Heap Recycling for Lazy Languages

Jurriaan Hage Stefan Holdermans

Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{jur,stefan}@cs.uu.nl

Abstract

Pure functional programming languages preclude destructive updates of heap-allocated data. In such languages, all newly computed algebraic values claim freshly allocated heap space, which typically causes idiomatic programs to be notoriously inefficient when compared to their imperative and impure counterparts. We partly overcome this shortcoming by considering a syntactically light language construct for enabling user-controlled in-place updates of algebraic values. The resulting calculus, that is based on a combination of type-based uniqueness and constructor analysis, is guaranteed to maintain referential transparency and is fully compatible with existing run-time systems for nonstrict, pure functional languages.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Theory

Keywords compile-time garbage collection, lazy functional programming, type-based program analysis

1. Introduction

Functional programming languages can be classified along several axes: we can distinguish between pure and impure langauges as well as between langauges with strict and nonstrict semantics. In practice, not all combinations make sense. Nonstrict languages better be pure, because reasoning about unrestricted side-effects becomes more complicated when the order of evaluation gets less predictable.

Purity has some clear advantages. For example, it enables equational reasoning and it opens the road to memoization, common subexpression elimination, and parallel evaluation strategies. The driving force that enables these opportunities is referential transparency: in a pure language, each of a program's terms can, at any time, be replaced by its value without changing the meaning of the program as a whole.

Referential transparency can only be achieved if side-effects are excluded from the language—or at least dealt with in some special way. In the language Haskell [18], for instance, all potentially side-effecting computations are encapsulated in a monad [20]. Unfortunately, a monadic programming style typically differs fundamentally from an ordinary functional style. While this is not as much of a problem if the programming task at hand is side-effecting by nature, it becomes an issue when side effects are only involved peripherally: then, a programmer will be keen to maintain a functional look and feel to the program and not have encapsulation of side effects dominate the overall style of the program.

Consider, for example, in-place updates of data structures. Overwriting arbitrary values in memory is a definite threat to referential transparency and, in a pure language, will have to be dealt with carefully. If in-place updates are a key ingredient of an algorithm—as they are, for instance, of the union-find algorithm—it makes good sense to make them available in a controlled, monadic setting (see, for instance, Launchbury and Peyton Jones [16]) and have the programmer write down the algorithm in monadic style. However, if a single in-place update is only needed for some localized performance tuning to an otherwise completely functional snippet of code, we really want to provide the programmer with an opportunity to add the performance tweak without being forced to abandon a purely functional style of coding. In order to keep the advantages of such a style, the compiler then has to make sure that the tweak does not compromise referential transparency.

To this end, we present a small and syntactically light language construct that indeed enables programmers to incorporate in-place updates of data structures in idiomatic functional programs. Specifically, our contributions are the following:

- We embed a destructive assignment operator in a purely functional expression language in order to provide some lightweight support for *compile-time garbage collection*. Judicious use of the operator enables the programmer to take explicit control over the reuse of heap cells that are already allocated but, provably, no longer in use. We illustrate and motivate its use by means of some concrete examples (Section 2).
- A formal account of the proposed construct is given in terms of a dynamic and a static semantics for a small higher-order let-polymorphic call-by-need lambda-calculus with lists (Section 4). The dynamic semantics are designed to be compatible with existing memory models for lazy functional languages, while the static semantics rely on a combination of type-based usage analysis and constructor analysis (Section 3).
- We briefly discuss some of the design space around our proposal and sketch some variations on our approach (Section 5).

2. Motivation

Before we delve into the more technical aspects of our proposal, let us first motivate and demonstrate our approach by means of some examples.

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

2.1 List Reversal

Consider the standard Haskell program for list reversal:

```
reverse l = rev l []
  where
    rev []     acc = acc
    rev (x:xs) acc = rev xs (x:acc).
```

The definition of the function *reverse* makes use of a so-called accumulating parameter [2]. The locally defined helper function *rev* traverses the spine of the input list l and constructs a fully new spine for the result list. A potential source of inefficiency lies in the fact that, in typical implementations of lazy functional languages, a fresh heap cell is allocated for each element in the reversed list. However, in situations in which the input to *reverse* is known to be used only once, as, for example, in the following program,

```
main = do
  input ← readFile "in"
  acc "out" (reverse input),
```

we know that each time *rev* needs to construct a cons-node, an already allocated heap cell has just become available for reuse: the cell in which the cons-node that we pattern matched against was stored.

Therefore, we propose a small syntactic extension to lazy functional languages like Haskell that allows programmers to take advantage of this knowledge and explicitly recycle available heap cells. With this extension, an in-place version of *reverse* is

```
reverse• l = rev l []
  where
    rev []     acc = acc
    rev l@(x:xs) acc = rev xs l@(x:acc).
```

Here, we have used a conjunctive pattern $l@(x:xs)$ in the cons-case of the helper function: in Haskell, such a pattern provides the programmer with a means to associate a name with a value that is succesfully matched against a specific pattern. Our extension amounts to also allowing the @-construct at the right-hand side of function definitions. There, a term of the form $x@e$, where x is a variable and e a constructor expression, denotes that the heap cell in which the value associated with x is stored is to be overwritten with the result of e. So, in our example, in the second case of *rev*, we have indicated that the memory occupied by the top-level constructor of $(x:xs)$ is to be reused in the construction of $(x:acc)$.

In our proposal, we expect the compiler to ensure referential transparency and, hence, a program that uses a function like *reverse*$^\bullet$ may fail to compile. Effectively, this means that, by default, explicit overwrites of heap cells are prohibited, unless we can be absolutely certain that the overwritten heap cells do not store data that is still in use. In the *main* program above, this is indeed the case: *reverse* has private access to the list *input* and so it is safe to replace the call to *reverse* by a call to *reverse*$^\bullet$. Determining whether or not a value can still be in use is a delicate task that should therefore be handled by the compiler rather than the programmer (see Sections 3 and 4).

2.2 Sorting

As a second example, we look at a program in which the declarative beauty of functional languages is nicely manifested: the well-known quicksort algorithm.

```
qsort []     = []
qsort (x:xs) =
  qsort (filter (<x) xs) ++ [x] ++ qsort (filter (≥ x) xs)
```

Although this version of quicksort has some obvious notational advantages over an imperative implementation in, say, C, there are also a few major disadvantages. Unsurprisingly, these have to do with the fact that the efficiency of the above implementation is way behind that of an in-place C-implementation.

