Heap Recycling for Lazy Languages

Stefan Holdermans
(Joint work with Jurriaan Hage)

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: stefan@cs.uu.nl
Web pages: http://people.cs.uu.nl/stefan/

January 8, 2008
PEPM 2008
Lazy languages better be pure

Functional languages can be classified along several axes:

- pure vs. impure;
- strict (eager) vs. nonstrict (lazy).

Not all combinations make sense: reasoning about side-effects in a nonstrict context is hard.
Referential transparency

- Pure languages are *referential transparent*: each term can always be safely replaced by its value.
- Referential transparency enables equational reasoning.
- Referential transparency enables memoization, common subexpression elimination, parallel evaluation strategies, etc.

 Referential transparency follows directly from purity.
Monads can do the job

- Referential transparency requires us to either ban side-effects or deal with them in some special way.
- Example: monadic encapsulation of side-effects in Haskell.

\[
\begin{align*}
main &:: IO () \\
main &= do \ input &\leftarrow \ readFile \ "in" \\
&\quad \ ByteArrayInputStream \ "out" \ (reverse \ input)
\end{align*}
\]

Monads come with their own programming style.

Reasoning about monadic code can be hard.
Don’t overdo

- Combining the monadic and “ordinary” functional style is okay if side-effects are fundamental to the program.
- If side-effects are only peripheral, a purely functional look and feel is preferred.
- Example: use of an I/O monad makes sense for programs that are indeed about I/O, but not for the occasional debug statement.

```haskell
revSort :: [Int] → [Int]
revSort = (trace "applying revSort") (reverse ∘ sort)
```

- Similarly, monadic in-place updates make sense for the union-find algorithm, but not for the occasional performance tweak.
Idiomatic list reversal allocates runs in linear space:

\[
\text{reverse} :: [a] \rightarrow [a] \\
\text{reverse} \ l \ = \ \text{rev} \ l \ [] \\
\text{where} \\
\text{rev} \ [] \ acc = acc \\
\text{rev} \ (x : xs) \ acc = \text{rev} \ xs \ (x : acc)
\]

\begin{itemize}
  \item \(\text{rev}\) constructs a new heap cell for every node in the input.
\end{itemize}

If the input list is used \textit{only once}, we would like to \textit{reuse} its cons-nodes and only use constant space.
Monadic in-place list reversal

In-place list reversal can be implemented with lazy state threads (Lauchbury and Peyton Jones, PLDI’94):

```haskell
type STList s a = STRef s (L s a)
data L s a = STNil | STCons (STRef s a) (STRef s (STList s a))
reverse' :: STList s a → ST s (STList s a)
reverse' r = do acc ← newSTRef STNil
  rev r acc
  where
    rev r acc = do l ← readSTRef r
      case l of
        STNil → return acc
        STCons hd tl → do r' ← readSTRef tl
                          writeSTRef tl acc
                          rev r' r
```

⚠️ A lot of work for a simple performance tweak!
We propose a small language extension:

\[
\begin{align*}
\text{reverse}'' & : [a] \to [a] \\
\text{reverse}'' & l = \text{rev} l [] \\
\text{where} & \\
\text{rev} & [] \quad \text{acc} = \text{acc} \\
\text{rev} & l@(x : xs) \quad \text{acc} = \text{rev} xs l@(x : \text{acc})
\end{align*}
\]

We allow the \texttt{@}-construct not only at the left-hand side of a function definition, but also at the right-hand side, where it denotes explicit reuse of a heap node.
Challenges

Q How do we ensure that in-place updates do not compromise referential transparency?

Q How do we ensure that in-place updates make sense with respect to the underlying memory model?

