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 ($\Diamond$) and operational semantics featuring:
  - *In-place update*
  - *Non-size-increasing heap usage*
  - fast execution ($\Longleftarrow$ 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
  $\Longrightarrow$ 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: $A \quad ::= \quad \Diamond \mid \mathsf{Bool} \mid \mathsf{L}(A)$

Pre-terms:

$$
\begin{array}{rcl}
e & ::= & x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid f(x_1, \ldots, x_n) \\
 & \mid & \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & \mid & \mathsf{nil} \mid \mathsf{cons}(x_h, x_t, x_d) \\
 & \mid & \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_1 | \mathsf{cons}(x_h, x_t, x_d){\Rightarrow}e_2
\end{array}
$$

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

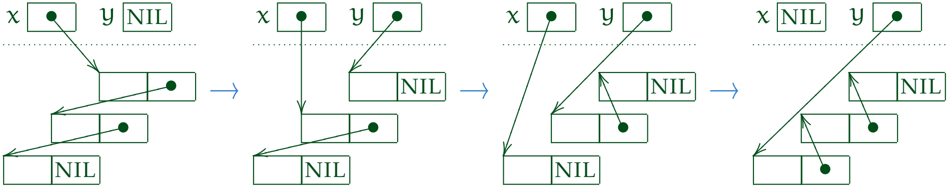
full expressions instead of variables: use let

variables $\Longrightarrow$ simpler typing rules

Example: Reverse

$$reverse_A(x) = revaux_A(x, \mathsf{nil})$$

$$\begin{aligned} revaux_A(x, y) = \ & \mathsf{match}\ x\ \mathsf{with} \\ & \quad \mathsf{nil} \Rightarrow y \\ & \quad \mid \mathsf{cons}(x_h, x_t, x_d) \Rightarrow \\ & \qquad revaux(x_t, \mathsf{cons}(x_h, y, x_d)) \end{aligned}$$



Unconstrained Typing: Examples (Diamond Trading)

$$\frac{}{\vdash \mathsf{nil} : \mathsf{L}(A)} \qquad \text{(NIL)}$$

$$\frac{}{x_h : A, x_t : \mathsf{L}(A), x_d : \Diamond \vdash \mathsf{cons}(x_h, x_t, x_d) : \mathsf{L}(A)} \qquad \text{(CONS)}$$

$$\frac{\Gamma_1 \vdash e_1 : B \qquad \Gamma_2, x_h : A, x_t : \mathsf{L}(A), x_d : \Diamond \vdash e_2 : B \qquad \Gamma_1, \Gamma_2 \subseteq \Gamma}{\Gamma, x : \mathsf{L}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_1 | \mathsf{cons}(x_h, x_t, x_d) \Rightarrow e_2 : B} \qquad \text{(LIST-ELIM)}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B} \qquad \text{(LET)}$$

Semantics

- Denotational

  Standard, ignoring diamond arguments of cons.

  $[\![\Diamond]\!] = \{0\}$, $[\![\mathsf{Bool}]\!] = \{\mathrm{ff}, \mathrm{tt}\}$,

  $[\![\mathsf{L}(A)]\!] = \{[a_1, \ldots, a_n] \mid a_1, \ldots, a_n \in [\![A]\!]\}$

  $[\![\mathsf{cons}(h, t, d)]\!] = [[\![h]\!] | [\![t]\!]]$, $[\![\mathsf{nil}]\!] = []$, ...

  Least fixpoint semantics of recursively defined functions.

- Operational—with *in-place update*

  Not by term reduction. Lists are stored using a *heap*.

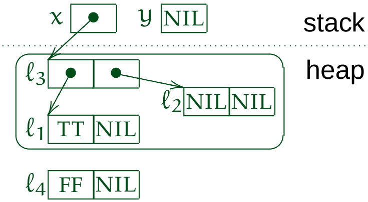
  Values of diamond type are *pointers* into the heap.

  Call-by-value evaluation (e_1 before e_2 in $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$).

