Functional In-place Update with Layered Datatype Sharing

Michal Konečný

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

Abstract. Hofmann's LFPL is a functional language with constructs which 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 this semantics is achieved by a linear typing. We introduce a non-linear typing of first-order LFPL programs which is more permissive than the recent effect-based typing of Aspinall and Hofmann. The system efficiently infers separation assertions as well as destruction and re-use effects for individual layers of recursive-type values. Thus it is suitable for in-place update algorithms with complicated data aliasing.

1 Introduction

First-order LFPL (Linear Functional Programming Language) [1] is a functional programming language which has a straightforward compositional denotational semantics and at the same time can be evaluated imperatively without dynamic memory allocation. The higher-order version of this language is interesting among other reasons because it captures non-size increasing polynomial time/space computation with primitive/full recursion [2]. We focus on the first order version of LFPL in this paper.

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

$$\begin{aligned} append_A(x,y) = \ & \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \Rightarrow y \\ & \mid \mathsf{cons}(h,t)@d \Rightarrow \mathsf{cons}(h, append_A(t,y))@d \end{aligned}$$

In the denotational semantics of the program, the attachment $@d$ of cons is (virtually) ignored in both cases and thus the semantics is list concatenation as expected. Nevertheless, in the operational semantics $@d$ indicates that the cons-cell is (or will be) at location d on the heap. Thus the above program appends the lists in-place, rewriting the first list's end with a reference to the second list. This amounts to changing its last cons-cell if the first list is not empty. The other cons-cells of the first list are overwritten with the same content.

This evaluation strategy can go wrong easily, for example for $append_A(x,x)$ with a non-empty list x. Such terms might fail to evaluate or evaluate with incorrect results. We will call a term *sound* if it evaluates in harmony with its

denotational semantics independently of the values of its free variables and how they are represented on the heap.

We also need a finer notion of soundness relative to some *extra condition* on the representation of the arguments on the heap. For example, we would like to declare $append_A(x, y)$ correct under certain condition on how lists x and y are represented. A necessary and sufficient condition for $append_A(x, y)$ to evaluate correctly is that the heap region occupied by the cons-cells of x should be separated (disjoint) from the whole region of y.

Apart from stating conditions for correctness, we need to mark certain *guarantees* about the heap representation of the result and the change of heap during the evaluation. For example, $append_A(x, y)$ preserves the value of y and this might be crucial for showing the correctness of a bigger term of which this is a subterm.

LFPL uses linearity to achieve soundness of its terms. This means that LFPL maintains the "single pointer property". This implies the precondition that different variables always refer to disjoint regions on the heap.

It is impossible to achieve a full characterisation of semantical correctness of the first-order LFPL evaluation by a simple typing system. Nevertheless, one can improve LFPL by relaxing its linearity in order to recognise more of the correct algorithms. This means losing the single pointer property and considering sharing. One such system, which we call UAPL, is provided by Aspinall and Hofmann [3]. It assigns a *usage aspect* to each variable in the typing context.

UAPL still rejects many correct programs, e.g. those with $append_A(x, y)$ where the whole regions of x and y are not guaranteed to be disjoint. UAPL needs this stronger precondition because it cannot distinguish between the cons-cell (shallow) and data (deep) levels of a list. This paper develops a typing for the underlying language of LFPL which improves over UAPL in that it can distinguish between these two levels (and more than that) for arbitrary recursive types (with no restriction on nesting).

More precisely, we note each place in a type term which may involve a heap location in the operational representation of the values of the type. For example, there is one such place within each list type constructor. We give names (ranged over by ζ) to all such places and mark them in the type (e.g. $\mathsf{L}^{[\mathbf{a}]}(\_)$). To each such place we logically associate a portion of the heap region taken by a value of the type (e.g. the locations of the cons-cells of a list).

We formulate preconditions and rely-guarantees using assertions about the portion names given in the types within a typing judgement. The most straightforward atomic assertion is $\zeta_1 \otimes \zeta_2$ which stands for the separation (disjointness) of two portions ζ_1, ζ_2 on the heap[1]. We need other atomic separation assertions related to the unfolding of a recursive type. For example, we need to express that a list has its elements separated from each other or that a tree's skeleton is laid out without overlapping.