A We put statically enforced restrictions on the contexts in which updates occur.
Referential transparency at stake

In-place filter:

\[
\text{filter}' :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

\[
\text{filter}' \quad p \quad [] = []
\]

\[
\text{filter}' \quad p \quad l@ (x : xs) = \text{if} \quad p \ x \quad \text{then} \quad l@ (x : \text{filter}' \quad p \quad xs) \quad \text{else} \quad \text{filter}' \quad p \quad xs
\]

Putting odd numbers before even numbers:

\[
\text{let} \quad l = [1 .. 10]
\]

\[
\text{in} \quad \text{filter}' \quad \text{odd} \quad l \ + \ \text{filter}' \quad \text{even} \quad l
\]

Yields \([1, 3, 5, 7, 9]\)! What happened to \([2, 4, 6, 8, 10]\)?
Keeping track of single-threadedness

- We only allow in-place updates of values that are passed around single-threadedly.
- Single-threadedness in enforced through type-based uniqueness analysis.
- We annotate typing judgements with uniqueness annotations \( \varphi: 1 \) for single-threaded terms, \( \omega \) for multi-threaded terms (with \( 1 \sqsubseteq \omega \)).
- For example: \( l ::^1 \ [Int^\omega] \) indicates that the list \( l \) is passed around single-threadedly, but its elements may be used multi-threadedly.
Uniqueness analysis for in-place filter

Possible analysis for \( \text{filter}' \text{ even} \):

\[
\text{filter}' \text{ even} : \omega_1 \left[ \text{Int}^\omega_2 \right]_3 \rightarrow \omega_4 \left[ \text{Int}^\omega_5 \right]_6
\]

1. The filter may be passed around multi-threadedly.
2. The elements of the argument list may be passed around multi-threadedly.
3. The argument list must be passed around single-threadedly!
4. The filter is not subjected to any containment restriction (see paper).
5. The elements of the result list may be passed around multi-threadedly.
6. The result list may be passed around multi-threadedly.
Judgements for uniqueness analysis

The typing rules for uniqueness analysis are of the form \( \Gamma \vdash t :: \varphi \sigma \), where \( \sigma \) can contain annotations.

\[
\Gamma = \Gamma_1 \otimes \Gamma_2 \quad \Gamma_1 \vdash t_1 :: \varphi_1 \tau_2 \varphi_2 \rightarrow \varphi_0 \tau \varphi \\
\Gamma \vdash \varphi_1 \subseteq \varphi_0 \quad \Gamma_2 \vdash t_2 :: \varphi_2 \tau_2 \\
\frac{}{\Gamma \vdash t_1 \ t_2 :: \varphi \tau}
\]

The auxiliary judgement \( \Gamma = \Gamma_1 \otimes \Gamma_2 \) ensures that single-threaded variables are not passed down to multiple subterms.

\( \Gamma \vdash \varphi_1 \subseteq \varphi_0 \) enforces a containment restriction.

The analysis allows for both type polymorphism and uniqueness polymorphism (cf. Hage et al., ICFP 2007).
Fitting the memory model

- Often, a language specification does not prescribe a particular memory model: so, we only allow updates that are likely to be implementable in all implementations of lazy languages.
- For example: replacing a nil-cell by a cons-cell will in most cases be problematic and should therefore be prohibited.
- The scheme we adopt only allows updates with values built by the same constructor.
- To keep track the constructors values are built by, we store them in the typing context $\Gamma$ in bindings of the form $x ::= \varphi | \psi \sigma$, where $\psi$ is either a constructor $C$ or $\epsilon$. 
Rule for updates

Both aspects (referential transparency and the memory model) show up in the typing rule for in-place updates:

\[
\Gamma = \Gamma_1 \otimes \Gamma_2 \\
\Gamma_1(x) =^1|C \quad \sigma_0 \quad \Gamma_2 \vdash C \ t_1 \ldots \ t_n :: \varphi \quad \sigma \\
\Gamma \vdash x@ (C \ t_1 \ldots \ t_n) :: \varphi \quad \sigma
\]

- $x$ is required to be passed around single-threadedly.
- $x$ is required to be built by $C$. 

[Faculty of Science Information and Computing Sciences]
Using an instrumented natural semantics, with judgements of the form

\[ H; \eta; t \downarrow_n H'; \eta'; w \]

(with \( H \) a heap, \( \eta \) a mapping from variables to heap locations, \( w \) a weak-head normal form, and \( n \) the number of heap cells allocated), we can demonstrate a subject-reduction result.

Furthermore, we can show that adding well-behaved updates to a program preserves the meaning of the original program and the new program requires at most the same amount of space.
Assessment

- Should update instructions be inferred?
- Do we need two versions of `reverse`? Do we need two versions of `filter`? What about `zip`?
- Do we expose annotated types to the programmer?
- How does our system relate to Clean?