Even so, the execution time of the functional version can be improved easily by avoiding the two separate list traversals and instead using a single function for breaking up the list in two parts. Such a function takes a pivot k and a list l, and splits l into two sublists: one with elements smaller than k and one with elements at least k.

```
split• k l = pivot l [] []
  where
    pivot []     accl accr = (accl, accr)
    pivot l@(x:xs) accl accr
      | x < k              = pivot xs l@(x:accl) accr
      | otherwise          = pivot xs accl l@(x:accr)
qsort• l@[]     = l
qsort• l@(x:xs) = qsort• left ++• l@(x:qsort• right)
  where
    (left, right) = split• x xs.
```

Except for the update markers at the right-hand sides of the definitions, this is just an idiomatic functional program. The update markers are there to enforce that *qsort*$^\bullet$ operates in-place, allocating only a constant amount of fresh heap space. Referential transparency is guaranteed under the provision that, after sorting, the input list is no longer used.

The presented example makes use of an append operator that destructively reuses its first argument:

```
[]     ++• ll = ll
lr@(x:xs) ++• ll = lr@(x:(xs ++• ll)).
```

2.3 Rotations in Binary Search Trees

In the construction of efficient search-tree representations, such as AVL-trees and red-black trees [5], invariants are often maintained with help of *rotations*. Using update markers, a functional implementation of in-place left rotations for binary search trees,

```
data Tree a = Leaf | Node (Tree a) a (Tree a),
```

can be written as follows:

```
rotate• t@Leaf       = t
rotate• t@(Node l x r) = rot r
  where
    rot Leaf           = t
    rot r@(Node lr y rr) = r@(Node t@(Node l x lr) y rr).
```

In the second case, r is the right child of the input node t, with lr the left child of r. In the rotated tree, t is the left child of r and lr the right child of t, and so the search-tree property is retained.

In an imperative setting, rotations in binary search trees typically amount to redirecting two pointers—and, effectively, this is just what happens in the snippet above.

3. Approach

In the previous section, we have presented some examples that illustrate how update markers can be used for deriving destructive, space-efficient versions of idiomatic functional programs. In this section, we switch from the programmer's perspective on explicit heap-recycling annotations to that of the language designer.

In particular, we provide answers to two important questions that arise when one considers adding programmer-controlled destructive updates to a lazy language:

1. How do we ensure that destructive updates do not compromise referential transparancy? and
2. How do we ensure that destructive updates make sense with respect to the underlying memory model?

It turns out that both issues can be resolved by putting restrictions on the programs in which update markers occur. In our proposal, these restrictions are enforced statically: the compiler rejects programs that do not satisfy the imposed constraints.

3.1 Maintaining Referential Transparency

Our main motivation for proposing explicit update markers for pure functional languges was our desire to write idiomatic functional code even in the presence of destructive updates. In a lazy setting, besides retaining a functional style of programming, we also want to keep referential transparency, so that reasoning about programs remains manageable.

Obviously, maintaining referential transparency in the presence of destructive updates is far from trivial. Consider, for example, the following program:

let $l = [1\,..\,10]$
in *filter odd* $l +\!\!+$ *filter even* l.

One would expect it to produce the list $[1,3,5,7,9,2,4,6,8,10]$—and so it does. However, if we replace the calls to the standard *filter* function by calls to a version that reuses parts of the spine of its input list to produce its output,

$$\begin{array}{lll} \mathit{filter}^\bullet\ p\ l@[\,] & & = l \\ \mathit{filter}^\bullet\ p\ l@(x:xs) & \mid p\ x & = l@(x:\mathit{filter}^\bullet\ p\ xs) \\ & \mid \mathit{otherwise} & = \mathit{filter}^\bullet\ p\ xs, \end{array}$$

we actually end up with the list $[1,3,5,7,9]$, because the second invocation of *filter*$^\bullet$ is then bound to traverse a list that contains nothing but odd numbers. In general, such referential opaqueness renders reasoning about programs a tedious undertaking.

So, to be able to use update markers without giving up referential transparency, we have to come up with a means to exclude programs like the one described above. In our approach, we do so by only allowing destructive updates to be performed on values that flow through a program in a *single-threaded* fashion. In the expression *filter*$^\bullet$ *odd* $l +\!\!+$ *filter*$^\bullet$ *even* l, for example, the value associated with l branches over the operands of $(+\!\!+)$ and, hence, does not flow single-threadedly.

To keep track of single-threadedness, we subject programs to uniqueness analysis [1, 9]. To this end, we adopt a static typing discipline in which we annotate the typing judgements for each term with so-called uniqueness annotations. Such an annotation is either 1, to indicate that a term is used single-threadedly, or ω, to indicate that a term may flow multi-threadedly. For instance, a possible typing judgement for the list l from the example above may then read

$$\Gamma \vdash l :^{\omega} \mathsf{List}\ \mathsf{Int}^{\omega}.$$

Here, the annotations on the copula and on the type constructor Int indicate that, respectively, the spine of the list l and its elements may be used multi-threadedly. Since the function *filter*$^\bullet$ performs a destructive update on the spine of its argument list, the typing of l is not compatible with any valid typing of *filter*$^\bullet$ *even*. An example of such a typing for *filter*$^\bullet$ *even* is

$$\Gamma \vdash \mathit{filter}^\bullet\ \mathit{even} :^{\omega} (\mathsf{List}\ \mathsf{Int}^{\omega})^{1} \xrightarrow{\omega} (\mathsf{List}\ \mathsf{Int}^{\omega})^{\omega}.$$

The restrictive nature of *filter*$^\bullet$ *even* is expressed by the 1-annotation on its domain: this annotation indicates that the function can only consume arguments that are guaranteed to be used single-threadedly. The ω-annotations in the typing of *filter*$^\bullet$ *even* express that the function itself may be passed around multi-threadedly (the annotation on the copula), that the elements of both the argument and the result list may be passed around multi-threadedly (the annotations on the occurrences of the type constructor Int), that the result list may be passed around multi-threadedly (the annotation on the codomain), and that use of the function is not subjected to any containment restriction (the annotation on the function-space constructor; see Section 4.3 for a discussion of containment).

3.2 Fitting the Memory Model

Now that we have ensured that updates do not destruct values that are still in use, we have to consider how to determine whether or not a programmer-proposed update can actually be performed within the boundaries of the underlying memory model.

What happens for example, for lists, if we try to update a nil-cell with a cons-cell? A common in-memory representation for values of algebraic data types consists of (a pointer to) some constructor-related information followed by a vector of pointers that refer to the representations of the constructor arguments. But then, overwriting a nil-cell with a cons-cell is problematic, because the amount of space reserved for a nil-cell (a single pointer) is not enough to contain a cons-cell (three pointers).