For this reason we give names also to each occurrence of a recursive type constructor within the types of a typing judgement. For a list or tree constructor

[1] A set of such assertions corresponds to a *may-alias relation* in alias analysis [4]

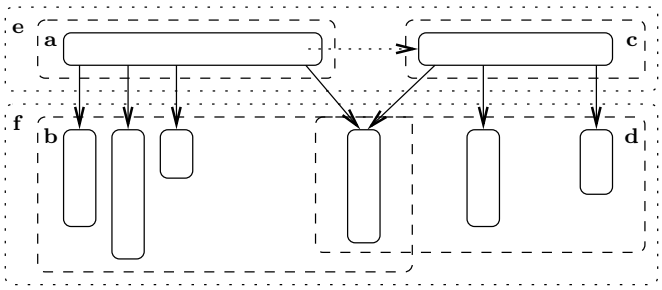

Fig. 1. Heap portions in the *append* program

we will simply reuse the name of the only portion associated with it. Given a recursive type occurrence name ζ_R and a portion name ζ which occurs within the scope of ζ_R, we will define two assertions $\otimes\zeta/\zeta_R$ and $\otimes\zeta\curlywedge\zeta_R$. The former one states that the sub-portions of ζ which correspond to the unfoldings of the type ζ_R are pairwise disjoint. The latter one is the same but requires disjointness only for pairs of unfoldings in which one is not a sub-unfolding of the other (see Def. 4 for details).

Now we can express that the elements of a list of the type $\mathsf{L}^{[\mathbf{a}]}(\mathsf{L}^{[\mathbf{b}]}(\mathsf{Bool}))$ do not share by $\otimes\mathbf{b}/\mathbf{a}$ and that the skeleton of a tree of the type $\mathsf{T}_{\mathsf{b}}^{[\mathbf{c}]}(\mathsf{Bool})$ has no confluences by $\otimes\mathbf{c}\curlywedge\mathbf{c}$.

The typing for $append_A$ in our new system would be:

$$x : \mathsf{L}^{[\mathbf{a}]}(A^{[\mathbf{b}]}), y : \mathsf{L}^{[\mathbf{c}]}(A^{[\mathbf{d}]}); \{\mathbf{a}\otimes\mathbf{b}, \mathbf{a}\otimes\mathbf{c}, \mathbf{a}\otimes\mathbf{d}\} \vdash$$
$$append_A(x, y) : \mathsf{L}^{[\mathbf{e}\subseteq\{\mathbf{a},\mathbf{c}\}]}(A^{[\mathbf{f}\subseteq\{\mathbf{b},\mathbf{d}\}]}); \{\mathbf{a}\}; \otimes\mathbf{f}/\mathbf{e} \Longleftarrow \{\mathbf{b}\otimes\mathbf{d}, \otimes\mathbf{b}/\mathbf{a}, \otimes\mathbf{d}/\mathbf{c}\}$$

The participating heap portions are illustrated in Fig. 1. Just before $\vdash$ there is a precondition demanding that three pairs of argument portions should be separated. Whenever this condition holds, the term will evaluate in harmony with the obvious denotational semantics. The result type has also two portions, named $\mathbf{e}$ and $\mathbf{f}$. The notation with $\subseteq$ indicates the guarantee that the corresponding result portion is contained within the union of the given argument portions. Behind the result type is the set of possibly destroyed portions, i.e. $\{\mathbf{a}\}$ in this case. The last guarantee is a list of implications giving a set of separation preconditions necessary for each basic separation guarantee about the result. The expression $\otimes\mathbf{f}/\mathbf{e}$ indicates internal separation between individual elements of the resulting list which depends on the same property for both argument lists and the separation of $\mathbf{b}$ from $\mathbf{d}$.

The containment and destruction-limitation guarantees in our system play a role analogous to that of usage aspects in UAPL.

Our motivating example of an algorithm which makes use of the distinction between shallow and deep levels is an in-place update binary tree reversal program $paths_A$ converting an A-labelled binary tree to the list containing for each leaf the list of node labels as read along the path from the leaf to the root. The

program can be found in the report version of this paper [5] (as well as many other details omitted here). Automatically inferred annotation for $paths_{\mathsf{L(Bool)}}$ can be found in Sect. 6.

In the operational point of view, the program $paths_A(t)$ simply reverses all the pointers in the structure of the tree t and turns its leaves into cons-cells of the resulting list. It is important here that leaves are not heap-free and their locations can be reused. Consequently, any labels on leaves are ignored.

