Typing with Conditions and Guarantees for Functional In-place Update

Michal Konečný

LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
`mkonecny@inf.ed.ac.uk`,
WWW: http://homepages.inf.ed.ac.uk/mkonecny/

Abstract. Hofmann's LFPL is a functional language with constructs that can be interpreted as referring to heap locations. In this view, the language is suitable for expressing and verifying in-place update algorithms. Correctness of in-place evaluation is guaranteed by a linear typing. As linearity prevents sharing on the heap, LFPL rejects many sound, natural in-place update algorithms with sharing. Recently, Aspinall and Hofmann added usage aspects to parameters of terms in first-order LFPL in order to type-check sound non-linear programs. Nevertheless, soundness of this system has not been fully established.
We show a more subtle meaning of the usage aspects as pre-conditions and (rely-)guarantees about the heap layout before and after evaluation. This interpretation allows a manageable proof of soundness for Aspinall and Hofmann's system. Secondly, we present an algorithm for inferring the strongest sound usage aspects for typable recursive programs.
We outline two other annotated typings of LFPL as systems inferring pre-conditions and (rely-)guarantees, both extending usage aspects. One is Atkey's system based on explicit indication of sharing among parameters in typing contexts and the other one is a system by the author which admits LFPL programs in which datatypes share at different layers. The latter is based on the author's conditions-and-guarantees approach to usage aspects.

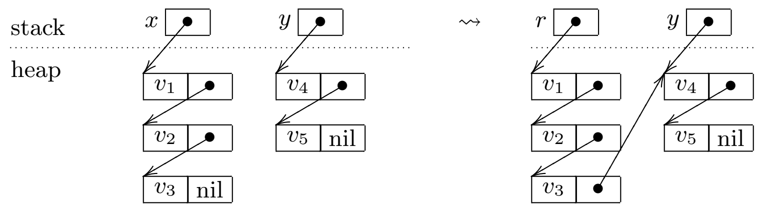
1 Introduction

This paper is based on the research language LFPL (Linear Functional Programming Language) introduced by Hofmann [5]. Hofmann has proved several important complexity theoretic results about the language in [5, 7]. Here we will build on the more operational view of LFPL as first explored by Hofmann in [6]. The language turns out to be well suited for writing functional programs that can be evaluated efficiently with in-place update. The approach is currently confined to first-order with arbitrary recursion provided via named functions.

For example, in LFPL we can write the following program to append two lists:

$$\begin{aligned} \mathit{append}(x,y) = {} & \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \rightarrow y \\ & \qquad\qquad \mid \mathsf{Cons}(h,t)@d \rightarrow \mathsf{Cons}(h, \mathit{append}(t,y))@d \end{aligned}$$

Fig. 1. Evaluation of *append* in LFPL



The new feature of LFPL over, say, ML is that most datatype constructors take an additional argument. In a suitable operational interpretation, this argument stands for the heap location that the constructor occupies. Its type is $\Diamond$ which is a new basic type called diamond. Heap locations are virtually ignored in the denotational semantics and therefore the program is a definition of list concatenation as claimed. Nevertheless, the operational semantics will append the lists in-place: it modifies the last cons-cell of the first list if the list is not empty and it overwrites the other cons-cells of the first list with the same contents (see Fig. 1). We will review LFPL and its operational semantics in Section 2.

This evaluation strategy can go wrong easily in the presence of sharing. For example, $\mathit{append}(x, x)$ evaluates to a heap with a circular structure for a non-empty list x. Various terms might fail to evaluate or evaluate with incorrect results. We will call a term *(operationally) sound* if it evaluates in harmony with its denotational semantics. The theme of this paper is the approximation of this semantical property with a typing system.

Very few terms are operationally sound in all circumstances and therefore we consider soundness subject to some *pre-condition* on the layout of the values of its free variables (parameters) on the heap. For example, $\mathit{append}(x, y)$ is sound whenever x is separated from y.

In the original LFPL from [6], soundness is achieved via affine linear typing and the ambient pre-condition that all parameters are separated from each other and, moreover, the heap satisfies the single pointer property.

Affine linear typing is neat but it appears to be too restrictive for practical programming as it rejects many natural and sound in-place update algorithms. For example, the terms (x, x) (a tensor pair) and $\mathsf{Cons}(\mathit{even}(x), x)@d$ are sound under the LFPL separation pre-conditions but do not type-check. (Where *even* is an obviously defined function testing the parity of the length of a list.)

A solution to these problems has been introduced by Aspinall and Hofmann [1]. They distinguish three ways in which a parameter is used during the evaluation of a term. Each variable in a typing context Γ within a typing judgement $\Gamma \vdash e : A$ is assigned a number 1, 2 or 3, called *usage aspect*, which represents a *guarantee* on how the parameter might be used during an evaluation of the term:

- 1: potentially destructive use *(weakest guarantee)*
- 2: read-only use, potentially sharing with the result
- 3: read-only use, not sharing with the result *(strongest guarantee)*

In their system, *append* and *even* can have the following typings:

$$x :^1 \mathsf{L}(A), y :^2 \mathsf{L}(A) \vdash \mathit{append}(x, y) : \mathsf{L}(A) \qquad x :^3 \mathsf{L}(A) \vdash \mathit{even}(x) : \mathsf{Bool}$$

reflecting that the first argument of *append* may be destroyed (if not empty) and the second one is not destroyed but may (if not empty) be shared with the result. Thanks to usage aspects the expression $\mathsf{Cons}(\mathit{even}(x), x)@d$ mentioned above type-checks. Moreover, usage aspects allow a sound treatment of cartesian products and thus a pair (x, x) which type-checks can be constructed.

The idea of usage aspects is similar to that of *use types* in linear logic [4] and also to *passivity* [14] within syntactic control of interference [16, 17].

It turned out that it was apparently rather hard to prove the soundness of the typing rules given in [1]. The paper contains only an outline of the soundness proof. The details were provided by the present author and are being prepared for publication. The key idea is to consider carefully both the *conditions and guarantees* that usage aspects represent, not only the guarantees.

This paper has the following goals, pursued in Sections 3 and 4, respectively:

1. Present a refined view of usage aspects as conditions and guarantees which
   - allows an easier proof of soundness,
   - provides an efficient algorithm inferring the strongest sound usage aspects for typable recursive programs. (In particular, strongest sound usage aspects exist.)
2. Abstract away from usage aspects and portray from this point of view other, more expressive, typing systems for LFPL.