Heap

Locations hold cons cells:

Location	Contents	Denotation
ℓ_1 :	$\{\mathsf{hd} = \mathrm{TT}, \mathsf{tl} = \mathrm{NIL}\}$	$[\mathsf{tt}]$
ℓ_2 :	$\{\mathsf{hd} = \mathrm{NIL}, \mathsf{tl} = \mathrm{NIL}\}$	$[[]]$
ℓ_3 :	$\{\mathsf{hd} = \ell_1, \mathsf{tl} = \ell_2\}$	$[[\mathsf{tt}], []]$
ℓ_4 :	$\{\mathsf{hd} = \mathrm{FF}, \mathsf{tl} = \mathrm{NIL}\}$	$[\mathsf{ff}]$



more general types $\Longrightarrow$ other kinds of values in locations

Heap region of a list representation: all *reachable* locations.

Evaluation Relation

For all $\Gamma \vdash e : A$, define an evaluation relation

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

where

σ, σ' are heaps—initial and final

$v \in \mathrm{Val}$ is an *operational value* (heap σ' address, NIL, TT or FF)

v, σ' *represent* a value (called *result*) from $[\![A]\!]$

$S\colon \mathrm{Dom}(\Gamma) \to \mathrm{Val}$ is an *environment*

S, σ *represent* a tuple of values (called *arguments*) from $[\![\Gamma]\!]$

inductively, e.g.:

$$\frac{}{S, \sigma \vdash \mathsf{cons}(x_h, x_t, x_d) \leadsto S(x_d), \sigma\left[S(x_d) \mapsto \{\mathsf{hd} = S(x_h), \mathsf{tl} = S(x_t)\}\right]}$$

Example: Incorrect

Some terms are not (operationally) correct:

$$[a_1, a_2, \ldots] \;\downarrow\; [a_1, a_1, a_2, a_2, \ldots]$$

$$\begin{aligned}
&\textit{double} : \mathsf{L}(A) \to \mathsf{L}(A)\\
&\textit{double}(x) = \text{ match } x \text{ with}\\
&\qquad \text{nil} \Rightarrow \text{nil}\\
&\qquad |\ \text{cons}(h, t, d) \Rightarrow \text{let } t_2 = \textit{double}(t) \text{ in}\\
&\qquad\qquad \text{let } y = \text{cons}(h, t_2, d) \text{ in}\\
&\qquad\qquad\quad \text{cons}(h, y, d)
\end{aligned}$$

Solution in original LFPL: *linear let*

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2, x : A \vdash e_2 : B \qquad \mathrm{Dom}(\Gamma_1) \cap \mathrm{Dom}(\Gamma_2) = \emptyset}{\Gamma_1, \Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : B}$$

(LIN-LET)

Examples: Correct

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

$$\begin{aligned} \mathit{isLonger}_{A,B} &: \mathsf{L}(A), \mathsf{L}(B) \to \mathsf{Bool} \\ \mathit{maxList}_{A} &: \mathsf{L}(\mathsf{L}(A)) \to \mathsf{L}(A) \\ \mathit{reverse}_{A} &: \mathsf{L}(A) \to \mathsf{L}(A) \end{aligned}$$

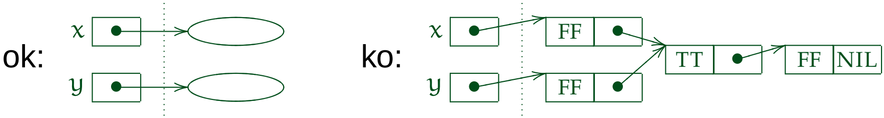
(see above)

Correct for every possible representation of arguments on the heap.

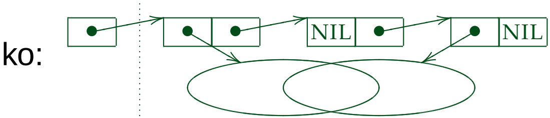
Examples: Conditionally Correct

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

- External separation: $append_A : \mathsf{L}(A), \mathsf{L}(A) \to \mathsf{L}(A)$
  (arguments must not overlap)



- Internal separation: $reverseItems_A : \mathsf{L}(\mathsf{L}(A)) \to \mathsf{L}(\mathsf{L}(A))$
  (certain argument components must not overlap)



Examples: Correct thanks to Extra Guarantees

let $x = e_1$ in e_2: result of e_1 has to meet conditions of e_2

$\Longrightarrow$ extra *guarantees* for e_1 have to be derived, e.g.:

- non-destruction (y not destroyed in e_1):

  ok: let $x = \mathit{maxList}(y)$ in y

  ko: let $x = \mathit{reverse}(y)$ in y

- separation of argument from result (in e_1):

  ok: let $x = \mathit{second}(y, z)$ in $\mathit{append}(x, y)$

  ko: let $x = y$ in $\mathit{append}(x, y)$

Original LFPL

Guarantees correctness by

- linear typing (e.g. LIN-LET)

and the implicit *preconditions*:

- arguments *do not overlap* on the heap
- arguments are *not internally sharing*

Linear typing *guarantees* that the result is not internally sharing.