It should be noted that some memory models, most notably the model that is used by the GRIN-system [4, 3], actually do allow cells to be updated with values that require more space than the original cell. This typically amounts to splitting the new cell in two parts and having the last word of the first part point to the first word of the second part. However, using such a scheme for destructive updates in our situation defeats, to a large extent, its purpose for it still requires us to allocate fresh heap space for the second part of the cell.

The other way around, updating a larger cell with a smaller cell, could also be troublesome. Such an update would leave the last part of the original cell, i.e., the part that is not used by the replacing cell, as garbage. To reclaim this superfluous space, special arrangements have to be made by an automatic garbage collector and each cell then needs to store, in addition to the constructor information and pointers to its arguments, the total number of unused memory words that it has trailing. But this per-cell overhead would inevitably have a dramatic negative impact on the overall space consumption of programs!

These observations lead us to only considering updates that involve equally sized cells—a restriction we already conformed to in the examples from Section 2. In fact, we adopt an even more restrictive scheme: we only allow values of algebraic data types to be updated by values that were built by the same constructor as the original value. We believe this is the most transparent scheme; possible relaxations are discussed in Section 5.

To be able to enforce the described discipline, we need the compiler to flag updates that involve different constructors invalid. Now, approximately determining at compile time by which constructor a potential data value will have been built, is relatively easy and amounts to some straightforward data-flow analysis. Again, we choose a type-based approach.

In the following definition, for instance,

$$\begin{array}{ll} \mathit{rev}\ [\,] & \mathit{acc} = \mathit{acc} \\ \mathit{rev}\ l@(x:xs) & \mathit{acc} = \mathit{rev}\ xs\ l@(x:\mathit{acc}), \end{array}$$

uniqueness analysis requires arguments that are bound to the parameter l to be used single-threadedly (cf. Section 3.1):

$$\Gamma \vdash l :^{1} \forall \mathsf{a}.\ \mathsf{List}\ \mathsf{a}^{\omega}.$$

In addition, for the second case, our data-flow analysis determines that the only values for l that can flow into the right-hand side are those that are constructed with the cons-constructor $(:)$. This

Identifiers

$x \in \mathbf{Var}$		*(term variables)*
$\alpha \in \mathbf{TyVar}$		*(type variables)*
$\beta \in \mathbf{AnnVar}$		*(annotation variables)*
$h \in \mathbf{Loc}$		*(heap locations)*

Terms

$a \in \mathbf{Atm}$	$::=$	$x \mid x@Nil \mid x@(Cons\ x\ x)$
$t \in \mathbf{Tm}$	$::=$	$a \mid \lambda x.\ t \mid t\ a \mid \mathbf{let}\ x = t\ \mathbf{in}\ t$
	$\mid$	$Nil \mid Cons\ x\ x$
	$\mid$	$\mathbf{case}\ x\ \mathbf{of}\ \{Nil \Rightarrow t;\ Cons\ x\ x \Rightarrow t\}$

Types

$\varphi \in \mathbf{UnqAnn}$	$::=$	$1 \mid \omega \mid \beta$
$\psi \in \mathbf{ConAnn}$	$::=$	$\epsilon \mid Nil \mid Cons$
$\pi \in \mathbf{Qual}$	$::=$	$\varphi \sqsubseteq \varphi$
$\tau \in \mathbf{Ty}$	$::=$	$\alpha \mid \tau^{\varphi} \xrightarrow{\varphi} \tau^{\varphi} \mid \mathsf{List}\ \tau^{\varphi}$
$\rho \in \mathbf{QTy}$	$::=$	$\tau \mid \pi \Rightarrow \rho$
$\sigma \in \mathbf{TyScheme}$	$::=$	$\rho \mid \forall\alpha.\ \sigma \mid \forall\beta.\ \sigma$
$\Gamma \in \mathbf{Ctx}$	$::=$	$\emptyset \mid \Gamma, x :^{\varphi\mid\psi} \sigma \mid \Gamma, \pi$

Evaluation

$w \in \mathbf{Whnf}$	$::=$	$Nil \mid Cons\ x\ x \mid \lambda x.\ t$
$H \in \mathbf{Hp}$	$::=$	$\emptyset \mid H, h \mapsto (t;\ \eta) \mid H, h \mapsto h$
$\eta \in \mathbf{Env}$	$::=$	$\emptyset \mid \eta, x \mapsto h$

Figure 1. Syntax

information is stored in the context Γ, where we replace the binding for l by a binding that is annotated with information about the constructor:

$$l :^{1\mid(:)} \forall \mathsf{a}.\ \mathsf{List}\ \mathsf{a}^{\omega}.$$

Then, at the update site $l@(x : xs)$, all information that is needed to flag this particular use of the update marker correct is available in the context.

A more formal account of both our uniqueness analysis and our constructor analysis is given in the next section.

4. Formalities

To formalize our approach, we present a small higher-order let-polymorphic call-by-need lambda-calculus with lists and explicit update markers. We proceed by discussing its abstract syntax (Section 4.1), and its dynamic and static semantics (Sections 4.2 and 4.3, respectively). In Section 4.4, we list some properties of the calculus.

4.1 Syntax

The abstract syntax of our calculus is given in Figure 1.

Assuming an infinite supply of identifier symbols, terms are built from variables, updates, lambda-abstractions, function applications, local definitions, constructor expressions, and case analyses. Constructors can only be applied to variables, while functions applications are restricted to atomic argument terms. Such an atomic term is either a variable or an update. (These syntactic restrictions merely facilitate the definition of a relatively straightforward dynamic semantics and are by no means essential: constructor applications with nonvariable arguments and function applications with nonatomic arguments are easily desugared into terms that meet the restrictions. To do so, one simply introduces fresh let-bindings for the noncompliant arguments.)

We use the notation $\mathrm{fv}(t)$ to refer to the set of variables that appear free in t. As always, we assume substitution of free variables to be capture avoiding, performing alpha-renaming when necessary.

In the type language, types are constructed from type variables, function types, and list types. Types are annotated with uniqueness anotations φ. A uniqueness anotation is either one of the constants 1 and ω, or an annotation variable β. We impose a total order on uniqueness annotations, characterized by $1 \sqsubset \omega$. During typing, the order between annotations is captured in qualifiers [13] of the form $\varphi_1 \sqsubseteq \varphi_2$. These qualifiers can be stored in type schemes, which are obtained by quantifying over type and annotation variables. Typing contexts are formed by bindings for variables and by qualifiers. Bindings for variables are annotated with both a uniqueness annotation and a constructor annotation: $x :^{\varphi\mid\psi} \sigma$. A constructor annotation is either one of the constructors *Nil* and *Cons* or a special constant ϵ that denotes the absence of information. We write $\Gamma \setminus x$ for the context obtained by removing all bindings for x from Γ.