The crucial point in the algorithm is when after recursive calls for the two branches of a tree, two intermediate result lists are appended. These are lists whose spines are disjoint but whose data are lists that share among each other extensively.

2 Underlying language

First, we define the types, terms, typing judgements and denotational semantics of a language which is a version of LFPL without linearity and with arbitrary heap-aware recursive types. The types include (nested) recursive types, sums and products, unit and LFPL-like memory-resource types:

$$A ::= \Diamond \mid {}^{\Diamond}A \mid \mathsf{Unit} \mid A_1 \times A_2 \mid A_1 + A_2 \mid X \mid \mu X.A$$

where X ranges over a set of type variables. The type ${}^{\Diamond}A$ has the same values as A but in the operational semantics a value would be represented by a pointer to a heap location which contains its ordinary type A representation[2].

The pre-terms feature function calls which provide for general recursion as well as pointer manipulation operators:

$$\begin{aligned} e ::= {} & x \mid f(x_1,\ldots,x_n) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid \mathsf{unit} \mid (x_1,x_2) \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1,x_2) \Rightarrow e \\ & \mid \mathsf{inL}\,(x) \mid \mathsf{inR}\,(x) \mid \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inL}\,(x_L) \Rightarrow e_L | \mathsf{inR}\,(x_R) \Rightarrow e_R \\ & \mid \mathsf{fold}_{\mu X.A}\,(x) \mid \mathsf{unfold}_{\mu X.A}\,(x) \\ & \mid x@d \mid \mathsf{loc}\,(x) \mid \mathsf{get}\,(x) \end{aligned}$$

where x, d range over a set of variables and f over a set of function symbols.

The plain typing defines *typing judgements* of the form $\Gamma \vdash e : A$ and is standard apart from the axiom rules for the new heap-aware term constructors:

[PUT] $x : A, d : \Diamond \vdash x@d : {}^{\Diamond}A$ [GET] $x : {}^{\Diamond}A \vdash \mathsf{get}\,(x) : A$ [LOC] $x : {}^{\Diamond}A \vdash \mathsf{loc}\,(x) : \Diamond$

All types in a typing judgement are closed (without free type variables).

[2] Operationally viewed, the construction ${}^{\Diamond}A$ is analogous to Standard ML reference type A ref as well as to the pointer type construction `(A *)` in C. Unlike in SML, our constructor is transparent to the denotational semantics.

$$
\begin{array}{ll}
\mathsf{Bool} = \mathsf{Unit} + \mathsf{Unit} & \mathsf{tt} = \mathsf{inL}\,(\mathsf{unit})\,, \mathsf{ff} = \mathsf{inR}\,(\mathsf{unit}) \\
\mathsf{L}(A) = & \mathsf{nil}_A = \mathsf{fold}_{\mathsf{L}(A)}\,(\mathsf{inL}\,(\mathsf{unit})) \\
\quad \mu X.\mathsf{Unit} + {}^{\Diamond}(A \times X) & \mathsf{cons}(h,t)@d = \mathsf{fold}_{\mathsf{L}(A)}\,(\mathsf{inR}\,((h,t)@d)) \\
\mathsf{T_b}(A) = & \mathsf{leaf_b}(a)@d = \mathsf{fold}_{\mathsf{T_b}(A)}\,(\mathsf{inL}\,(a@d)) \\
\quad \mu X.{}^{\Diamond}A + {}^{\Diamond}\big(A \times (X \times X)\big) & \mathsf{node_b}(a,l,r)@d = \mathsf{fold}_{\mathsf{T_b}(A)}\,(\mathsf{inR}\,((a,(l,r))@d))
\end{array}
$$

Fig. 2. Shortcut notation for booleans, lists and trees.

A *program* P is a finite domain partial function from function symbols to typing judgements (i.e. $P(f) = \Gamma_f \vdash e_f : A_f$) which capture the signature Γ_f, A_f and the definition e_f of each symbol f.

