Typing in-place update

David Aspinall

LFCS
Edinburgh

Martin Hofmann

Institut für Informatik
Munich

Motivation and background

- Goal: use in-place update rather than fresh creation of memory cells and GC *when it's safe*. "Safe" means to implement the functional semantics.
- Examples:
  — implement list append by altering first list, but ensure result is indistinguishable from a functional append.
  — implement array update as in-place update but ensure result is indistinguishable from a functional update: `set:array,int,val -> array`.
- Background: languages & type systems capturing complexity classes (Hofmann).
- Possible applications: embedded systems, smartcards, HDLs.

Programming with diamonds

- **LFPL** [MH, ESOP 2000] is prototypical first-order linear functional programming language with recursively defined functions and the following types:

$$A \quad ::= \quad \mathsf{N} \quad | \quad \Diamond \quad | \quad \mathsf{L}(A) \quad | \quad \mathsf{T}(A) \quad | \quad A_1 \otimes A_2$$

  The *diamond type* $\Diamond$ stands for a unit of heap space.

- Diamonds give the programmer control over heap space in an abstract and type-safe way.

- Many standard examples can be typed in LFPL.

Diamond trading

```
def list reverse(list l) = reverse_aux(l, nil)

def list reverse_aux(list l,list acc) =
      match l with
              nil -> acc
            | cons(d,h,t) -> reverse_aux(t,cons(d,h,acc))
```

- The first argument to cons has type ◊.
- Computing with *bounded heap space*: the only way to obtain a ◊ is by pattern matching.
- Can easily add `malloc:() -> ◊` and `free:◊ -> ()`.

Imperative operational semantics

- LFPL is executed imperatively, using in-place update.
- Simple compilers have been written which translate to imperative languages: C, Java, JVML, and HBAL.
- More abstractly, we can give a stack-based operational semantics which updates a heap.

$$S, \sigma \vdash e \leadsto v, \sigma'$$

$S : \mathsf{Var} \to \mathsf{SVal}$	stack
$v : \mathsf{SVal}$	stack value: integer, location, NULL, or tuple thereof
$\sigma : \mathsf{Loc} \to \mathsf{HVal}$	heap
$h : \mathsf{HVal}$	heap value: stack value or record $\{\mathsf{id}_1 = v_1 \ldots \mathsf{id}_n = v_n\}$

- Diamond arguments evaluate to heap locations:

$$\frac{S,\sigma \vdash e_d \rightsquigarrow l_d,\sigma' \qquad S,\sigma' \vdash e_h \rightsquigarrow v_h,\sigma'' \qquad S,\sigma'' \vdash e_t \rightsquigarrow v_t,\sigma'''}{S,\sigma \vdash \texttt{cons}(e_d,e_h,e_t) \rightsquigarrow l_d,\sigma'''[l_d \mapsto \{\mathsf{hd}{=}v_h,\mathsf{tl}=v_t\}]}$$

$$\frac{\begin{array}{c} S,\sigma \vdash e \rightsquigarrow l,\sigma' \qquad \sigma'(l) = \{\mathsf{hd}{=}v_h,\mathsf{tl}{=}v_t\} \\ S[x_d \mapsto l, x_h \mapsto v_h, x_t \mapsto v_t],\sigma' \vdash e_c \rightsquigarrow v,\sigma'' \end{array}}{S,\sigma \vdash \texttt{match}\ e\ \texttt{with nil} \Rightarrow e_n \ \mid\ \texttt{cons}(x_d,x_h,x_t) \Rightarrow e_c \rightsquigarrow v,\sigma''}$$

- The typing rules must ensure type safety, and that the operational (in-place update) interpretation agrees with the set-theoretic (functional) interpretation.

- In LFPL, *linearity for heap-types* ensures this agreement. But this is overly conservative…

A drawback of LFPL

```
def sumdigits(l) =
      match l with
            nil -> 0
          | cons(d,h,t) -> h + (10 * sumdigits(t))
```

After evaluating `sumdigits(l)`, list `l` is considered destroyed. We can avoid this by reconstructing the argument:

```
def sumdigits'(l) =
      match l with
            nil -> (nil,0)
          | cons(d,h,t) -> let (t',n) = sumdigits'(t)
                           in (cons(d,h,t'), h + (10 * n))
```

But this is tedious and inefficient; we would rather relax linearity for calls to `sumdigits`, since it is quite safe to do so.