The set of free type variables in a context Γ is written as $\mathrm{ftv}(\Gamma)$. Likewise, the set of free annotation variables in Γ is written as $\mathrm{fav}(\Gamma)$. Occasionally, contexts are treated as finite maps from variables to types schemes and annotation pairs, and we write $\Gamma(x)$ for the type scheme and annotation pair that are associated with the rightmost binding for x in Γ.

In our dynamic semantics, heaps are modelled as bindings of heap locations h to either closures or other heap locations. The latter type of binding is referred to as an *indirection*. Closures consist of a term and an environment, while environments bind variable names to heap locations. During evaluation, terms reduce to weak-head normal forms, which are either constructor applications or lambda-abstractions.

We write $H(h)$ for the closure or location that is associated with the rightmost binding for h in the heap H and, similarly, we let $\eta(x)$ denote the location that is associated with the rightmost occurrence of x in the environment η. Additionally, each heap H gives rise to a metafunction $\overline{H} : \mathbf{Loc} \to \mathbf{Loc}$ that "traverses" chains of indirections until it hits a closure:

$$\overline{H}(h) = \begin{cases} h & \text{if } H(h) = (t;\ \eta), \\ \overline{H}(h') & \text{if } H(h) = h'. \end{cases}$$

4.2 Dynamic Semantics

The dynamic semantics of our calculus is given as a Launchbury-style natural semantics for call-by-need evaluation [15]. The main departure from Launchbury's approach is that we model variable binding by means of an explicit mapping from names to locations, whereas in Launchbury's formulation, variables and memory locations coincide. Because, in our system, updateable structures are identified by names, we believe our technique is more convenient in the present setting as it explicates name management.

The semantics is presented in Figures 2 and 3 as a deduction system with judgements of the form

$$H;\ \eta;\ t \Downarrow_n H';\ \eta';\ w.$$

The semantics is instrumented in the sense that it associates with each judgement of the evaluation relation a natural number n that indicates how many fresh heap locations are required by the evaluation.

The result of evaluating a term t in the context of a heap H and an environment η, is captured by a triple consisting of an updated heap H', an updated environment η', and a weak-head normal form w. The rules of the evaluation relation maintain the invariant that heap indirections never point to other indirections, i.e., the size of any chain of indirections is at most 1. Moreover, indirections always point to closures that contain weak-head normal forms.

Lambda-abstractions and saturated constructor applications are already in weak-head normal form and, hence, require no further evaluation. This is expressed by the rules E-ABS, E-NIL, and E-CONS in Figure 2.

Evaluation $\boxed{H;\ \eta;\ t \Downarrow_n H';\ \eta';\ w}$

$$H;\ \eta;\ \lambda x.\ t_1 \Downarrow_0 H;\ \eta;\ \lambda x.\ t_1 \qquad \text{(E-Abs)}$$

$$H;\ \eta;\ \mathit{Nil} \Downarrow_0 H;\ \eta;\ \mathit{Nil} \qquad \text{(E-Nil)}$$

$$H;\ \eta;\ \mathit{Cons}\ x_1\ x_2 \Downarrow_0 H;\ \eta;\ \mathit{Cons}\ x_1\ x_2 \qquad \text{(E-Cons)}$$

$$\frac{\begin{array}{c}\eta(x) = h \\ H(h) = (t'';\ \eta'') \quad t'' \notin \mathbf{Atm} - \mathbf{Var} \\ H;\ \eta'';\ t'' \Downarrow_n H'';\ \eta';\ w \quad h' \notin \mathrm{dom}(H) \\ H' = H'', h' \mapsto (w;\ \eta'), h \mapsto h'\end{array}}{H;\ \eta;\ x \Downarrow_{n+1} H';\ \eta';\ w} \qquad \text{(E-VarClose)}$$

$$\frac{\eta(x) = h \quad H(h) = h' \quad H(h') = (w;\ \eta')}{H;\ \eta;\ x \Downarrow_0 H;\ \eta';\ w} \qquad \text{(E-VarInd)}$$

$$\frac{\begin{array}{c}H;\ \eta;\ t_1 \Downarrow_{n_1} H'';\ \eta'';\ \lambda x''.\ t'' \quad \eta(x) = h \\ H'';\ \eta'', x'' \mapsto h;\ t'' \Downarrow_{n_2} H';\ \eta';\ w\end{array}}{H;\ \eta;\ t_1\ x \Downarrow_{n_1+n_2} H';\ \eta';\ w} \qquad \text{(E-AppVar)}$$

$$\frac{\begin{array}{c}h \notin \mathrm{dom}(H) \quad \eta'' = \eta, x \mapsto h \\ H, h \mapsto (t_1;\ \eta'');\ \eta'';\ t_2 \Downarrow_{n_0} H';\ \eta';\ w\end{array}}{H;\ \eta;\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \Downarrow_{n_0+1} H';\ \eta';\ w} \qquad \text{(E-Let)}$$

$$\frac{H;\ \eta;\ x \Downarrow_{n_1} H'';\ \eta'';\ \mathit{Nil} \quad H'';\ \eta'';\ t_1 \Downarrow_{n_2} H';\ \eta';\ w}{H;\ \eta;\ \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Nil} \Rightarrow t_1;\ \mathit{Cons}\ x_1\ x_2 \Rightarrow t_2\} \Downarrow_{n_1+n_2} H';\ \eta';\ w} \qquad \text{(E-CaseNil)}$$

$$\frac{\begin{array}{c}H;\ \eta;\ x \Downarrow_{n_1} H'';\ \eta'';\ \mathit{Cons}\ x_1''\ x_2'' \\ \eta'' = \eta_1'', x_1 \mapsto \eta_1''(x_1''), x_2 \mapsto \eta_1''(x_2'') \\ H'';\ \eta'';\ t_2 \Downarrow_{n_2} H';\ \eta';\ w\end{array}}{H;\ \eta;\ \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{Nil} \Rightarrow t_1;\ \mathit{Cons}\ x_1\ x_2 \Rightarrow t_2\} \Downarrow_{n_1+n_2} H';\ \eta';\ w} \qquad \text{(E-CaseCons)}$$

