of the last rules for

[7] Another less well-known example of this kind is model checking with binary decision diagrams, see [3].

`mul` and `pow` have their recursive calls in the first rather than the second argument of `add` and `mul`. Note that `add n m` is linear in `n` and constant in `m`.

However, the response time of an Unlimp implementation turned out to be fairly stable under switching the arguments of `add` and `mul` in the mentioned rules. The reason is that several addition terms reappear in this process, because computing $n + m$ involves also the computation of $k + m$ for all k less than m.

The following table compares the number of evaluation steps to compute 4^2, 4^3, and 4^4. The left figures show the number of steps for the above definition, the right figures refer to the version obtained by switching the arguments of the mentioned calls of `add` and `mul`.

Table 3. Reduction Steps for a Successor Arithmetic

strategy	4^2		4^3		4^4	
strict, nomemo	40	42	554	116	8748	382
strict, memo	22	40	83	109	324	358
lazy, nomemo	96	195	444,678	1,611	too many	13,123
lazy, memo	27	42	192	116	2295	382

The suspected bad behaviour of exponentiation does not appear under memoization and strict evaluation, here it is even slightly better than the ordinary definition. Only for lazy evaluation, memoization cannot fully compensate for the “bad” algorithm.

As in the `pairing` example, we can again observe different kinds of improvement, depending on how “badly” the algorithm behaves. In both cases, the effects appeared in the small, i.e. they had no fancy recursive structure (as the `prime` example), the functions were linear recursive.

11 Conclusion

A unique representations for expressions can affect compilation, execution and usage of functional languages.

We tried to convey the spirit of thinking in unique representations and of exploiting it for different purposes, e.g. for compilation. The given modelling by hypergraphs stays close to the machine level and allows several meta-observations on a rather abstract level. We showed how memoization and side-effects can happily coexist, even in the hypergraph modelling.

The effect of memoization on program execution seems to be well-known, but the analysis of the given examples suggest that it is not well-known enough. When using full memoization, i.e. storing *every* evaluation result, an important and often unexpected phenomenon appears: a cumulative speed-up by saving minor, but numerous computations. This phenomenon encourages a more problem-oriented programming style.

References

1. Marianne Baudinet and David MacQueen. Tree pattern matching for ML. In *Conference on Functional Programming Languages and Computer Architecture*, 1987. LNCS 274.
2. Richard Bird. Lectures on Constructive Functional Programming. Technical Monograph PRG-69, Programming Research Group, Oxford University Computing Laboratory, 1988.
3. Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for symbolic model checking in CCS. In *Third Workshop on Computer Aided Verification*, pages 263–278, Aalborg, 1991.
4. A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.
5. G. H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1984.
6. Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle Evaluation. *Fundamentae Informaticae*, 15(1):37–60, 1991.
7. Berthold Hoffmann. Term Rewriting with Sharing and Memoïzation, 1992. (to appear in: Proceedings, Algebraic and Logic Programming 1992).
8. Berthold Hoffmann and Detlef Plump. Jungle Evaluation for Efficient Term Rewriting. Technical Report 4/88, Universität Bremen, Studiengang Informatik, 1988. (short version in LNCS 343).
9. Berthold Hoffmann and Detlef Plump. Implementing Term Rewriting by Jungle Evaluation. *Informatique théorique et Applications*, 25(5):445–472, 1991.
10. P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell, a Non-strict, Purely Functional Language. Technical report, University of Glasgow, 1992. (also in SIGPLAN Notices 27(5), May 1992).
11. R. J. M. Hughes. Lazy memo functions. In *Conference on Functional Programming and Computer Architecture*, pages 129–146. Springer, 1985. LNCS 201.
12. Jan Willem Klop and Roel de Vrijer. Extended Term Rewriting Systems. Technical Report CS-R9107, Centrum voor Wiskunde en Informatica, January 1991.
13. D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.
14. Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.
15. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
16. William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Symposion on Principles of Programming Languages*, pages 315–328, 1989.
17. Masataka Sassa and Eiichi Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(2):31–34, June 1976.
18. Robert Sedgewick. *Algorithms.* Addison-Wesley, 1988.
19. Stefan Sokołowski. *Applicative High Order Programming.* Chapman & Hall Computing, 1991.
20. Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, pages 113–128, 1985. LNCS 201.
21. Ben Wegbreit and Jay M. Spitzen. Proving properties of complex data structures. *Journal of the ACM*, 23(2):389–396, 1976.
22. Burkhart Wolff. Sharing-Analyse in funktionalen Sprachen. Technical Report 6/91, Universität Bremen, Studiengang Informatik, 1991. (in German).

This article was processed using the LaTeX macro package with LLNCS style