Relaxing linearity for heap data

- We want to express that `sumdigits` operates in a **read-only** fashion on its argument. Moreover, it returns a result which no longer refers to the list. So

  ```
  cons(d,sumdigits(l),reverse(l))
  ```

  is correctly evaluated, assuming left-to-right eval order.

- Other functions are read-only, but give a result which shares with the argument, e.g., **`nth_tail(n,l)`**. But now

  ```
  cons(d,nth_tail(2,l),cons(d',reverse(l),nil))
  ```

  is *not* soundly evaluated by the imperative op sems. If `l=[1,2,3]`, we get `[[1],[3,2,1]]`, not `[[3],[3,2,1]]`. Later uses of `l` should only be allowed if they are also non-destructive.

Usage aspects

- The op. sems and examples motivate *usage aspects* for sub-expressions:

1	Destructive	e.g., `l` in `reverse(l)`
2	Non-destructive but shared	e.g., `l` in `append(k,l)`
3	Non-destructive, not shared	e.g., `l` in `sumdigits(l)`

- Aspects express relationship between heap region of arguments of a function and the heap region of its result.
- Our aspects are novel AFAWK, but related to some previous analyses of linear type systems.
  Wadler: *sequential let*. Odersky: *observer annotations* (cf.2).
  Kobayashi: δ-*annotations* (cf.3).

An improved LFPL

- We track usage aspects of variables in the context. Each variable is annotated with an aspect $i \in \{1, 2, 3\}$:

$$x_1 \overset{i_1}{:} A_1, \ldots, x_n \overset{i_n}{:} A_n \vdash e : A$$

- Each argument of a function is annotated:

$$\begin{array}{lcl}
\texttt{+, -} & : & \mathsf{N}^3, \mathsf{N}^3 \to \mathsf{N} \\
\texttt{nil}_A & : & \mathsf{L}(A) \\
\texttt{cons}_A & : & \Diamond^1, A^2, \mathsf{L}(A)^2 \to \mathsf{L}(A)
\end{array}$$

- Function applications and other expressions are restricted to variables to track aspects. The **let rule** combines contexts, and assumes an evaluation order.

Variable typing rules

$$\frac{}{x \overset{2}{:} A \vdash x : A} \qquad (\quad)$$

$$\frac{\Gamma, x \overset{i}{:} A \vdash e : B \qquad j \leq i}{\Gamma, x \overset{j}{:} A \vdash e : B} \qquad (\quad)$$

$$\frac{\Gamma \vdash e : A \qquad A \text{ heap-free (no } \Diamond, \mathsf{L}(A), \mathsf{T}(A))}{\Gamma^3 \vdash e : A} \qquad (\quad)$$

Γ^i means Γ with any 2-aspect $x_k \overset{2}{:} A_k$ replaced by $x_k \overset{i}{:} A_k$.

List typing rules

$$\frac{}{\vdash \texttt{nil}_A : \mathsf{L}(A)}$$

$$\frac{}{x_d \overset{1}{:} \lozenge, x_h \overset{2}{:} A, x_t \overset{2}{:} \mathsf{L}(A) \vdash \texttt{cons}_A(x_d, x_h, x_t) : \mathsf{L}(A)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_n : B \\ \Gamma, x_d \overset{i_d}{:} \lozenge, x_h \overset{i_h}{:} A, x_t \overset{i_t}{:} \mathsf{L}(A) \vdash e_c : B \qquad i = \min(i_d, i_h, i_t)\end{array}}{\Gamma, x \overset{i}{:} \mathsf{L}(A) \vdash \texttt{match}\ x\ \texttt{with nil => } e_n \ \ \texttt{|} \ \ \texttt{cons}(x_d, x_h, x_t) \texttt{ => } e_c : B}$$

The let rule

$$\frac{S, \sigma \vdash e_a \rightsquigarrow v, \sigma' \qquad S[x \mapsto v], \sigma' \vdash e_b \rightsquigarrow v', \sigma''}{S, \sigma \vdash \texttt{let}\ x = e_a\ \texttt{in}\ e_b \rightsquigarrow v', \sigma''}$$

$$\frac{\Gamma, \Delta_a \vdash e_a : A \qquad \Delta_b, \Theta, x \overset{i}{:} A \vdash e_b : B \qquad \text{side condition}}{\Gamma^i, \Theta, \Delta_a^i \wedge \Delta_b \vdash \texttt{let}\ x = e_a\ \texttt{in}\ e_b : B}$$

Side condition prevents common variables $z \in \mathrm{dom}(\Delta_a) = \mathrm{dom}(\Delta_b)$ being modified before being referenced and prevents "internal" sharing in heap regions reachable from the stack.