Figure 2. Instrumented natural semantics

Evaluation (updates) $\boxed{H;\ \eta;\ t \Downarrow_n H';\ \eta';\ w}$

$$\frac{\begin{array}{c}\eta(x) = h \quad H(h) = (x'@\mathit{Nil};\ \eta') \\ \eta(x') = h' \quad H(h') = (\mathit{Nil}, \eta'') \\ H' = H, h' \mapsto (\mathit{Nil}, \eta'), h \mapsto h'\end{array}}{H;\ \eta;\ x \Downarrow_0 H';\ \eta';\ \mathit{Nil}} \qquad \text{(E-VarNil)}$$

$$\frac{\begin{array}{c}\eta(x) = h \quad H(h) = (x'@(\mathit{Cons}\ x_1'\ x_2');\ \eta') \\ \eta(x') = h' \quad H(h') = (\mathit{Cons}\ x_1''\ x_2'', \eta'') \\ H' = H, h' \mapsto (\mathit{Cons}\ x_1'\ x_2';\ \eta'), h \mapsto h'\end{array}}{H;\ \eta;\ x \Downarrow_0 H';\ \eta';\ \mathit{Cons}\ x_1'\ x_2'} \qquad \text{(E-VarCons)}$$

$$\frac{\begin{array}{c}H';\ \eta;\ x@\mathit{Nil} \Downarrow_0 H'';\ \eta'';\ w'' \\ H'';\ \eta;\ t_1\ x \Downarrow_n H';\ \eta';\ w\end{array}}{H;\ \eta;\ t_1\ x@\mathit{Nil} \Downarrow_n H';\ \eta';\ w} \qquad \text{(E-AppNil)}$$

$$\frac{\begin{array}{c}H';\ \eta;\ x@(\mathit{Cons}\ x_1\ x_2) \Downarrow_0 H'';\ \eta'';\ w'' \\ H'';\ \eta;\ t_1\ x \Downarrow_n H';\ \eta';\ w\end{array}}{H;\ \eta;\ t_1\ x@(\mathit{Cons}\ x_1\ x_2) \Downarrow_n H';\ \eta';\ w} \qquad \text{(E-AppCons)}$$

$$\frac{\begin{array}{c}\eta(x) = h \quad \overline{H}(h) = h' \\ H(h') = (\mathit{Nil};\ \eta'') \quad H' = H, h' \mapsto (\mathit{Nil};\ \eta)\end{array}}{H;\ \eta;\ x@\mathit{Nil} \Downarrow_0 H';\ \eta;\ \mathit{Nil}} \qquad \text{(E-UpdNil)}$$

$$\frac{\begin{array}{c}\eta(x) = h \quad \overline{H}(h) = h' \\ H(h') = (\mathit{Cons}\ x_1''\ x_2'';\ \eta'') \\ H' = H, h' \mapsto (\mathit{Cons}\ x_1\ x_2;\ \eta)\end{array}}{H;\ \eta;\ x@(\mathit{Cons}\ x_1\ x_2) \Downarrow_0 H';\ \eta;\ \mathit{Cons}\ x_1\ x_2} \qquad \text{(E-UpdCons)}$$

Figure 3. Instrumented natural semantics (updates)

The rule E-VarClose operates on variables that map to closures that have no update marker at the top-level of their term component. For these, a fresh heap location is allocated and the original closure is updated with an indirection to the new cell in which the result of evaluating the closure is stored. If a variable is associated with an indirection (rule E-VarInd), the indirection is followed in order to immediately obtain the appropriate weak-head normal form.

For function applications with a variable argument (rule E-AppVar), we first force the function component into a lambda-form and then evaluate the body of the lambda-abstraction with its parameter bound to the argument location.

The rule for local definitions (E-Let) allocates a new heap location and binds it to a closure for the definition. The body of the let-term is then evaluated in an extended environment.

In the rules for case analyses (E-CaseNil and E-CaseCons), the scrutinee is forced to a constructor application, after which evaluation proceeds with the appropriate right-hand side in a suitably extended environment.

The evaluation rules in Figure 3 deal with explicit updates. In the evaluation of variables that are associated with updates (rules E-VarNil and E-VarCons), we perform the desired update and simultaneously place an indirection from the location of the original closure to the updated heap binding.

Function applications to @-forms are handled by rules E-AppNil and E-AppCons. These first perform the associated update and then continue evaluation with the appropriate function application.

Stand-alone updates, as in rules E-UpdNil and E-UpdCons, evaluate to the contained constructor form, but, as side effect, perform the indicated heap rewrite.

4.3 Static Semantics

Essentially, the static semantics of our calculus are a blend of our earlier work on usage analysis [9] and a type-based approach to constructor analysis.

The details of the analyses are presented in Figures 4 and 5, where the judgements of the typing relation take the form

$$\Gamma \vdash t :^{\varphi} \sigma,$$

indicating that within context Γ, the term t can be assigned the type scheme σ and the uniqueness annotation φ.

The typing rules make use of a number of subsidiary judgements. Context splitting, with judgements of the form

$$\Gamma = \Gamma_1 \bowtie \Gamma_2,$$

is used for typing terms that introduce branches in a program's control-flow graph, such as applications and local definitions. The idea is to split up all single-threaded variables that appear in the context of such a term and to distribute them over the contexts that are passed down to its children in the syntax tree. This way, we can guarantee that all updateable values indeed flow single-threadedly.

We employ an entailment relation on qualifiers,

$$\Gamma \Vdash \pi,$$

that basically establishes that our order on uniqueness annotations is indeed reflexive and transitive, and has 1 and ω as, respectively, its least and greatest elements.

The typing rule for variables, T-Var in Figure 4, expresses that for typing variables we simply retrieve the type scheme, uniqueness annotation, and constructor annotation for the variable from the context and forget about the constructor annotation.

In the rule for lambda-abstractions, a subtlety arises. In principle, the uniqueness annotation for an abstraction is, besides by its use in the program, restricted by the annotations for the variables that appear free in its body: these impose a lower bound on the an-

Context splitting $\boxed{\Gamma = \Gamma_1 \bowtie \Gamma_2}$

$$\emptyset = \emptyset \bowtie \emptyset \quad \text{(C-Empty)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi|\psi} \sigma = \Gamma_{11}, x :^{\varphi|\psi} \sigma \bowtie \Gamma_{12} \setminus x} \quad \text{(C-VarOnce1)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi|\psi} \sigma = \Gamma_{11} \setminus x \bowtie \Gamma_{12}, x :^{\varphi|\psi} \sigma} \quad \text{(C-VarOnce2)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\omega|\psi} \sigma = \Gamma_{11}, x :^{\omega|\psi} \sigma \bowtie \Gamma_{12}, x :^{\omega|\psi} \sigma} \quad \text{(C-VarMany)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, \pi = \Gamma_{11}, \pi \bowtie \Gamma_{12}, \pi} \quad \text{(C-Qual)}$$