2 LFPL

We introduce the types and terms of the language, following [1] with the exception that we leave out binary trees and replace natural numbers with booleans to avoid arithmetic features which are irrelevant to our study:

$$\begin{aligned}
A ::= {} & \Diamond \mid \mathsf{Bool} \mid A_1 \times A_2 \mid A_1 \otimes A_2 \mid \mathsf{L}(A) \\
e ::= {} & x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid f(x_1, \ldots, x_n) \\
\mid {} & \mathsf{TT} \mid \mathsf{FF} \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
\mid {} & (e_1, e_2) \mid \mathsf{fst}(x) \mid \mathsf{snd}(x) \\
\mid {} & x_1 \otimes x_2 \mid \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \to e \\
\mid {} & \mathsf{Nil}_A \mid \mathsf{Cons}(x_h, x_t)@x_d \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \to e_1 | \mathsf{Cons}(x_h, x_t)@x_d \to e_2
\end{aligned}$$

The language can be extended for arbitrary inductive datatypes the same way as in [3] or in [11].

Notice the extensive use of variables instead of expressions in term constructors. All of the usual, more general, forms of the terms can be simulated by the use of let. The reason for this restriction is to confine most of the reasoning about sharing and destroying of heap resources around the let expressions.

The plain typing (i.e. typing judgements $\Gamma \vdash e : A$ without usage aspects) is defined as usual apart from the rules for list constructors and destructor where the new location argument d in $\mathsf{Cons}(h,t)@d$ is has to have the diamond type $\Diamond$. The typing rules can be obtained from those in Fig. 3 by removing usage aspects.

In LFPL there is no means of allocating new space on the heap which corresponds to the fact that there is no closed term of type $\Diamond$. This means that the heap space of the arguments is the only space available to an LFPL function. Consequently, all functions defined in LFPL are non-size-increasing. This is significant to the complexity theoretic study of LFPL but not crucial in our context. We could therefore extend the language with a term new whose type is $\Diamond$ and the rest of the paper would be easily extended accordingly.

A *program* P is a map from a finite set of function names to valid typing judgements in which the order of parameters is significant: $P(f) = \Gamma_f \vdash e_f : A_f$. All function names used in e_f have to be in $\mathrm{Dom}(P)$. The variables in Γ_f are the named arguments of f. The pair (Γ_f, A_f) captures the signature of f.

A denotational semantics $[\![A]\!]$ for each type A is given using flat Scott domains. For the diamond type we out $[\![\Diamond]\!] = \{\diamond\}_\perp$ and the rest is standard. The denotation of programs $[\![P]\!]$ and terms $[\![e]\!]_{\eta,[\![P]\!]}$ with a valuation η of variables (including the ones that are free in e) is defined by a least fixed point as usual.

2.1 Heap

Before we can formalise the evaluation of LFPL described in the Introduction, we need to specify how the denotational values can be represented on a heap. A heap σ is a finite map from locations to heap values. In the case of our simple language, the only values stored in heap locations are list cons-cells represented as $\{\mathrm{hd} = v_h, \mathrm{tl} = v_t\}$ where v_h, v_t are *operational values* of the following form:

$$v ::= \mathrm{tt} \mid \mathrm{ff} \mid (v_1, v_2) \mid \mathrm{nil} \mid \ell$$

where ℓ ranges over heap locations[1]. These operational values are also assigned to variables in environments. (See Fig. 1 in the Introduction for an illustration.)

Thus an operational value v together with a heap σ may represent a denotational value $a \in [\![A]\!]$, e.g. a list or a pair. This relationship is formalised by the relation $v \Vdash^{\sigma}_{A,\infty} a$ and extended to tuples by the relation $S \Vdash^{\sigma}_{\Gamma,\infty} \eta$. The definition of $v \Vdash^{\sigma}_{A,\infty} a$ is more or less derivable from the rules for the evaluation

[1] We could take a different strategy and have $\{\mathrm{hd} = \ell_h, \mathrm{tl} = \ell_t\}$ with two pointers, one for head and one for tail. Nevertheless, this would require that basic values like tt, ff and the empty list occupy a heap location. This would result in having $\mathsf{TT}@d$, $\mathsf{FF}@d$ and $\mathsf{Nil}_A@d$ which would make it clumsier to write programs in LFPL.

relation and the definition of region below. For illustration, we quote the clauses defining the representation relation for lists:

$$\frac{}{\text{nil} \Vdash^{\sigma}_{\mathsf{L}(A),\infty} []} \qquad \frac{\sigma(\ell) = \{\text{hd} = v_h, \text{tl} = v_t\} \quad v_h \Vdash^{\sigma}_{A,\infty} h \quad v_t \Vdash^{\sigma}_{\mathsf{L}(A),\infty} t}{\ell \Vdash^{\sigma}_{\mathsf{L}(A),\infty} h :: t}$$

The full definition can be found in [12]. Notice that this representation of lists does not require the head not to share heap with the tail. In fact, this relation does not put any restrictions on aliasing of parts of the value, not even on the components of a tensor pair. Therefore we define another representation relation $v \Vdash^{\sigma}_{A,\circ\circ} a$ which differs from the above only by asserting certain separation conditions[2]. The change affects only the rules for tensor pairs and cons-cells (apart from replacing ∞ with $\circ\circ$). The latter one becomes:

$$\frac{\begin{array}{c}\sigma(\ell) = \{\text{hd} = v_h, \text{tl} = v_t\} \quad v_h \Vdash^{\sigma}_{A,\circ\circ} h \quad v_t \Vdash^{\sigma}_{\mathsf{L}(A),\circ\circ} t \\ R_A(v_h, \sigma) \cap R_{\mathsf{L}(A)}(v_t, \sigma) = \emptyset\end{array}}{\ell \Vdash^{\sigma}_{\mathsf{L}(A),\circ\circ} h :: t}$$

where $R_A(v, \sigma)$ is the *region* of the value of type A represented by v on σ, i.e. the set of locations in σ "reachable" from v. Whenever it holds $v \Vdash^{\sigma}_{A,\infty} a$, the region is defined as follows:

$$\begin{aligned}
R_{\mathsf{Bool}}(\text{tt}, \sigma) &= R_{\mathsf{Bool}}(\text{ff}, \sigma) = \emptyset \\
R_{A_1 \otimes A_2}((v_1, v_2), \sigma) &= R_{A_1 \times A_2}((v_1, v_2), \sigma) = R_{A_1}(v_1, \sigma) \cup R_{A_2}(v_2, \sigma) \\
R_{\mathsf{L}(A)}(\text{nil}, \sigma) &= \emptyset \\
R_{\mathsf{L}(A)}(\ell, \sigma) &= \{\ell\} \cup R_A(v_h, \sigma) \cup R_{\mathsf{L}(A)}(v_t, \sigma) \\
&\quad \text{if } \sigma(\ell) = \{\text{hd} = v_h, \text{tl} = v_t\} \\
R_{\Diamond}(\ell, \sigma) &= \{\ell\}
\end{aligned}$$

The notion of region is easily extended to tuples of values represented by an environment: $R_{\Gamma}(S, \sigma) = \bigcup_{x \in \text{Dom}(S)} R_{\Gamma(x)}(S(x), \sigma)$.