A contraction rule for aspect 3 variables is derivable.

Correctness proof

- Aim: prove that operational semantics agrees with denotational semantics (soundness and adequacy).
- Denotational sems $[\![e]\!]_\eta$ is usual set-theoretic semantics. Interpret $\Diamond$ as a unit type, ignore `d` in `cons(d,h,t)`.

1. Define **heap region** $R_A(v,\sigma)$ associated to value v at type A:
   — $R_{\mathsf{N}}(n,\sigma) = \emptyset$.
   — $R_{\Diamond}(l,\sigma) = \{l\}$.
   — $R_{\mathsf{L}(A)}(\mathsf{NULL},\sigma) = \emptyset$.
   — $R_{\mathsf{L}(A)}(l,\sigma) = \{l\} \cup R_A(h,\sigma) \cup R_{\mathsf{L}(A)}(t,\sigma)$
   when $\sigma(l) = \{\mathsf{hd} = h, \mathsf{tl} = t\}$.

2. Define relation $v \Vdash^{\sigma}_{A,i} a$ to connect **meaningful stack values** v (to be used at aspect $i \leq 2$) to semantic values.

   — $n \Vdash^{\sigma}_{\mathsf{N},i} n'$, if $n = n'$.

   — $l \Vdash^{\sigma}_{\diamondsuit,i} 0$, if $l \in \mathrm{dom}(\sigma)$.

   — $\mathsf{NULL} \Vdash^{\sigma}_{\mathsf{L}(A),i} \mathtt{nil}$.

   — $l \Vdash^{\sigma}_{\mathsf{L}(A),i} \mathtt{cons}(h,t)$,
   if $\sigma(l) = \{\mathsf{hd}{=}v_h, \mathsf{tl} = v_t\}$, $l \Vdash^{\sigma}_{\diamondsuit,i} 0$, $v_h \Vdash^{\sigma}_{A,i} h$, $v_t \Vdash^{\sigma}_{\mathsf{L}(A),i} t$.
   Additionally, $R_{\diamondsuit}(l,\sigma)$, $R_A(v_h,\sigma)$, $R_{\mathsf{L}(A)}(v_t,\sigma)$ are pairwise disjoint in case $i = 1$.

3. Prove that for a typable expression $\Gamma \vdash e : C$,

$$S, \sigma \vdash e \leadsto v, \sigma' \qquad \text{iff} \qquad [\![e]\!]_\eta \Downarrow \ \text{ and } \ v \Vdash^{\sigma}_{C,i} [\![e]\!]_\eta$$

   for $i = 2$ and (with condition on η), for 1. Moreover, regions in σ' relate to those in σ as expected by aspects in Γ.

Further details

- Paper gives full typing rules. Also discusses sharing in data-structures, and both $\otimes$ and $\times$ products.
- Home page: `http://www.dcs.ed.ac.uk/home/resbnd`
- Experimental compilers available on our web pages:

target	*features*	*author*
C		Nick Brown
C	tail-recursion opt	Christian Kirkegaard
HBAL	dedicated typed AL	Matthieu Lucotte
C / JVML	datatypes	Robert Atkey
Java	**usage aspects**, datatypes	DA & MH

Future and ongoing work on LFPL

- Consider further ways to relax linearity, handle internal sharing

  separation sets $x_k :^{i_k}_{M_k} A_k \vdash e : A$ (Michal Konečný)

  sharing sets $x_k :^{S_k} A_k \vdash e : A, S, D$ (Robert Atkey)

- Inference mechanisms

  Reconstruct $\Diamond$ arguments (Steffen Jost, Dilsun Kırlı)

- Higher-order functions

  MH (POPL 2002) bounded space with HO

- Other features: arrays, polymorphism, . . .

- Related project: *Mobile Resource Guarantees* investigating PCC for resource constraints.