We consider the straightforward least fixpoint denotational semantics of types as sets in which ${}^{\Diamond}(\_)$ is ignored and $\Diamond$ does not play any significant role either: $[\![\Diamond]\!] = \{\diamond\}$, $[\![{}^{\Diamond}A]\!] = [\![A]\!]$. Denotation of programs $[\![P]\!]$ and terms $[\![e]\!]_{\eta,[\![P]\!]}$ with valuation η of free variables in e is defined as usual apart from the pointer-related constructs which are virtually ignored:

– $[\![\mathsf{loc}\,(x)]\!]_{\eta,[\![P]\!]} = \diamond$ – $[\![x@d]\!]_{\eta,[\![P]\!]} = [\![\mathsf{get}\,(x)]\!]_{\eta,[\![P]\!]} = [\![x]\!]_{\eta,[\![P]\!]} = \eta(x)$

In examples, we will use a straightforward *shortcut notation* for booleans, lists and labelled binary trees as shown in Fig. 2 (elimination terms are omitted).

3 Operational semantics

In order to gain a good intuition for the annotated typing which will follow, let us first consider the details of the operational semantics to which the annotations will refer.

Data representation. The occurrences of ${}^{\Diamond}(\_)$ in types correspond to the intended heap layout of their values. For example, compare the given type of binary trees with $\mu X.{}^{\Diamond}(A + A \times (X \times X))$

Any binary tree would take the same amount of heap locations according to both of the types. A representation of a value of the above type is a pointer to a location with a value of the sum type while a value of the type in Fig. 2 is a sum value which contains a pointer to either a leaf or to a node. This difference is irrelevant at this point. It could make a difference later when a heap portion is assigned to each diamond in the type. The chosen type then allows one to divide the heap portion taken by a binary tree skeleton into two parts, one with the leaves and one with the nodes.

Let Loc be a set of *locations* which model memory addresses on a heap. We use ℓ to range over elements of Loc. Let Val be the set of (operational) *values* defined as follows:

$$v ::= \mathrm{unit} \mid \ell \mid (v_1, v_2) \mid \mathrm{inl}\,(v) \mid \mathrm{inr}\,(v)$$

An *environment* is a partial mapping $S\,\mathrm{Var} \rightharpoonup \mathrm{Val}$ from variables to operational values. A *heap* $\sigma\,\mathrm{Loc} \rightharpoonup \mathrm{Val}$ is a partial mapping from heap locations to operational values.

The way in which a denotational value a of type A is represented by an operational value v on a heap σ is formalised using a 4-ary relation $v \Vdash^{\sigma}_{A} a$ defined in the obvious way (see [5]), e.g.

$$\ell \Vdash^{\sigma}_{\Diamond} \diamond \qquad v \Vdash^{\sigma}_{\mu X.A} a \text{ if } v \Vdash^{\sigma}_{A[\mu X.A/X]} a \qquad \ell \Vdash^{\sigma}_{\Diamond A} a \text{ if } \sigma(\ell) \Vdash^{\sigma}_{A} a$$

Any heap location can be used for values of any type and thus there is no general bound on the size of a heap location. For a particular program it is desirable that such a bound could be derived statically. A simple sufficient condition for this is that all recursive types mentioned in P are bounded. We call a recursive type $\mu X.A$ *bounded* if all occurrences of X in A are within some ${}^{\Diamond}(\_)$.

Term evaluation. Using the heap representation of values, define the operational semantics by a big-step evaluation relation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ which is defined as usual (see [5]). Folding and unfolding has no operational effect.

The only heap-modifying operation is $x@d$ which overwrites a previously referenced location:

$$S, \sigma \vdash x@d \rightsquigarrow S(d), \sigma[S(d) \mapsto S(x)]$$

Heap regions. Whenever we have $v \Vdash^{\sigma}_{A} a$, we denote by $R_A(v, \sigma) \subseteq \mathrm{Dom}(\sigma)$ the straightforwardly defined complete heap region taken by v on the heap.

As we said in the introduction, we will view all recursive types as container types, i.e. we will view the region taken by a representation of a value of a recursive type $\mu X.A$ on the heap as consisting of two parts. Firstly, it is the "skeleton" layer corresponding to the locations associated with those $\Diamond$'s within A which are not within any deeper recursive type. The data layer then consist of the locations associated with all the other $\Diamond$'s within A. For example, in a value of the list type

$$\mathsf{L}(\mathsf{L}(\mathsf{Bool})) = \mu X.\mathsf{Unit} + {}^{\Diamond}\big(\big(\mu Y.\mathsf{Unit} + {}^{\Diamond}(\mathsf{Bool} \times Y)\big) \times X\big)$$

there are locations holding cons-cells of the top level list corresponding to the outer-level ${}^{\Diamond}(\_)$ and cons-cells of the element lists of the top-level list that correspond to the deeper ${}^{\Diamond}(\_)$.