Values of some types, e.g. $\mathsf{Bool} \times \mathsf{Bool}$, do not use heap at all. Such types are called *heap-free.*

2.2 Evaluation

The operational semantics of LFPL terms is expressed by a big-step evaluation relation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ where

- S is the environment mapping the free variables of e to operational values which refer to the initial heap σ,
- v is the result operational value referring to the result heap σ'.

The definition of the evaluation relation is in Fig. 2.

[2] The symbols ∞ and $\circ\circ$, suggested by Martin Hofmann, stand for sharing and separation, respectively.

Fig. 2. Definition of Evaluation Relation

[VAR]
$$\frac{}{S, \sigma \vdash x \rightsquigarrow S(x), \sigma}$$

[FUNC]
$$\frac{S, \sigma \vdash e_f \rightsquigarrow v, \sigma' \quad \Gamma_f = y_1 : A_1, \ldots, y_n : A_n}{S \circ (x_i \mapsto y_i), \sigma \vdash f(x_1, \ldots, x_n) \rightsquigarrow v, \sigma'}$$

[LET]
$$\frac{S, \sigma \vdash e_1 \rightsquigarrow v, \sigma' \quad S[x \mapsto v], \sigma' \vdash e_2 \rightsquigarrow v', \sigma''}{S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v', \sigma''}$$

[TRUE]
$$\frac{}{S, \sigma \vdash \mathsf{TT} \rightsquigarrow \mathrm{tt}, \sigma}$$

[FALSE]
$$\frac{}{S, \sigma \vdash \mathsf{FF} \rightsquigarrow \mathrm{ff}, \sigma}$$

[IF-TRUE]
$$\frac{S(x) = \mathrm{tt} \quad S, \sigma \vdash e_1 \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v, \sigma'}$$

[IF-FALSE]
$$\frac{S(x) = \mathrm{ff} \quad S, \sigma \vdash e_2 \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v, \sigma'}$$

[TENSOR-PAIR]
$$S, \sigma \vdash x_1 \otimes x_2 \rightsquigarrow (S(x_1), S(x_2)), \sigma$$

[TENSOR-ELIM]
$$\frac{S(x) = (v_1, v_2) \quad S[x_1 \mapsto v_1][x_2 \mapsto v_2], \sigma \vdash e \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \to e \rightsquigarrow v, \sigma'}$$

[CART-PAIR]
$$\frac{S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma' \quad S, \sigma' \vdash e_2 \rightsquigarrow v_2, \sigma''}{S, \sigma \vdash (e_1, e_2) \rightsquigarrow (v_1, v_2), \sigma''}$$

[FST]
$$\frac{S(x) = (v_1, v_2)}{S, \sigma \vdash \mathsf{fst}(x) \rightsquigarrow v_1, \sigma}$$

[SND]
$$\frac{S(x) = (v_1, v_2)}{S, \sigma \vdash \mathsf{snd}(x) \rightsquigarrow v_2, \sigma}$$

[NIL]
$$S, \sigma \vdash \mathsf{Nil}_A \rightsquigarrow \mathrm{nil}, \sigma$$

[CONS]
$$S, \sigma \vdash \mathsf{Cons}(x_h, x_t)@x_d \rightsquigarrow S(x_d), \sigma[S(x_d) \mapsto \{\mathrm{hd} = S(x_h), \mathrm{tl} = S(x_t)\}]$$

[MATCH-LIST-NIL]
$$\frac{S(x) = \mathrm{nil} \quad S, \sigma \vdash e_1 \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \to e_1 | \mathsf{Cons}(x_h, x_t)@x_d \to e_2 \rightsquigarrow v, \sigma'}$$

[MATCH-LIST-CONS]
$$\frac{\sigma(S(x)) = \{\mathrm{hd} = v_h, \mathrm{tl} = v_t\} \quad S[x_h \mapsto v_h, x_t \mapsto v_t, x_d \mapsto S(x)], \sigma \vdash e_2 \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \to e_1 | \mathsf{Cons}(x_h, x_t)@x_d \to e_2 \rightsquigarrow v, \sigma'}$$

Lemma 1. *For every evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ with the representation of parameters $S \Vdash^{\sigma}_{\Gamma,\infty} \eta$ and of result $v \Vdash^{\sigma'}_{A,\infty} a$, it holds:*

1. $\mathrm{Dom}(\sigma) = \mathrm{Dom}(\sigma')$
2. $R_A(v, \sigma') \subseteq R_\Gamma(S, \sigma)$
3. $\forall \ell \in \mathrm{Dom}(\sigma) \setminus R_\Gamma(S, \sigma)$, $\sigma(\ell) = \sigma'(\ell)$

This lemma can be proved easily by induction on the structure of e from the full definition of evaluation relation and region. If we would add heap allocation to LFPL, statements (1) and (2) would have to be adjusted.

3 Usage Aspects

In this section, we review the usage aspects introduced by Aspinall and Hofmann in [1], expose our foundations for proving the soundness of the system and, finally, discuss an efficient aspect inference algorithm.

As shown in the Introduction, the syntax of typing judgements annotated with usage aspects is as follows:

$$x_1 :^{i_1} A_1, \ldots, x_n :^{i_n} A_n \vdash e : A \qquad n \in \mathbb{N}, i_j \in \{1, 2, 3\} \text{ for } 1 \leq j \leq n$$

Unless stated otherwise, from now on, assume that every Γ_f is a typing judgement with usage aspects, i.e. that we deal with annotated programs.

A full list of typing rules with usage aspects can be found in Fig. 3. The rules use the following notation: $\Gamma[x]$ stands for the aspect of x in Γ and Γ^i stands for the context which arises from Γ by changing any usage aspect 2 in Γ to i. The joined context $\Gamma_1 \wedge \Gamma_2$ is defined when the contexts are compatible (i.e. $\Gamma_1(x) = \Gamma_2(x)$ for every $x \in |\Gamma_1| \cap |\Gamma_2|$ where $|\Gamma|$ is the set of all variables in Γ) and annotations are calculated as the minimum where the contexts Γ_1, Γ_2 overlap: $\forall x \in |\Gamma_1| \cap |\Gamma_2|. (\Gamma_1 \wedge \Gamma_2)[x] = \min(\Gamma_1[x], \Gamma_2[x])$.

