Relaxing a Linear Typing for In-Place Update

Michal Konečný
LFCS, University of Edinburgh

Joint work with
David Aspinall, Martin Hofmann, Robert Atkey
Overview: Main Points

- **LFPL** (Hofmann, 2000)—functional language with *heap*-aware types (◊) and operational semantics featuring:
  - *In-place update*
  - *Non-size-increasing heap usage*
  - fast execution (⇐ no GC, no heap space allocation)
  - fits environments with tight fixed memory constraints

- In-place update semantics made *correct* via *affine linear typing* (*completeness* impossible: correctness of terms *undecidable*)

- **Relaxations** of linearity for LFPL
  → more of the correct terms typed

- Several *existing relaxations* are examples of a *general method*
A Mini Version of LFPL

First order; Full recursion

Types: \( A ::= \diamondsuit | \text{Bool} | L(A) \)

Pre-terms: \( e ::= x | \text{let } x = e_1 \text{ in } e_2 | f(x_1, \ldots, x_n) \)
\[ | \quad \text{tt} | \text{ff} | \text{if } x \text{ then } e_1 \text{ else } e_2 \]
\[ | \quad \text{nil} | \text{cons}(x_h, x_t, x_d) \]
\[ | \quad \text{match } x \text{ with } \text{nil} \Rightarrow e_1 | \text{cons}(x_h, x_t, x_d) \Rightarrow e_2 \]

(Could add \( N, \times, + \), recursive types.)

full expressions instead of variables: use let variables \( \Rightarrow \) simpler typing rules
Example: Reverse

\[ reverse_A(x) = revaux_A(x, \text{nil}) \]

\[ revaux_A(x, y) = \text{match } x \text{ with} \]
\[ \text{nil} \mapsto y \]
\[ | \text{cons}(x_h, x_t, x_d) \mapsto \]
\[ revaux(x_t, \text{cons}(x_h, y, x_d)) \]

\[ x \quad y \quad \text{NIL} \]

\[ x \quad y \quad \text{NIL} \quad \rightarrow \quad x \quad y \quad \text{NIL} \]

\[ x \quad y \quad \text{NIL} \quad \rightarrow \quad x \quad y \quad \text{NIL} \]

\[ x \quad y \quad \text{NIL} \quad \rightarrow \quad x \quad y \quad \text{NIL} \]

\[ x \quad y \quad \text{NIL} \quad \rightarrow \quad x \quad y \quad \text{NIL} \]
Unconstrained Typing: Examples (Diamond Trading)

\( \vdash \text{nil} : L(A) \) (NIL)

\( x_h : A, x_t : L(A), x_d : \diamond \vdash \text{cons}(x_h, x_t, x_d) : L(A) \) (CONS)

\[
\Gamma_1 \vdash e_1 : B \\
\Gamma_2, x_h : A, x_t : L(A), x_d : \diamond \vdash e_2 : B \\
\Gamma_1, \Gamma_2 \subseteq \Gamma \\
\Gamma, x : L(A) \vdash \text{match } x \text{ with } \text{nil} \Rightarrow e_1 \mid \text{cons}(x_h, x_t, x_d) \Rightarrow e_2 : B
\] (LIST-ELIM)

\[
\Gamma \vdash e_1 : A \\
\Gamma, x : A \vdash e_2 : B \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : B
\] (LET)
Semantics

• Denotational
  Standard, ignoring diamond arguments of \texttt{cons}.
  
  $\llbracket \diamond \rrbracket = \{0\}$, $\llbracket \text{Bool} \rrbracket = \{\texttt{ff}, \texttt{tt}\}$,
  
  $\llbracket \text{L}(A) \rrbracket = \{[a_1, \ldots, a_n] \mid a_1, \ldots, a_n \in \llbracket A \rrbracket\}$
  
  $\llbracket \text{cons}(h, t, d) \rrbracket = \llbracket h \rrbracket | \llbracket t \rrbracket$, $\llbracket \text{nil} \rrbracket = \emptyset, \ldots$

  Least fixpoint semantics of recursively defined functions.

• Operational—with \textit{in-place update}
  Not by term reduction. Lists are stored using a \textit{heap}.
  Values of diamond type are \textit{pointers} into the heap.
  \textit{Call-by-value} evaluation ($e_1$ before $e_2$ in let $x = e_1$ in $e_2$).
Locations hold cons cells:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>Denotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1$</td>
<td>{hd = TT, tl = NIL}</td>
<td>[tt]</td>
</tr>
<tr>
<td>$\ell_2$</td>
<td>{hd = NIL, tl = NIL}</td>
<td>[]</td>
</tr>
<tr>
<td>$\ell_3$</td>
<td>{hd = $\ell_1$, tl = $\ell_2$}</td>
<td>[[tt], []]</td>
</tr>
<tr>
<td>$\ell_4$</td>
<td>{hd = FF, tl = NIL}</td>
<td>[ff]</td>
</tr>
</tbody>
</table>