Entailment $\boxed{\Gamma \Vdash \pi}$

$$\frac{\pi \in \Gamma}{\Gamma \Vdash \pi} \quad \text{(Q-Mono)}$$

$$\Gamma \Vdash \varphi \sqsubseteq \varphi \quad \text{(Q-Refl)}$$

$$\frac{\Gamma \Vdash \varphi_1 \sqsubseteq \varphi_2 \quad \Gamma \Vdash \varphi_2 \sqsubseteq \varphi_3}{\Gamma \Vdash \varphi_1 \sqsubseteq \varphi_3} \quad \text{(Q-Trans)}$$

$$\Gamma \Vdash 1 \sqsubseteq \varphi \quad \text{(Q-Bot)}$$

$$\Gamma \Vdash \varphi \sqsubseteq \omega \quad \text{(Q-Top)}$$

Typing $\boxed{\Gamma \vdash t :^{\varphi} \sigma}$

$$\frac{\Gamma(x) =^{\varphi|\psi} \sigma}{\Gamma \vdash x :^{\varphi} \sigma} \quad \text{(T-Var)}$$

$$\frac{\begin{array}{c}\mathrm{fv}(t_1) - \{x\} = \{x_1, \ldots, x_n\} \\ \left.\begin{array}{c}\Gamma(x_i) =^{\varphi_{x_i}|\psi_{x_i}} \sigma_{x_i} \\ \Gamma \Vdash \varphi_0 \sqsubseteq \varphi_{x_i}\end{array}\right\} \text{for each } i \in \{1, \ldots, n\} \\ \Gamma, x :^{\varphi_1|\epsilon} \tau_1 \vdash t_1 :^{\varphi_2} \tau_2\end{array}}{\Gamma \vdash \lambda x.\, t_1 :^{\varphi} \tau_1^{\varphi_1} \xrightarrow{\varphi_0} \tau_2^{\varphi_2}} \quad \text{(T-Abs)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \quad \Gamma_2 \vdash x :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x :^{\varphi} \tau} \quad \text{(T-AppVar)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \sigma_1 \quad \Gamma_2, x :^{\varphi_1|\epsilon} \sigma_1 \vdash t_2 :^{\varphi} \sigma}{\Gamma \vdash \textbf{let } x = t_1 \textbf{ in } t_2 :^{\varphi} \sigma} \quad \text{(T-Let)}$$

$$\Gamma \vdash \mathit{Nil} :^{1} \mathsf{List}\ \tau_1^{\varphi_1} \quad \text{(T-Nil)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash x_1 :^{\varphi_1} \tau_1 \quad \Gamma_2 \vdash x_2 :^{\varphi} \mathsf{List}\ \tau_1^{\varphi_1} \quad \Gamma \Vdash \varphi \sqsubseteq \varphi_1}{\Gamma \vdash \mathit{Cons}\ x_1\ x_2 :^{\varphi} \mathsf{List}\ \tau_1^{\varphi_1}} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash x :^{\varphi_2} \mathsf{List}\ \tau_1^{\varphi_1} \quad \Gamma, x :^{\varphi_2|\mathit{Nil}} \mathsf{List}\ \tau_1^{\varphi_1} \vdash t_1 :^{\varphi} \sigma \quad \Gamma, x :^{\varphi_2|\mathit{Cons}} \mathsf{List}\ \tau_1^{\varphi_1}, x_1 :^{\varphi_1|\epsilon} \tau_1, x_2 :^{\varphi_2|\epsilon} \mathsf{List}\ \tau_1^{\varphi_1} \vdash t_2 :^{\varphi} \sigma}{\Gamma \vdash \textbf{case } x \textbf{ of } \{\mathit{Nil} \Rightarrow t_1;\ \mathit{Cons}\ x_1\ x_2 \Rightarrow t_2\} :^{\varphi} \sigma} \quad \text{(T-Case)}$$

$$\frac{\Gamma, \pi \vdash t :^{\varphi} \rho_1}{\Gamma \vdash t :^{\varphi} \pi \Rightarrow \rho_1} \quad \text{(T-Qual)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \pi \Rightarrow \rho \quad \Gamma \Vdash \pi}{\Gamma \vdash t :^{\varphi} \rho} \quad \text{(T-Res)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \sigma_1 \quad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash t :^{\varphi} \forall \alpha.\, \sigma_1} \quad \text{(T-TyGen)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall \alpha.\, \sigma_1}{\Gamma \vdash t :^{\varphi} [\alpha \mapsto \tau]\sigma_1} \quad \text{(T-TyInst)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \sigma_1 \quad \beta \notin \mathrm{fav}(\Gamma)}{\Gamma \vdash t :^{\varphi} \forall \beta.\, \sigma_1} \quad \text{(T-EffGen)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall \beta.\, \sigma_1}{\Gamma \vdash t :^{\varphi} [\beta \mapsto \varphi_0]\sigma_1} \quad \text{(T-EffInst)}$$

$$\frac{\Gamma \vdash t :^{\varphi_0} \sigma \quad \Gamma \vdash \varphi_0 \sqsubseteq \varphi}{\Gamma \vdash t :^{\varphi} \sigma} \quad \text{(T-Sub)}$$

Figure 4. Annotated type system

Typing (updates) $\boxed{\Gamma \vdash t :^{\varphi} \sigma}$

$$\frac{\Gamma = \Gamma_1 \bowtie (\Gamma_{21} \bowtie \Gamma_{22}) \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \quad \Gamma_{21}(x) =^{1|\mathit{Nil}} \sigma_0 \quad \Gamma_{22} \vdash \mathit{Nil} :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x@\mathit{Nil} :^{\varphi} \tau} \quad \text{(T-AppNil)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie (\Gamma_{21} \bowtie \Gamma_{22}) \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \quad \Gamma_{21}(x) =^{1|\mathit{Cons}} \sigma_0 \quad \Gamma_{22} \vdash \mathit{Cons}\ x_1\ x_2 :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x@(\mathit{Cons}\ x_1\ x_2) :^{\varphi} \tau} \quad \text{(T-AppCons)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1(x) =^{1|\mathit{Nil}} \sigma_0 \quad \Gamma_2 \vdash \mathit{Nil} :^{\varphi} \sigma}{\Gamma \vdash x@\mathit{Nil} :^{\varphi} \sigma} \quad \text{(T-UpdNil)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1(x) =^{1|\mathit{Cons}} \sigma_0 \quad \Gamma_2 \vdash \mathit{Cons}\ x_1\ x_2 :^{\varphi} \sigma}{\Gamma \vdash x@(\mathit{Cons}\ x_1\ x_2) :^{\varphi} \sigma} \quad \text{(T-UpdCons)}$$