3.1 Soundness

We aim to define when a typing judgement annotated with usage aspects is sound in such a way that each typing rule would transform sound premises to a sound conclusion. This consists in finding an interpretation of usage aspects which is more subtle than the one given in [1]. The interpretation which follows has evolved during the search for a proof of the desired property:

- $\underline{x :^1 A}$: *condition*: x is separated from all the other parameters
  condition: x is represented without internal sharing
  guarantee: none (x could be even destroyed)
- $\underline{x :^2 A}$: *condition*: x is separated from all parameters with aspect 1
  guarantee: x is preserved during evaluation
- $\underline{x :^3 A}$: *condition*: x is separated from all parameters with aspect 1
  guarantee: x is preserved during evaluation
  guarantee: x is separated from the result with the exception of its portion which shares with parameters of aspect 2

Fig. 3. LFPL typing rules with usage aspects

[DROP]
$$\frac{\Gamma, x :^{i} A \vdash e : A' \quad j \leq i}{\Gamma, x :^{j} A \vdash e : A'}$$

[RAISE]
$$\frac{\Gamma \vdash e : A \quad A \text{ heap-free}}{\Gamma^{3} \vdash e : A}$$

[VAR]
$$\frac{}{\Gamma, x :^{2} A \vdash x : A}$$

[WEAK]
$$\frac{\Gamma \vdash e : A'}{\Gamma, x :^{3} A \vdash e : A'}$$

[LET]
$$\frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2, x :^{i} A \vdash e_2 : A' \quad \begin{array}{l}\text{Either } \forall z \in |\Gamma_1| \cap |\Gamma_2| . \Gamma_1[z] = 3, \\ \text{or } i = 3, \forall z \in |\Gamma_1| \cap |\Gamma_2| . \Gamma_1[z], \Gamma_2[z] \geq 2\end{array}}{\Gamma_1^{i} \wedge \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : A'}$$

[TRUE]
$$\frac{}{\vdash \mathsf{TT} : \mathsf{Bool}}$$

[FALSE]
$$\frac{}{\vdash \mathsf{FF} : \mathsf{Bool}}$$

[IF]
$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma, x :^{3} \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}$$

[CART-INTRO]
$$\frac{\Gamma_1 \vdash e_1 : A_1 \quad \Gamma_2 \vdash e_2 : A_2 \quad \forall z \in |\Gamma_1| \cap |\Gamma_2| . \Gamma_1[z], \Gamma_2[z] \geq 2}{\Gamma_1 \wedge \Gamma_2 \vdash (e_1, e_2) : A_1 \times A_2}$$

[FST]
$$\frac{}{x :^{2} A_1 \times A_2 \vdash \mathsf{fst}(x) : A_1}$$

[SND]
$$\frac{}{x :^{2} A_1 \times A_2 \vdash \mathsf{snd}(x) : A_2}$$

[TENS-INTRO]
$$\frac{}{x_1 :^{2} A_1, x_2 :^{2} A_2 \vdash x_1 \otimes x_2 : A_1 \otimes A_2}$$

[TENS-ELIM]
$$\frac{\Gamma, x_1 :^{i_1} A_1, x_2 :^{i_2} A_2 \vdash e : A' \quad i = \min(i_1, i_2)}{\Gamma, x :^{i} A_1 \otimes A_2 \vdash \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \to e : A'}$$

[NIL]
$$\frac{}{\vdash \mathsf{Nil}_A : \mathsf{L}(A)}$$

[CONS]
$$\frac{}{h :^{2} A, t :^{2} \mathsf{L}(A), d :^{1} \Diamond \vdash \mathsf{Cons}(h, t)@d : \mathsf{L}(A)}$$

[LIST-ELIM]
$$\frac{\Gamma \vdash e_1 : A' \quad \Gamma, h :^{i_h} A, t :^{i_t} \mathsf{L}(A), d :^{i_d} \Diamond \vdash e_2 : A'}{\Gamma, x :^{\min(i_h, i_t, i_d)} \mathsf{L}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Nil} \to e_1 | \mathsf{Cons}(h, t)@d \to e_2 : A'}$$

Moreover, we can give a guarantee that the result is represented without internal sharing if the same holds for all parameters with aspect 2 and there is no sharing between them.

The guarantees associated with the usage aspects in the original motivating interpretation are a special case of the above interpretation when the preconditions are strengthened to allow sharing only between and within parameters with aspect 3 and nowhere else. Then variables with aspect 3 are truly separated from the result.

Our more encompassing interpretation is required when proving the soundness of the [LET] rule. E.g. consider $\mathsf{let}\ x = y\ \mathsf{in}\ (\mathit{even}(x), y)$ and see that in the subterm $(\mathit{even}(x), y)$ the variables x, y share despite having usage aspects $3, 2$, respectively.

In order to formalise this interpretation succinctly, we introduce more notation. New operators rg, sep, lin, pres stand for region, separation, (linear) internal separation and preservation, respectively. They are defined in the context of a typing judgement $\Gamma \vdash e : A$ with usage aspects and an evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ as follows:

$$\begin{aligned}
\mathrm{rg}(x) &= R_{\Gamma(x)}(S(x), \sigma),\\
\mathrm{sep}(x, y) &\equiv \mathrm{rg}(x) \cap \mathrm{rg}(y) = \emptyset, \quad \mathrm{sep}(T, T') \equiv \textstyle\bigwedge_{x \in T, x' \in T', x \neq x'} \mathrm{sep}(x, x')\\
\mathrm{lin}(x) &\equiv (\exists a)\big(S(x) \Vdash^{\sigma}_{\Gamma(x), \circ\circ}\, a\big), \quad \mathrm{lin}(T) = \textstyle\bigwedge_{x \in T} \mathrm{lin}(x)\\
\mathrm{pres}(x) &\equiv (\forall \ell \in \mathrm{rg}(x))\,\big(\sigma(\ell) = \sigma'(\ell)\big), \quad \mathrm{pres}(T) = \textstyle\bigwedge_{x \in T} \mathrm{pres}(x)
\end{aligned}$$

In the same context, the symbol r is treated as a special variable representing the result on the heap σ', i.e. $S(r) = v$, $\Gamma(r) = A$ and in relation to r the heap σ' is used instead of σ. Thus one can say, e.g. $\mathrm{lin}(r)$ or $\mathrm{sep}(x, r)$.

When it is not clear which heap(s), environment or result value these operators refer to, we will state them explicitly after the assertion like this: $\mathrm{sep}(x, y) \dashv S, \sigma$ or $\mathrm{pres}(x) \dashv S, \sigma, v, \sigma'$. Types of variables are usually clear from the context.

For a typing context Γ with usage aspects, we let $|\Gamma|_i$ denote the set of those variables in Γ that have aspect i. Let $|\Gamma|_{i,j}$ stand for the union of $|\Gamma|_i$ and $|\Gamma|_j$.