Heap region of a list representation: all reachable locations.

more general types $\implies$ other kinds of values in locations
For all $\Gamma \vdash e : A$, define an evaluation relation

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

where

- $\sigma, \sigma'$ are heaps—initial and final
- $v \in \text{Val}$ is an operational value (heap $\sigma'$ address, NIL, TT or FF)
- $v, \sigma'$ represent a value (called result) from $[\![A]\!]$
- $S : \text{Dom}(\Gamma) \rightarrow \text{Val}$ is an environment
- $S, \sigma$ represent a tuple of values (called arguments) from $[\![\Gamma]\!]$

inductively, e.g.:

$$S, \sigma \vdash \text{cons}(x_h, x_t, x_d) \leadsto S(x_d), \sigma[S(x_d) \mapsto \{\text{hd} = S(x_h), \text{tl} = S(x_t)\}]$$
Example: Incorrect

Some terms are not (operationally) correct:

\[ [a_1, a_2, \ldots] \]

\[ \downarrow \]

\[ [a_1, a_1, a_2, a_2, \ldots] \]

\[ \text{double} : L(A) \rightarrow L(A) \]

\[ \text{double}(x) = \text{match } x \text{ with} \]

\[ \text{nil} \Rightarrow \text{nil} \]

\[ | \text{cons}(h, t, d) \Rightarrow \text{let } t_2 = \text{double}(t) \text{ in} \]

\[ \text{let } y = \text{cons}(h, t_2, d) \text{ in} \]

\[ \text{cons}(h, y, d) \]

Solution in original LFPL: \textit{linear let}

\[
\Gamma_1 \vdash e_1 : A \quad \Gamma_2, x : A \vdash e_2 : B \quad \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) = \emptyset
\]

\[ \Gamma_1, \Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : B \]

\[ \text{(LIN-LET)} \]
Examples: Correct

Some functions (with obvious meaning) simply defined in LFPL:

\[ isLonger_{A,B} : L(A), L(B) \to \text{Bool} \]
\[ maxList_A : L(L(A)) \to L(A) \]
\[ reverse_A : L(A) \to L(A) \]

Correct for every possible representation of arguments on the heap.
Examples: Conditionally Correct

Correct under some *separation conditions*, e.g.:

- **External separation**: $\text{append}_A : L(A), L(A) \rightarrow L(A)$
  (arguments must not overlap)

  ![Diagram of external separation]

  - **ok:**
  - **ko:**

- **Internal separation**: $\text{reverseItems}_A : L(L(A)) \rightarrow L(L(A))$
  (certain argument components must not overlap)

  ![Diagram of internal separation]

  - **ko:**
Examples: Correct thanks to Extra Guarantees

\[
\text{let } x = e_1 \text{ in } e_2: \text{ result of } e_1 \text{ has to meet conditions of } e_2
\]

\[\implies \text{ extra guarantees for } e_1 \text{ have to be derived, e.g.:}\]

- **non-destruction** (*y* not destroyed in *e₁*):
  
  \[
  \textbf{ok: let } x = \text{maxList}(y) \text{ in } y
  \]
  
  \[
  \textbf{ko: let } x = \text{reverse}(y) \text{ in } y
  \]

- **separation of argument from result** (in *e₁*):
  
  \[
  \textbf{ok: let } x = \text{second}(y, z) \text{ in } \text{append}(x, y)
  \]
  
  \[
  \textbf{ko: let } x = y \text{ in } \text{append}(x, y)
  \]
Guarantees correctness by

- linear typing (e.g. \texttt{LIN-LET})

and the implicit \textit{preconditions}:

- arguments \textit{do not overlap} on the heap
- arguments are \textit{not internally sharing}

Linear typing \textit{guarantees} that the result is not internally sharing.

No indication whether arguments could be preserved are considered. (Which actually enforces linearity.)

\textbf{Problem:}
\texttt{isLonger} \texttt{A,B}(x, y) needs to return reconstructed copies of its arguments
Relaxing Linearity

Motivation: typecheck more correct algorithms

Goal: Find weaker restrictions so that:

- external sharing is sometimes permitted
- “readonly” use is recognised

Method: explicit conditions and guarantees about heap layout.

Plan:

- Review two concrete existing relaxations.
- Discuss a new one.
• A variant by (Aspinall, Hofmann 2002), call it \textit{UAPL}

• One \textit{usage aspect} $\in \{1, 2, 3\}$ assigned to each argument.

• Both conditions and guarantees are expressed via these aspects.