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

Problem:
$\mathit{isLonger}_{A,B}(x, y)$ 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.

LFPL with Usage Aspects

- A variant by (Aspinall, Hofmann 2002), call it *UAPL*
- One *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

$$\frac{}{x_h :^2 A, x_t :^2 \mathsf{L}(A), x_d :^1 \Diamond \vdash \mathsf{cons}(x_h, x_t, x_d) : \mathsf{L}(A)} \quad (\text{CONS})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \qquad \Delta_2, \Theta, x :^i A \vdash e_2 : B \qquad \forall z. \phi(i, \Delta_1[z], \Delta_2[z])}{\Gamma^i, \Theta, \Delta_1^i \wedge \Delta_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B} \quad (\text{LET})$$

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

i	1			2			3		
$\Delta_1[z] \backslash \Delta_2[z]$	1	2	3	1	2	3	1	2	3
1	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	$\surd$	$\surd$
3	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Usage Aspects as Conditions and Guarantees

Examples:

$$x :^1 \mathsf{L}(A), y :^2 \mathsf{L}(A) \vdash \mathit{append}_A(x, y) : \mathsf{L}(A)$$
$$x :^3 \mathsf{L}(A), y :^3 \mathsf{L}(B) \vdash \mathit{isLonger}_{A,B}(x, y) : \mathsf{Bool}$$

- 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

LFPL with Explicit Sharing

A variant by Robert Atkey (2002), work in progress, call it *ESPL*.

Syntax of typing judgement + (C, G):

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

where Γ contains assumptions $x : (A_x, S_x)$

$S_x \subseteq \mathrm{Dom}(\Gamma)$: arguments which x is allowed to share with
$S \subseteq \mathrm{Dom}(\Gamma)$: arguments allowed to share with result (aspect 2)
$D \subseteq \mathrm{Dom}(\Gamma)$: arguments allowed to be destroyed (aspect 1)

Examples:

$$x : (\mathsf{L}(\mathsf{N}), \{x\}), y : (\mathsf{L}(\mathsf{N}), \{y\}) \vdash \mathit{append}_{\mathsf{N}}(x, y) : \mathsf{L}(\mathsf{N}), \{y\}, \{x\}$$

$$\frac{\Gamma \vdash e_1 : A, S_1, D_1 \qquad \Gamma[\backslash D_1, x \mapsto (A, S_1)] \vdash e_2 : B, S_2, D_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B, S_2 \setminus \{x\}, (D_1 \cup D_2) \setminus \{x\}} \quad (\text{LET})$$

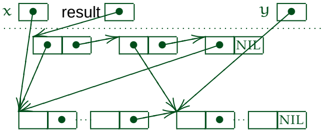
Comparison

- UAPL can be *embedded* into ESPL

  $\Longrightarrow$ UAPL is weaker than ESPL

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

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



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

- ESPL has simpler rules

- UAPL is more suitable for extending to higher order

  $\Longleftarrow$ information is kept per-argument only

Computing with Internally Shared Structures

Neither language typechecks $\mathit{reverse}(x)$ allowing x to share internally:

$$\begin{aligned} \mathit{revaux}_A(x, y) = \ & \text{match } x \text{ with} \\ & \quad \text{nil} \Rightarrow y \\ & \quad \mid \text{cons}(h, t, d) \Rightarrow \\ & \qquad \mathit{revaux}_A(t, \text{cons}(h, y, d)) \end{aligned}$$

d, y cannot share $\Longrightarrow x, y$ cannot share

<u>Refined</u>: d, y cannot share $\Longrightarrow x, y$ cannot share *control structure*
can share *on element level*

Need to distinguish *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 $\longrightarrow$ C,JVM
- Extend UAPL to *higher order*
- Define LFPL distinguishing *deep and shallow* levels