Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

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.

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathbf{do}\ \mathit{input} \leftarrow \mathit{readFile}\ \texttt{"in"} \\
&\qquad\qquad\quad \mathit{writeFile}\ \texttt{"out"}\ (\mathit{reverse}\ \mathit{input})
\end{aligned}
$$

☞ 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.

$$\begin{aligned}&\mathit{revSort} :: [\mathit{Int}] \to [\mathit{Int}]\\&\mathit{revSort} = (\mathit{trace}\ \texttt{"applying revSort"})\ (\mathit{reverse} \circ \mathit{sort})\end{aligned}$$

- Similarly, monadic in-place updates make sense for the union-find algorithm, but not for the occasional performance tweak.

Idiomatic list reversal

Idiomatic list reversal allocates runs in linear space:

$$
\begin{array}{l}
\mathit{reverse} :: [a] \to [a] \\
\mathit{reverse}\quad l \quad = \mathit{rev}\ l\ [\,] \\
\quad \textbf{where} \\
\qquad \mathit{rev}\ [\,] \qquad\quad \mathit{acc} = \mathit{acc} \\
\qquad \mathit{rev}\ (x : \mathit{xs})\ \mathit{acc} = \mathit{rev}\ \mathit{xs}\ (x : \mathit{acc})
\end{array}
$$

☞ *rev* constructs a new heap cell for every node in the input.

If the input list is used only once, we would like to 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):

```
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!

Idiomatic in-place list reversal

We propose a small langauge extension:

$$
\begin{array}{l}
\mathit{reverse}'' :: [a] \rightarrow [a] \\
\mathit{reverse}'' \quad l \quad = \mathit{rev}\ l\ [\,] \\
\quad \textbf{where} \\
\qquad \mathit{rev}\ [\,] \qquad\qquad \mathit{acc} = \mathit{acc} \\
\qquad \mathit{rev}\ l@(x : xs)\ \mathit{acc} = \mathit{rev}\ xs\ l@(x : \mathit{acc})
\end{array}
$$

☞ We allow the @-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:

$$
\begin{array}{llll}
\mathit{filter}' :: (a \rightarrow \mathit{Bool}) \rightarrow [a] & & \rightarrow & [a] \\
\mathit{filter}' \quad p & [\,] & = & [\,] \\
\mathit{filter}' \quad p & l@(x : xs) & = & \mathbf{if} \quad p\ x \\
 & & & \mathbf{then}\ l@(x : \mathit{filter}'\ p\ xs) \\
 & & & \mathbf{else}\ \ \mathit{filter}'\ p\ xs
\end{array}
$$

Putting odd numbers before even numbers:

$$
\begin{array}{l}
\mathbf{let}\ l = [1\,..\,10] \\
\mathbf{in}\ \ \mathit{filter}'\ \mathit{odd}\ l + \! + \mathit{filter}'\ \mathit{even}\ l
\end{array}
$$

☞ 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 φ: 1 for single-threaded terms, ω 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 $\mathit{filter}'\ \mathit{even}$:

$$\mathit{filter}'\ \mathit{even}\ ::^{\omega}_{\boxed{1}}\ [\mathit{Int}^{\omega}_{\boxed{2}}]^{1}_{\boxed{3}} \rightarrow^{\omega}_{\boxed{4}}\ [\mathit{Int}^{\omega}_{\boxed{5}}]^{\omega}_{\boxed{6}}$$

- $\boxed{1}$ The filter may be passed around multi-threadedly.
- $\boxed{2}$ The elements of the argument list may be passed around multi-threadedly.
- $\boxed{3}$ The argument list must be passed around single-threadedly!
- $\boxed{4}$ The filter is not subjected to any containment restriction (see paper).
- $\boxed{5}$ The elements of the result list may be passed around multi-threadedly.
- $\boxed{6}$ 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 σ can contain annotations.

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \qquad \Gamma_1 \vdash t_1 \;::^{\varphi_1}\; {\tau_2}^{\varphi_2} \to^{\varphi_0} \tau^{\varphi} \qquad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \qquad \Gamma_2 \vdash t_2 \;::^{\varphi_2}\; \tau_2}{\Gamma \vdash t_1\ t_2 \;::^{\varphi}\; \tau}$$

- The auxiliary judgement $\Gamma = \Gamma_1 \bowtie \Gamma_2$ ensures that single-threaded variables are not passed down to multiple subterms.
- $\Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0$ 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 langauges.
- 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 Γ in bindings of the form $x\ ::^{\varphi|\psi}\ \sigma$, where ψ is either a constructor C or ϵ.

Rule for updates

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

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \qquad \Gamma_1(x) \;=^{1|C}\; \sigma_0 \qquad \Gamma_2 \vdash C\ t_1 \ldots t_n \;::^{\varphi}\; \sigma}{\Gamma \vdash x@(C\ t_1 \ldots t_n) \;::^{\varphi}\; \sigma}$$

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

Properties

Using an instrumented natural semantics, with judgements of the form

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

(with H a heap, η 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?