Definition 1. *A typing judgement $\Gamma \vdash e : A$ with usage aspects is* sound *if*

$$\left.\begin{array}{l}
- \ S, \sigma \vdash e \rightsquigarrow v, \sigma',\\
- \ S \Vdash^{\sigma}_{\Gamma, \infty}\, \eta,\\
- \ \mathrm{sep}(|\Gamma|_1, |\Gamma|) \ \textit{and}\\
- \ \mathrm{lin}(|\Gamma|_1)
\end{array}\right\} \Longrightarrow \left\{\begin{array}{l}
- \ v \Vdash^{\sigma'}_{A, \infty}\, [\![e]\!]_{\eta, [\![P]\!]},\\
- \ \mathrm{pres}(|\Gamma|_{2,3}),\\
- \ \mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma|_{1,2}) \ \textit{and}\\
- \ \mathrm{lin}(r) \ \textit{if also}\ \mathrm{lin}(|\Gamma|_2) \ \textit{and}\ \mathrm{sep}(|\Gamma|_2, |\Gamma|_2).
\end{array}\right.$$

Definition 2. *A typing rule is* sound *if whenever its premises are sound, so is its conclusion.*

Theorem 1. *All the typing rules in Fig 3 are sound.*

The theorem is fully proved in a paper in preparation, joined with Aspinall and Hofmann. Here, we show only a proof for one of the rules for illustration. A condensed proof of soundness for the most intricate rule [LET] and some others can be found in [12].

Fig. 4. Soundness of [CART-INTRO]

(Op)	$S, \sigma \vdash (e_1, e_2) \rightsquigarrow v, \sigma''$	
(R)	$S \Vdash^{\sigma}_{\Gamma_1 \wedge \Gamma_2, \infty} \eta$	
(S)	$\mathrm{sep}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_1, \lvert\Gamma_1 \wedge \Gamma_2\rvert)$	$\dashv S, \sigma$
(L)	$\mathrm{lin}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_1)$	$\dashv S, \sigma$
(Op12)	$S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma' \ \wedge\ S, \sigma' \vdash e_2 \rightsquigarrow v_2, \sigma'' \ \wedge\ v = (v_1, v_2)$	
(R1)	$S_1 \Vdash^{\sigma}_{\Gamma_1, \infty} \eta_1$ where $S_1 = S\vert_{\Gamma_1}, \eta_1 = \eta\vert_{\Gamma_1}$	
(SL1)	$\mathrm{sep}(\lvert\Gamma_1\rvert_1, \lvert\Gamma_1\rvert) \ \wedge\ \mathrm{lin}(\lvert\Gamma_1\rvert_1)$	$\dashv S_1, \sigma$
(D1)	$v_1 \Vdash^{\sigma'}_{A_1, \infty} \llbracket e_1 \rrbracket_{\eta_1, \llbracket P \rrbracket}$	
(P1)	$\mathrm{pres}(\lvert\Gamma_1\rvert_{2,3})$	$\dashv S_1, \sigma, v_1, \sigma'$
(C1)	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_1\rvert_{1,2})$	$\dashv S_1, \sigma, v_1, \sigma'$
(Lr1)	$\mathrm{lin}(r) \impliedby \mathrm{lin}(\lvert\Gamma_1\rvert_2) \ \wedge\ \mathrm{sep}(\lvert\Gamma_1\rvert_2, \lvert\Gamma_1\rvert_2)$	$\dashv S_1, \sigma, v_1, \sigma'$
(R2')	$S_2 \Vdash^{\sigma}_{\Gamma_2, \infty} \eta_2$ where $S_2 = S\vert_{\Gamma_2}, \eta_2 = \eta\vert_{\Gamma_2}$	
(R2)	$S_2 \Vdash^{\sigma'}_{\Gamma_2, \infty} \eta_2$	
(SL2)	$\mathrm{sep}(\lvert\Gamma_2\rvert_1, \lvert\Gamma_2\rvert) \ \wedge\ \mathrm{lin}(\lvert\Gamma_2\rvert_1)$	$\dashv S_2, \sigma$
(D2)	$v_2 \Vdash^{\sigma''}_{A_2, \infty} \llbracket e_2 \rrbracket_{\eta_2, \llbracket P \rrbracket}$	
(P2)	$\mathrm{pres}(\lvert\Gamma_2\rvert_{2,3})$	$\dashv S_2, \sigma', v_2, \sigma''$
(C2)	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_2\rvert_{1,2})$	$\dashv S_2, \sigma', v_2, \sigma''$
(Lr2)	$\mathrm{lin}(r) \impliedby \mathrm{lin}(\lvert\Gamma_2\rvert_2) \ \wedge\ \mathrm{sep}(\lvert\Gamma_2\rvert_2, \lvert\Gamma_2\rvert_2)$	$\dashv S_2, \sigma', v_2, \sigma''$
(D1')	$v_1 \Vdash^{\sigma''}_{A_1, \infty} \llbracket e_1 \rrbracket_{\eta, \llbracket P \rrbracket}$	
(D)	$(v_1, v_2) \Vdash^{\sigma''}_{A_2, \infty} \llbracket e_2 \rrbracket_{\eta_2, \llbracket P \rrbracket}$	
(P)	$\mathrm{pres}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_{2,3})$	$\dashv S, \sigma, v, \sigma''$
(C)	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_{1,2})$	$\dashv S, \sigma, v, \sigma''$
(Lr)	$\mathrm{lin}(r) \impliedby \mathrm{lin}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_2) \ \wedge\ \mathrm{sep}(\lvert\Gamma_1 \wedge \Gamma_2\rvert_2, \lvert\Gamma_1 \wedge \Gamma_2\rvert_2)$	$\dashv S, \sigma, v, \sigma''$

Proposition 1. *The* [CART-INTRO] *typing rule is sound.*

Proof. Let us state the rule again for convenience:

[CART-INTRO]
$$\frac{\Gamma_1 \vdash e_1 : A_1 \quad \Gamma_2 \vdash e_2 : A_2 \quad \forall z \in \lvert\Gamma_1\rvert \cap \lvert\Gamma_2\rvert. \Gamma_1[z], \Gamma_2[z] \geq 2}{\Gamma_1 \wedge \Gamma_2 \vdash (e_1, e_2) : A_1 \times A_2}$$

The main intermediate statements that arise in course of the proof are listed in Fig. 4. The list starts with the four assumptions and ends with the four conclusions from Def. 1 applied to the concluding judgement of the rule. Now we explain why each statement follows from the preceding ones.

Assume *(Op,R,S,L)*. From the operational semantics rule [CART-INTRO] and *(Op)* we get *(Op12)*. Statements *(R1,SL1)* are simple projections of *(R,S,L)* and together with *(Op12)* form the four conditions of the definition of soundness for the first premise. Thus we get the four conclusions *(D1,P1,C1,Lr1)*.