Figure 5. Annotated type system (updates)

Store typing $\boxed{H;\ \eta \vdash \Gamma}$

$$\frac{}{H;\ \emptyset \vdash \emptyset} \quad \text{(S-EMPTY)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma_1 \quad H(h) = (\mathit{Nil};\ \eta') \quad H;\ \eta' \vdash \Gamma' \quad \Gamma' \vdash \mathit{Nil} :^{\varphi} \sigma}{H;\ \eta_1, x \mapsto h \vdash \Gamma_1, x :^{\varphi|\mathit{Nil}} \sigma} \quad \text{(S-NIL)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma_1 \quad H(h) = (\mathit{Cons}\ x_1\ x_2;\ \eta') \quad H;\ \eta' \vdash \Gamma' \quad \Gamma' \vdash \mathit{Cons}\ x_1\ x_2 :^{\varphi} \sigma}{H;\ \eta_1, x \mapsto h \vdash \Gamma_1, x :^{\varphi|\mathit{Cons}} \sigma} \quad \text{(S-CONS)}$$

$$\frac{H;\ \eta_1 \vdash \Gamma_1 \quad H(h) = (t;\ \eta') \quad H;\ \eta' \vdash \Gamma' \quad \Gamma' \vdash t :^{\varphi} \sigma}{H;\ \eta_1, x \mapsto h \vdash \Gamma_1, x :^{\varphi|\epsilon} \sigma} \quad \text{(S-CLOSE)}$$

$$\frac{H(h) = h' \quad H;\ \eta_1, x \mapsto h' \vdash \Gamma}{H;\ \eta_1, x \mapsto h \vdash \Gamma} \quad \text{(S-IND)}$$

Figure 6. Store typing

notation for the abstraction. This is an instance of a more general scheme, that is referred to as the *containment restriction*: the annotation for a structure containing elements should not be greater than the annotation for any of its elements. It turns out that the containment restriction is essential to maintaining referential transparency [1]. However, as we see in a moment, our system admits *subeffecting* (cf. Hage et al. [9]) which basically allows uniqueness annotations on typings to be enlarged in an arbitrarily fashion. Without countermeasures this would allow bypassing the containment restriction and, effectively, destroying referential transparency. The countermeasure we adopt here is due to De Vries et al. [7]: we place an extra annotation φ_0 on the function-space constructor that stores an upper bound on the annotation for the lambda-abstraction. Now, whenever a function is applied, as in rule T-APPVAR, we make sure that the annotation for the function adheres to the upper bound and, if so, proceed as usual.

In both rule T-ABS and rule T-LET, we set the constructor annotation for new context bindings to ϵ, indicating that, at that point, we do not have enough information to uniquely determine the involved constructor.

The rules T-NIL and T-CONS deal with list construction. The invocation of qualifier entailment in T-CONS is once again necessary to meet the containment restriction.

In the rule for case analyses, the most important observation to make is that the contexts that are passed down to the terms at the right-hand sides of case-terms are refined to contain a more informative constructor annotation for the scrutinee.

The rules T-QUAL and T-RES deal with introduction and elimination of qualifiers. Generalization and instantiation is handled by the rules T-TYGEN, T-TYINST, T-EFFGEN, and T-EFFINST. Rule T-SUB introduces subeffecting.

The typing of updates is defined by rules T-APPNIL, T-APPCONS, T-UPDNIL, and T-UPDCONS in Figure 5. These rules ensure that the value to be updated is guaranteed to be used single-threadedly and is built with the right constructor.

4.4 Properties

Our static semantics is a conservative extension of the standard Hindley-Milner type system [17], equipped with a principal-type property. It admits a fully automatic type-reconstruction algorithm, that can formulated as an extension of Algorithm W [6].

Soundness with respect to the natural semantics is established by means of a subject-reduction result. To formulate this result, we define a relation between stores (consisting of a heap and an environment) and typing contexts. The relation is depicted in Figure 6 and has judgements of the form

$$H;\ \eta \vdash \Gamma.$$

Using this relation, our soundness result now reads:

Theorem 1 (Subject Reduction). If $H;\ \eta;\ t \Downarrow_n H';\ \eta';\ w$ with $H;\ \eta \vdash \Gamma$ and $\Gamma \vdash t :^{\varphi} \sigma$, then there exists a context Γ' such that $H';\ \eta' \vdash \Gamma'$ and $\Gamma' \vdash w :^{\varphi} \sigma$. ∎

While soundness is of course a crucial property of our static semantics, we also want to have made sure that adding validly placed update markers does indeed improve the space efficiency of terms. To this end, we first define the erasure $\lfloor t \rfloor$ of a term t. Erasure amounts to removing all update markers from a term, introducing fresh let-bindings to yield syntactically valid forms:

$$\begin{array}{ll}
\lfloor x \rfloor & = x \\
\lfloor \lambda x.\ t_1 \rfloor & = \lambda x.\ \lfloor t_1 \rfloor \\
\lfloor t_1\ x \rfloor & = \lfloor t_1 \rfloor\ x \\
\lfloor \textbf{let } x = t_1 \textbf{ in } t_2 \rfloor & = \textbf{let } x = \lfloor t_1 \rfloor \textbf{ in } \lfloor t_2 \rfloor \\
\lfloor \mathit{Nil} \rfloor & = \mathit{Nil} \\
\lfloor \mathit{Cons}\ x_1\ x_2 \rfloor & = \mathit{Cons}\ x_1\ x_2 \\
\lfloor \textbf{case } x \textbf{ of } \{\mathit{Nil} \Rightarrow t_1;\ \mathit{Cons}\ x_1\ x_2 \Rightarrow t_2\} \rfloor & \\
\quad = \textbf{case } x \textbf{ of } \{\mathit{Nil} \Rightarrow \lfloor t_1 \rfloor;\ \mathit{Cons}\ x_1\ x_2 \Rightarrow \lfloor t_2 \rfloor\} & \\
\lfloor t_1\ x@\mathit{Nil} \rfloor & = \textbf{let } x_0 = \mathit{Nil} \textbf{ in } \lfloor t_1 \rfloor\ x_0 \\
 & \quad \text{where } x_0 \text{ is fresh} \\
\lfloor t_1\ x@(\mathit{Cons}\ x_1\ x_2) \rfloor & = \textbf{let } x_0 = \mathit{Cons}\ x_1\ x_2 \textbf{ in } \lfloor t_1 \rfloor\ x_0 \\
 & \quad \text{where } x_0 \text{ is fresh} \\
\lfloor x@\mathit{Nil} \rfloor & = \mathit{Nil} \\
\lfloor x@(\mathit{Cons}\ x_1\ x_2) \rfloor & = \mathit{Cons}\ x_1\ x_2
\end{array}$$

Erasure extends naturally to terms contained in heap closures and we write $\lfloor H \rfloor$ for the heap that is obtained by applying erasure to all closures in a heap H.