Type addresses. In order to be able to make this distinction, we need to define the *portions* of a region $R_A(v, \sigma)$ corresponding to each $\Diamond$ in A. Thus we need to give formal addresses to the occurrences of $\Diamond$ in a type A, as well as the occurrences of recursive types and type variables within them. These addresses are sequences of the following symbols: L and R selecting sub-terms of sum types, $\mathsf{1}$ and $\mathsf{2}$ for product types, $\overline{\mu}_X$ for descending into a recursive type binding variable X and $\overline{\diamond}$ for descending into a boxed type. The address must end with $\overline{\diamond}$, $\overline{\mu}_X$ or $\overline{X}$

denoting an occurrence of a diamond or a box subtype, recursive type binder and a type variable, respectively. A formal definition can be found in [5].

Let $\mathrm{Addr}(A)$ be the set of all such addresses in A and let $\mathrm{AddrD}(A)$, $\mathrm{AddrR}(A)$ and $\mathrm{AddrX}_X(A)$ be its subsets consisting of the addresses which end with $\overline{\diamond}$, $\overline{\mu}_X$ and $\overline{X}$, respectively, where X is a given type variable.

For $\xi \in \mathrm{Addr}(A)$ let A_ξ be the subterm of A corresponding to the address ξ. For example, using the type $A = \Diamond \times \left(\mu Y.\mathsf{Unit} + {}^{\Diamond}Y\right)$ we get $A_{1\overline{\diamond}} = \Diamond$, $A_{2\overline{\mu}_Y} = \mu Y.\mathsf{Unit} + {}^{\Diamond}Y$ and $A_{2\overline{\mu}_Y \mathsf{R} \overline{Y}} = Y$.

Unfolded types. When it holds $v \Vdash^{\sigma}_{A} a$, we may unfold the recursive types within A arbitrarily and still yielding a valid type A' for which it holds $v \Vdash^{\sigma}_{A'} a$. Moreover, by such unfoldings we can "cover" any possible shape that a value might take on the heap. Thus we will define addresses into the infinite full unfolding of A to help us with the definition of heap portions later (see [5] for a completely formal definition).

For a closed type A, let $\mathrm{Uddr}(A) = \mathrm{Addr}(A^*)$ be the set of *unfolded addresses* of A where A^* is the infinite term obtained from A by co-inductively replacing each type variable by the recursive type which binds it. Analogously, we define $\mathrm{UddrD}(A)$ and $\mathrm{UddrR}(A)$ (there are no type variables in A^*).

Moreover, we associate to every unfolded address $\xi \in \mathrm{Uddr}(A)$ its corresponding non-unfolded address $\xi^{\mathrm{F},A} \in \mathrm{Addr}(A)$ which is a certain postfix of ξ (see [5] for details).

Assuming $v \Vdash^{\sigma}_{A} a$, for every unfolded address $\xi \in \mathrm{UddrD}(A)$ we define $v_\xi(A, v, \sigma) \in \mathrm{Dom}(\sigma)$ to be the unique location, if it exists, which we get by intuitively following the address. If that location does not exist, we let $v_\xi(A, v, \sigma)$ be undefined.

For example, consider a value of the list type $\mu X.\mathsf{Unit} + {}^{\Diamond}(A \times X)$. All the unfolded addresses $\overline{\mu}_X \mathsf{R} \left(\overline{\diamond} 2 \overline{\mu}_X\right)^n \overline{\diamond}$ for $n \in \mathbb{N}$ correspond to one and the same diamond in the type. If such a list has length m, then for every $n < m$ the address above can be associated with the heap location that contains the n-th cons-cell of this list. The rest of the addresses (i.e. for $n \geq m$) cannot be associated with any heap location.

Portions and subportions. Now we are ready to "collect" all the locations that correspond to a given occurrence of a diamond in a type. We will also define a bit more subtle sub-collection of these locations—those which are limited to a certain sub-value. The sub-value is given by an unfolded address of a recursive type which is then forbidden to unfold any more.