By another projection of *(R)* we get *(R2')*. To get *(R2)* as well as *(SL2)*, we need to show that none of the values referenced in S_2 has been modified during the evaluation of e_1. This follows from $\mathrm{sep}(|\Gamma_1|_1, |\Gamma_2|)$ (which is a consequence of *(S)*), *(P2)*, Lemma 1 (3) and the side condition that Γ_2 does not contain any variable which has aspect 1 in Γ_1. Now we get by the soundness of the second premise and *(Op12,R2,SL2)* the conclusions *(D2,P2,C2,Lr2)*.

By *(C1)*, the region $R_{A_1}(v_1, \sigma')$ has not been modified by the evaluation of e_2 thanks to $\mathrm{sep}(|\Gamma_2|_1, |\Gamma_2|)$ (a consequence of *(S)*). Thus we get *(D1')* which together with *(D2)* yields *(D)*.

From *(P1,P2)* and *(C1,C2)* it is easy to deduce *(P)* and *(C)*, respectively. Assuming the extra conditions in *(Lr)*, hold on σ then the analogous conditions holds for Γ_1 and Γ_2 on their respective heaps σ and σ'—the conditions of *(Lr1)* and *(Lr2)* are satisfied and we get that both v_1 and v_2 represent values on σ'' with internal separation which concludes the proof of the last statement *(Lr)*. $\square$

3.2 Inference

Imagine that a LFPL program is written which type-checks using the plain non-linear typing. Next, we will show a way to find out whether the program can be soundly annotated with usage aspects. Moreover, we will be able to find a kind of principal annotations for typable programs giving its strongest sound usage aspects.

The typing rules are formulated in such a way that type-checking of terms can be viewed as a combination of the ordinary type-checking of (non-linear) LFPL and the inference of usage aspects. Thus we take the approach of performing ordinary type-checking and deriving the usage aspects along the way.

Our tasks would be trivial if the typing system was strongly deterministic. There would be only one path of type-checking and it would either fail or succeed. If succeeded, it would give the only derivable usage aspects as a by-product. Nevertheless, the typing is not deterministic because of the rules [WEAK], [DROP] and [RAISE].

Fortunately, we can still specify a greedy deterministic type-checking strategy which type-checks all typable terms. Dealing with [WEAK] is standard. The two rules [DROP] and [RAISE] would be trivial without the usage aspects. They introduce nondeterminism to the type-checking by allowing the annotation to be weakened at any time and strengthened at certain situations. Our deterministic strategy consists in:

- giving [RAISE] priority over all other rules,
- using [WEAK] and [DROP] only if necessary to make two premises of another rule match each other.

We need to show that all rules are stable and monotone in a precise sense to be able to see that this strategy is optimal. To that end, let us first introduce an order on annotations.

Let the notation $\Gamma \leq \Gamma'$ mean that the annotated contexts Γ, Γ' do not differ apart from their usage aspects and it holds $\Gamma[x] \leq \Gamma'[x]$ for each $x \in |\Gamma|$.

Extend this order point-wise to annotated programs: $P \leq P'$ if the two programs differ only in their usage aspect annotations and for every $f \in \mathrm{Dom}(P)$ it holds $\Gamma_f \leq \Gamma'_f$ where $P(f) = \Gamma_f \vdash e_f : A_f$ and $P'(f) = \Gamma'_f \vdash e_f : A_f$.

This syntactical order on annotations agrees with the logical strength order:

Lemma 2. *If $\Gamma \leq \Gamma'$ and $\Gamma' \vdash e : A$ is sound, then $\Gamma \vdash e : A$ is sound.*

Proof. This follows directly from Def. 1 because the pre-conditions $\mathrm{sep}(|\Gamma|_1, |\Gamma|)$ and $\mathrm{lin}(|\Gamma|_1)$ as well as $\mathrm{lin}(|\Gamma|_2)$ and $\mathrm{sep}(|\Gamma|_2, |\Gamma|_2)$ weaken when aspects grow while the guarantees $\mathrm{pres}(|\Gamma|_{2,3})$, $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma|_{1,2})$ and $\mathrm{lin}(r)$ strengthen when aspects grow. □

As a result of this lemma, we know that there is a truly strongest annotation for each typing judgement, namely the one with aspect 3 for all parameters.

Definition 3. *A typing rule is* stable and monotone *if for each instance*

$$\frac{\Gamma_1 \vdash e_1 : A_1, \ldots, \Gamma_n \vdash e_n : A_n}{\Gamma \vdash e : A}$$

and any $\Gamma'_1 \geq \Gamma'_1$, ..., $\Gamma'_n \geq \Gamma_n$ there exists $\Gamma' \geq \Gamma$ such that the following is another instance

$$\frac{\Gamma'_1 \vdash e_1 : A_1, \ldots, \Gamma'_n \vdash e_n : A_n}{\Gamma' \vdash e : A}.$$

Proposition 2. *All the typing rules in Fig 3 are stable and monotone.*

The proof is an easy combinatorial exercise.

Now we can see that our type-checking strategy is optimal: If we follow another strategy, we either unnecessarily weaken the judgement or miss an opportunity to strengthen it (this follows from the fact that [RAISE] and [DROP] do indeed strengthen or weaken the judgement by Lemma 2). Stability and monotonicity of the rules then guarantee that such losses in strength cannot lead to a strengthening which would be missed by the greedy strategy.

We are now able infer the strongest annotations for terms given some annotations of all functions used in the term. The next step is to infer strongest annotation for a whole unannotated program P as follows:

- Annotate every parameter of every function in P with the strongest usage aspect 3 and thus get an annotated program P_0.
- Using this annotation for function symbols, infer usage aspects for all the functions in P if possible to get an annotated program P_1.
- Use annotation of P_i to infer P_{i+1} if possible.
- Repeat until $P_k = P_{k+1}$ holds and return P_k.

This process may fail if some of the terms cannot be annotated. We will show that in this case the program is not sound.

Assume that the program is sound. Take any sound annotation P' of P. This means that when inferring annotation for P using the annotations from P' we get an annotated program P'' with $P'' \geq P'$.

It has to hold $P' \leq P_0$. By stability and monotonicity of the typing rules and thus the typing of terms, we get that our algorithm will not fail but successfully produce all P_i's and it holds $P_i \geq P'' \geq P'$. Thus the same holds for an eventual P_k. This also means that P_k, if returned, is the strongest sound annotation.

From the stability and monotonicity and $P_1 \leq P_0$, we get $P_{i+1} \leq P_i$ for every i. Since there are only finitely many annotations for a program, the process will finish after finitely many steps. In fact, the number of steps is at most three times the total number of parameters in the program. Altogether, the runtime of the algorithm is asymptotically linear in the size of the program.