The following property of our system now expresses that validly adding update markers to a markerless program does not change the meaning of a program and, moreover, does not have a negative impact on the program's space behaviour:

Theorem 2. If $H;\ \eta;\ t \Downarrow_{n_1} H';\ \eta';\ w$ with $H;\ \eta \vdash \Gamma$ and $\Gamma \vdash t :^{\varphi} \sigma$, then there exist H'', η'', and n_2 with $\lfloor H \rfloor;\ \eta;\ \lfloor t \rfloor \Downarrow_{n_2} H'';\ \eta'';\ w$. Moreover, for all such H'', η'', and n_2, we have that $n_1 \leqslant n_2$. ∎

5. Further Exploration

So far, we have focussed on a single fixed approach within the design space around our construct. There are, however, some obvious directions for further investigation.

First of all, the reader may ask why we let the compiler check whether update markers are placed correctly by the programmer rather than take an unmarked source program and add update markers automatically. The answer is that we want to enable the programmer to use his knowledge of the program and express his intuitions about possible optimisations. The checks performed by the compiler then are a means of reassurance. If we were to rely on automatic optimizations exclusively, we would be in awkward position if it turned out that the compiler, for some reason, was not able to perform one or more of the optimizations we were expecting. The only thing we could do then, is to try and find out whether the optimizations could not be performed and rewrite the program in such a way that the compiler can actually optimize it according to our expectations, perhaps abandoning an idiomatic style of programming. In contrast, in our approach, the programmer could just leave the style of the program unchanged and add a marker to trigger the optimization. So, we do regard our approach as an addi-

tion to rather than a substitute for a system that decides on in-place updating fully automatically.

Still, from a usability point of view, there are some issues that need to be resolved. For instance, the type language of our calculus is far too complex to be exposed to the programmer. Instead, we envision that annotations in types are completely hidden from the surface of the language. The programming environment then needs to be equipped with other, demand-driven means to inform the programmer about the uniqueness properties of her program.

Another point of concern in our system is the need for duplicating the definition of general-purpose functions such as *reverse* in Section 2.1. A possible alleviation would be to have the unmarked version of such a function be generated from the marked function and to come up with syntax that allows the programmer to distinguish between calls to the two versions of the function.

In order to further investigate the design space, we would be in favour of experimenting with our approach by incorporating it in a real compiler for a large-scale programming language like Haskell.

6. Related Work

The Clean language [19] uses uniqueness typing [1] to maintain referential transparency in the presence of side effects. Uniqueness types are inferred as well as checked. The Clean compiler can, for arbitrary algebraic values, use the results of uniqueness analysis to emit code that performs destructive updates, but it only does so within the context of a fully automatic program optimization. In particular, Clean does not enable the programmer to explicitly write down assignments. Still, the optimization has proven to be quite effective, showing a dramatic, positive impact on both memory and time consumption of programs. Interestingly, our approach is complementary to Clean's: in our system uniqueness types are not visible to the programmer, but recycling is—while in Clean this is just the other way around.

Other techniques for automatically inserting destructive assignments into functional programs are proposed by Jones and Le Métayer [14], and by Jensen and Mogensen [12]. Contrary to the approach adopted in Clean and in the present paper, these analyses are not type-based and instead formulated as abstract interpretations of source programs. A more ad-hoc technique, targeted at logic languages rather than functional languages, is given by Gudjónsson and Winsborough [8].

Hofmann [10] shows how a linear typing discipline can be used to keep track of sites at which functional programs can perform in-place updates. Again, this analysis, which results in `malloc`-free C-code, is completely shielded from the programmer. Furthermore, the analysis is restricted to first-order programs, whereas ours is also applicable to higher-order languages.

Hofmann and Jost [11] present an analysis that is capable of statically establishing upper bounds for the memory usage of first-order functional programs. Such an analysis seems particulary useful within the context of embedded system that only have limited resources. We expect that Hofmann and Jost's analysis can easily be adapted to cope with explicit update markers. Moreover, it would be interesting to see whether our approach to uniqueness analysis can be of any help in extending their analysis to higher-order programs.

7. Conclusion

We have presented a type-based approach to guaranteeing referential transparency in the presense of programmer-controlled destructive updates, that can be incorporated in compilers for lazy functional languages. We believe that providing the programmer with a lightweight means of tweaking programs for optimization is a welcome addition to fully automatic optimizations that deserves further exploration.

Acknowledgments

The authors would like to thank Jeroen Fokker, Alexey Rodriquez, and the anonymous referees for their helpful comments. This work was supported by the Netherlands Organization for Scientific Research through its project on "Scriptable Compilers" (612.063.406).

References

[1] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 13th Conference, Bombay, India, December 15–17, 1993, Proceedings*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer-Verlag, 1993.

[2] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, London, 2nd edition, 1998.

[3] Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology and Göteborg University, 1999.

[4] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In Werner E. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, IFL'96, Bad Godesberg, Germany, September 16–18, 1996, Selected Papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 58–84. Springer-Verlag, 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2nd edition, 2001.

[6] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.

[7] Edsko de Vries, Rinus Plasmeijer, and David Abrahamson. Uniqueness typing redefined. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages, 18th International Workshop, IFL 2006, Budapest, Hungary, September 4–6, 2006, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 181–198. Springer-Verlag, 2007.

[8] Gudjón Gudjónsson and William H. Winsborough. Compile-time memory reuse in logic programming languages through update in place. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):430–501, 1999.

[9] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007*, pages 235–246. ACM Press, 2007.

[10] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[11] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 15–17, 2003*, pages 185–197. ACM Press, 2003.

[12] Thomas P. Jensen and Torben Æ. Mogensen. A backwards analysis for compile-time garbage collection. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15–18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 227–239. Springer-Verlag, 1990.

[13] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, 1994.

[14] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *FPCA '89 Conference on Functional Programming and Computer Architecture. Imperial College, London, England, 11–13 September 1989*, pages 54–74. ACM Press, 1989.

[15] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 144–154. ACM Press, 1993.

[16] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[18] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

[19] Rinus Plasmeijer and Marco van Eekelen. Concurrent Clean language report—version 1.3. Technical Report CSI-R9816, University of Nijmegen, 1998.

[20] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24–30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.