• Informal meaning:
  
  - 1: argument maybe destroyed
  - 2: argument possibly overlapping with the result
  - 3: argument separated from the result
Example UAPL Rules

\[
\begin{align*}
\text{(CONS)} \\
\chi_h :^2 A, \chi_t :^2 L(A), \chi_d :^1 \vdash \text{cons}(\chi_h, \chi_t, \chi_d) : L(A)
\end{align*}
\]

\[
\begin{align*}
\Delta_1, \Theta, x :^i A \vdash e_2 : B & \quad \forall z. \phi(i, \Delta_1[z], \Delta_2[z]) \\
\Gamma, \Delta_1 \vdash e_1 : A & \quad \Delta_2, \Theta, x :^i A \vdash e_2 : B & \quad \forall z. \phi(i, \Delta_1[z], \Delta_2[z]) \\
\Gamma, \Theta, \Delta_1 \land \Delta_2 \vdash \text{let } x = e_1 \text{ in } e_2 : B & \quad \text{(LET)}
\end{align*}
\]

where \(\phi(i, \Delta_1[z], \Delta_2[z])\) evaluates according to the table:

\[
\begin{array}{cccccccc}
\chi & \Delta_1[z] & 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\
\hline
\Delta_2[z] & 1 & x & x & x & x & x & x & x & x \\
2 & x & x & x & x & x & x & x & \checkmark & \checkmark \\
3 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\end{array}
\]
Examples:

\[
\begin{align*}
&x :^1 \text{L}(A), y :^2 \text{L}(A) \vdash \text{append}_A(x, y) : \text{L}(A) \\
&x :^3 \text{L}(A), y :^3 \text{L}(B) \vdash \text{isLonger}_{A,B}(x, y) : \text{Bool}
\end{align*}
\]

- **1:**
  - \(C\): argument separated from all the others
  - \(C\): list elements are separated on the heap
  - \(G\): no guarantee (argument could be even destroyed)

- **2:**
  - \(C\): argument separated from all the others
  - \(C\): list elements are separated on the heap
  - \(G\): argument preserved

- **3:**
  - \(C\): argument separated from arguments with aspect 1 or 2
  - \(G\): argument preserved and separated from result

- \(G\): list elements separated in the result
A variant by Robert Atkey (2002), work in progress, call it \textit{ESPL}.

Syntax of typing judgement $+ (C, G)$:

$$\Gamma \vdash e : A, S, D$$

where $\Gamma$ contains assumptions $x : (A_x, S_x)$

$S_x \subseteq \text{Dom}(\Gamma)$: arguments which $x$ is allowed to share with

$S \subseteq \text{Dom}(\Gamma)$: arguments allowed to share with result (aspect 2)

$D \subseteq \text{Dom}(\Gamma)$: arguments allowed to be destroyed (aspect 1)

Examples:

$$x : (L(N), \{x\}) , y : (L(N), \{y\}) \vdash \text{append}_N(x, y) : L(N), \{y\}, \{x\}$$

$$\Gamma \vdash e_1 : A, S_1, D_1 \quad \Gamma[\setminus D_1, x \mapsto (A, S_1)] \vdash e_2 : B, S_2, D_2$$

$$\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : B, S_2 \setminus \{x\}, (D_1 \cup D_2) \setminus \{x\} \quad (\text{LET})$$
Comparison

- UAPL can be *embedded* into ESPL
  \[\implies\] UAPL is weaker than ESPL

- ESPL produces *more kinds of internal sharing* (Atkey 2002):

  \[
  \text{let } x = \text{append}(z, y) \text{ in } \\
  \text{cons}(x, \text{cons}(y, \text{cons}(x, \text{nil}, d_3), d_2), d_1)
  \]

  UAPL requires that \(x\) and \(y\) not share (aspect 2)

- ESPL has simpler rules

- UAPL is more suitable for extending to higher order
  \[\iff\] information is kept per-argument only
Neither language typechecks \( \text{reverse}(x) \) allowing \( x \) to share internally:

\[
\text{revaux}_A(x, y) = \text{match } x \text{ with }\\
\quad \text{nil} \Rightarrow y \\
\quad | \text{cons}(h, t, d) \Rightarrow \\
\quad \quad \text{revaux}_A(t, \text{cons}(h, y, d))
\]

\( d, y \) cannot share \( \implies \) \( x, y \) cannot share

Refined: \( d, y \) cannot share \( \implies \) \( x, y \) cannot share \textbf{control structure} can share \textbf{on element level}

Need to distinguish \textit{deep and shallow} regions of values on the heap.
Conclusion

The general C-G approach helps to

- easily compare and extend the various LFPL variants
- formulate simpler proofs of correctness
- implement automatic derivation of product types

Further work:

- Implement compiler for ESPL $\rightarrow$ C,JVM
- Extend UAPL to *higher order*
- Define LFPL distinguishing *deep and shallow* levels