4 Beyond Usage Aspects

We can view also other annotation-based typings of LFPL as inferring conditions and guarantees about the heap layout of the parameters and the result. We will outline how this can be done with the language of Atkey [3] in Subsect. 4.1. Another such language designed by the present author [11] will be briefly described in Subsect. 4.2. This language is inspired by the analysis of usage aspects in Sect. 3 as pre-conditions and rely-guarantees (i.e. guarantees that rely on further pre-conditions) appear in it very explicitly.

For the language of Atkey, it was not possible to generalise the inference annotation algorithm because of non-existent strongest annotations for some terms. The second language is designed so that the inference algorithm from Subsect. 3.2 would apply. Nevertheless, its quadratic size of annotations makes the inference run asymptotically slower than linearly in the size of the program.

In general, if for an annotation-based typing of LFPL

- all typing rules are sound, stable and monotone,
- for every signature there exists a strongest annotation,
- for every annotated program there is an algorithm inferring strongest annotation for terms using the signature of the program,

then the iterative process described in Subsect. 3.2 infers the strongest annotation for any sound program P.

4.1 Explicit sharing

During the research that led to this paper, other typings for LFPL extending usage aspects were designed. Namely, Robert Atkey formulated a typing for LFPL in [3] that adds *explicit* pre-conditions about *sharing* among parameters in addition to the usage aspects. This contrasts with the system of Aspinall and Hofmann in which separation pre-conditions are deduced from usage aspects (as described in Subsect. 3.1) and are therefore tightly bound to the usage guarantees.

The syntax of a typing judgement in [3] is $\Gamma \vdash e : A, S, D$ where

- Γ contains assumptions of the form $x : (A_x, S_x)$
- S_x lists parameters that x is allowed to share with
- S lists parameters which may share with the result (aspect 2)
- D lists parameters which may get destroyed (aspect 1)

For example, *append* can be typed as follows:

$$x : (\mathsf{L}(A), \emptyset), y : (\mathsf{L}(A), \emptyset) \vdash \mathit{append}(x, y) : \mathsf{L}(A), \{y\}, \{x\}$$

The usage aspects are encoded using a different method. More importantly, they retain only their meaning as a guarantee. Separation pre-condition is expressed independently via a symmetrical anti-reflexive relation on the context which is encoded in the judgement via the S_x sets. This gives more flexibility to the typing despite maintaining the invariant:

$$y \in S_x \Longrightarrow ((x \in S \Longrightarrow y \in S) \wedge (x \in D \Longrightarrow y \in D)) \tag{1}$$

(I.e. parameters with different usage aspects cannot share.)

More formally, the annotation's meaning can be expressed as the following assertions (using the notation from Subsect. 3.1):

$$\begin{array}{rl} \text{condition:} & \bigwedge_{(x,S_x)\in\Gamma} \operatorname{sep}(x, |\Gamma| \setminus S_x) \wedge \operatorname{lin}(x) \\ \text{guarantee:} & \operatorname{pres}(|\Gamma| \setminus D) \wedge \operatorname{sep}(r, |\Gamma| \setminus (S \cup D)) \wedge \operatorname{lin}(r) \end{array}$$

The invariant (1) allows Atkey to formulate the following typing rule for let:

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

which is much simpler and easier to prove sound than [LET] from Fig. 3.

The extra flexibility leads to even more sound terms being type-checked but results in a more complex annotation inference algorithm. The complexity of type-checking seems to be inherent and caused by the fact that some typing judgements do not have any strongest correct annotation but a set of several maximal ones. For example, $x : A, y : A' \vdash x : A$ can be annotated in two incomparable ways:

$$x : (A, \emptyset), y : (A', \emptyset) \vdash x : A, \{x\}, \emptyset$$
$$x : (A, \{y\}), y : (A', \{x\}) \vdash x : A, \{x, y\}, \emptyset$$

Maybe this problem can be alleviated by a more subtle interpretation of the annotation which would make the second judgement weaker than the first one.

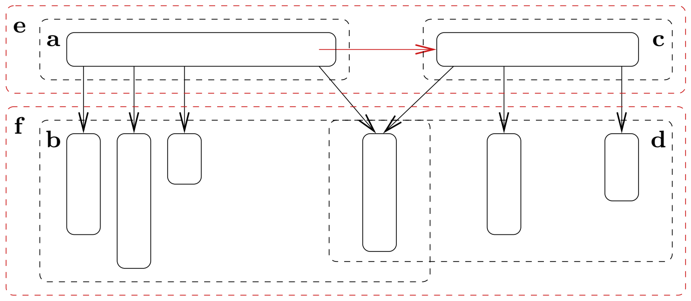


Fig. 5. Layers in *append*

Fig. 6. Layered typing of *append*

$x : \mathsf{L}^{[\mathbf{a}]}(\mathsf{L}^{[\mathbf{b}]}(\mathsf{Bool})), y : \mathsf{L}^{[\mathbf{c}]}(\mathsf{L}^{[\mathbf{d}]}(\mathsf{Bool}));$	parameter layers
$\{\mathbf{a}\otimes\mathbf{b}, \mathbf{a}\otimes\mathbf{c}, \mathbf{a}\otimes\mathbf{d}\} \vdash$	separation pre-condition
$\mathit{append}(x, y) : \mathsf{L}^{[\mathbf{e}\subseteq\{\mathbf{a},\mathbf{c}\}]}(\mathsf{L}^{[\mathbf{f}\subseteq\{\mathbf{b},\mathbf{d}\}]}(\mathsf{Bool}));$	result layers, containment
$\{\mathbf{a}\};$	destroyed layers
$\otimes\mathbf{f}/\mathbf{e} \Longleftarrow \{\mathbf{b}\otimes\mathbf{d}, \otimes\mathbf{b}/\mathbf{a}, \otimes\mathbf{d}/\mathbf{c}\}$	separation rely-guarantees

4.2 Layered sharing

Next, we outline the main ideas of the LFPL typing described in [11] and show how the system arose from the view of usage aspects as conditions and guarantees presented above.

In both of the typings shown earlier, one implicitly or explicitly derives the soundness pre-condition for *append*(x, y) that x and y have to be completely separated on the heap. This condition is unnecessarily strong because *append* is sound even when the elements of the two lists share with each other. The sufficient and necessary pre-condition of soundness for *append* is indeed that the *top-level cons-cells* of x (marked **a** in Fig.5) do not share heap space with y (marked **c** and **d**). The new system arose from the need to distinguish between different layers[3] of datatypes in some situations. Layered sharing has developed from the idea that usage aspects should be assigned to individual layers instead of the value as a whole.

Let us now describe the annotations within the typing of *append* shown in Fig. 6. Firstly, *layer names* $(\mathbf{a}, \mathbf{b}, \ldots)$ appear inside the types within the context as well as in the result type.

Like in Atkey's language, an explicit *separation pre-condition* is added to the typing context. It takes the form of a set of *basic separation assertions.* A prime example of a basic separation assertion is $\mathbf{a}\otimes\mathbf{b}$ which means that the two

[3] Layers are called portions in [11].

layers are separated. There is another kind of basic separation assertion which is introduced below.

An equivalent of the usage aspect 1 is the set of *destroyed layers* shown towards the end of the judgement. Equivalent to the distinction between aspects 2 and 3 are the containment guarantees shown next to the layer names within the result type (e.g. $\mathbf{e}{\subseteq}\{\mathbf{a},\mathbf{c}\}$). Notice that this containment is unrelated to whether the layers $\mathbf{a}$ and $\mathbf{c}$ are destroyed during the evaluation.

A new aspect of this typing are the *explicit separation rely-guarantees* consisting of a sequence of implications like $\otimes\mathbf{f}/\mathbf{e} \Longleftarrow \{\mathbf{b}{\otimes}\mathbf{d}, \otimes\mathbf{b}/\mathbf{a}, \otimes\mathbf{d}/\mathbf{c}\}$ which mean that the result satisfies the given basic separation assertion ($\otimes\mathbf{f}/\mathbf{e}$ in this case—its meaning will be explained shortly) on condition that all the given basic separation assertions ($\mathbf{b}{\otimes}\mathbf{d}, \otimes\mathbf{b}/\mathbf{a}, \otimes\mathbf{d}/\mathbf{c}$ in this case) hold for the parameters.

The basic separation assertion $\otimes\mathbf{f}/\mathbf{e}$ means that within any list marked by $\mathbf{e}$ all instances of the layer $\mathbf{f}$ are separated from each other. More specifically, the elements of the list do not share with each other. Such assertions do sometimes appear in the main separation pre-condition.

In [11], the idea of layered datatype sharing is formalised and generalised for arbitrary inductive datatypes. It is also shown there that the inference algorithm applies to the resulting typing system. An implementation of the inference algorithm exists and should be soon made available through the web interface [10].

5 Conclusion

We have extracted and made explicit the ideas that were behind the original design of usage aspects and thus managed to reformulate their meaning in a way in which it is manageable to prove the soundness of their typing rules and to design a simple annotation inference algorithm. We have also outlined from this perspective Atkey's early typing and given some hints for its further study. Lastly, we previewed a powerful typing for layered datatype sharing which has been formed as a result of this study.

We conjecture that also $\alpha\lambda$-calculus [15] which arises from the logic of Bunched Implications [13] can be interpreted in the present approach similarly to Atkey's typing but without considering preservation guarantees. This is very interesting because $\alpha\lambda$-calculus is a higher-order language.

The present approach might be beneficial in extending the usage aspects to higher order too. A suitable extension of the operational semantics with explicit allocation of closures has been suggested in [3] and [2]. The leading idea for designing the typing is that every function type constructor in the judgement has to be treated like a typing judgement (featuring its captured context). Thus a function type should be explicitly or implicitly annotated with a subset of possible annotations of the associated typing judgement.

The use of pre- and post-conditions for certifying in-place update with sharing is not new. Recent work includes *Alias types* [19, 20] which have been designed to express when heap is manipulated type-safely in a typed assembly language.

Alias types express properties about heap layout and could be considered as representations of our assertions. Unfortunately, they cannot express that two heap location variables *may* have the same value. In an alias type, two different location variables always take different values.

Separation Logic [18] may serve a similar purpose as alias types but for higher level imperative languages. We believe that it is expressive enough to encode the assertions present in the three systems which we described in this paper. Nevertheless, the encoding of layers would not be very natural.

There are plenty more studies related to certifying in-place update which we cannot possibly cover here.

The novelty of LFPL and its extensions is that one can write in them certified in-place update algorithms that evaluate in accordance with a simple functional denotational semantics. The new resource type $\Diamond$ makes in-place update explicit in a functional setting. In the annotated typings of LFPL presented here any *constructor* argument of type $\Diamond$ is given a "destroyed" aspect. Thus by distinguishing different ways in which the resources are used, the typing is made more flexible. The typings with usage aspects and layered sharing moreover feature a deterministic type-checking via usage aspect annotation inference thus liberating a programmer from writing any annotation in their programs. Complementary aid is given to an LFPL programmer by an automatic inference of constructor arguments of type $\Diamond$ [9, 8].

Acknowledgements. This research has been supported by the EPSRC grant GR/N28436/01. The author is grateful to David Aspinall and Robert Atkey for discussion and comments on this work. Many thanks go also to the anonymous referee who provided valuable hints on how to make the material easier to read.

References

1. David Aspinall and Martin Hofmann. Another type system for in-place update. In D. Le Métayer, editor, *Programming Languages and Systems, Proceedings of 11th European Symposium on Programming*, pages 36–52. Springer-Verlag, 2002. Lecture Notes in Computer Science 2305.
2. Robert Atkey. First year progress report and thesis proposal: Type systems with explicit sharing. Available from: http://www.dcs.ed.ac.uk/home/roba, August 2002.
3. Robert Atkey. LFPL with explicit sharing and destruction. An unpublished draft, June 2002.
4. Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.
5. Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*, pages 464–476. Computer Society Press, 1999.
6. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
7. Martin Hofmann. The strength of non size-increasing computation. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

8. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '03)*, January 2003.
9. Steffen Jost. Static prediction of dynamic space usage of linear functional programs. Master's thesis, Technische Universität Darmstadt, Fachbereich Mathematik, 2002.
10. C. Kirkegaard, R. Atkey, M. Konečný, D. Aspinall, and M. Hofmann. Prototype compilers with resource-bounded type systems. Available from: http://www.dcs.ed.ac.uk/home/resbnd/prototypes/, 2000-2003.
11. Michal Konečný. LFPL with types for deep sharing. Technical Report EDI-INF-RR-157, LFCS, Division of Informatics, University of Edinburgh, October 2002.
12. Michal Konečný. Typing with conditions and guarantees in LFPL. Technical Report EDI-INF-RR-0151, LFCS, Division of Informatics, University of Edinburgh, October 2002.
13. P. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–243, 1999.
14. P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228:211–252, 1999.
15. Peter W. O'Hearn. On bunched typing. To Appear in the Journal of Functional Programming, 2002.
16. John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM Press, 1978.
17. John C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium*, pages 704–722. Springer-Verlag, 1989. Lecture Notes in Computer Science 372.
18. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.
19. David Walker and Greg Morrisett. Alias types. In *ESOP 2000*, pages 366–381, 2000. Lecture Notes in Computer Science 1782.
20. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Types in Compilation 2000*, pages 177–206, 2001. Lecture Notes in Computer Science 2071.