Definition 1. *Whenever $v \Vdash^{\sigma}_{A} a$ and $\xi \in \mathrm{AddrD}(A)$, we define the* portion *of the region $R_A(v, \sigma)$ corresponding to the unfolded type address ξ as the set*

$$P^{\xi}_{A}(v, \sigma) = \left\{ v_{\xi'}(A, v, \sigma) \,\middle|\, \xi' \in \mathrm{UddrD}(A),\ (\xi')^{\mathrm{F},A} = \xi \right\}$$

For an unfolded address $\xi_R\xi \in \mathrm{UddrD}(A)$ *with* $A_{\xi_R} = \mu X.A'$*, its* subportion along ξ_R *is the set*

$$P_A^{\xi_R\xi/\xi_R}(v,\sigma) = \left\{ v_{\xi_R\xi'}(A,v,\sigma) \,\middle|\, \xi_R\xi' \in \mathrm{UddrD}(A),\ \xi' \text{ has no } \overline{\mu}_X,\ (\xi_R\xi')^{\mathrm{F},A} = (\xi_R\xi)^{\mathrm{F},A} \right\}$$

where A' *is assumed not to contain a* bound *type variable named* X*.*

Different heap portions of a value may overlap with each other in some cases.

4 Annotations

We will add several kinds of annotations to every typing judgement $\Gamma \vdash e : A$ as shown on an example in the Introduction. We will use ζ to range over names (e.g. $\mathbf{a}, \mathbf{b}, \ldots$) from a set Nm.

Types. We first need to formalise a type labelled at its diamond and rec-type addresses. We define two syntactical constructions of annotated types, one suitable for generic treatment of the whole type and one suitable for accessing specific labels in a specific annotated type.

An annotated type is an expression $A\left[\frac{f_D}{f_R}\right]$ where A is a type and

$$f_D\ \mathrm{AddrD}(A) \to T, f_R\ \mathrm{AddrR}(A) \to T$$

are functions. Another syntax for the same annotated type involves placing all the images of f_D and f_R at appropriate places withing the type A (see [5]).

For example, annotated versions of the list and tree types are listed below together with their shortcut notation incorporating a simplification enforcing the use of the same names at certain addresses:

$$\mathsf{L}^{[\zeta]}(A) = \mu X^{(\zeta)}.\mathsf{Unit} + {}^{\Diamond[\zeta]}(A \times X)$$
$$\mathsf{T}_{\mathsf{b}}^{[\zeta]}(A) = \mu X^{(\zeta)}.{}^{\Diamond[\zeta]}A + {}^{\Diamond[\zeta]}(A \times X \times X)$$

where A stands for an annotated type in this case.

Let $\alpha(\_)$ be the obvious polymorphic forgetful mapping from annotated to plain types. All the notions defined for plain types, in particular $\mathrm{Addr}(A)$ and A_ξ, extend by analogy to annotated types.

Definition 2 (Types annotated with portion names and containment). *Let* BType *and* EType *be the sets of all annotated types of the form* $A\left[\frac{\delta}{\rho}\right]$ *and* $A\left[\frac{(\delta,\gamma)}{\rho}\right]$*, respectively where*

$$\delta\ \mathrm{AddrD}(A) \to \mathrm{Nm},\ \rho\ \mathrm{AddrR}(A) \to \mathrm{Nm},\ \gamma\ \mathrm{AddrD}(A) \to \wp(\mathrm{Nm}).$$

In the other syntax for $A\left[\frac{(\delta,\gamma)}{\rho}\right]$*, a pair* $(\zeta, \{\zeta_1, \zeta_2, \ldots\})$ *from the image of* (δ,γ) *will be usually written as* $[\zeta \subseteq \{\zeta_1, \zeta_2, \ldots\}]$*. For* $E = A\left[\frac{(\delta,\gamma)}{\rho}\right] \in \mathrm{EType}$ *let*

$$\mathrm{N_D}(E) = \mathrm{Ran}(\delta),\ \mathrm{N_R}(E) = \mathrm{Ran}(\rho),\ \mathrm{N_C}(E) = \bigcup\nolimits_{\xi \in \mathrm{AddrD}(A)} \gamma(\xi)$$

and also $\mathrm{N}(E) = \mathrm{N_D}(E) \cup \mathrm{N_R}(E)$*. All these but* $\mathrm{N_C}()$ *apply to* $B \in \mathrm{BType}$ *too.*

From now on, whenever using the symbol B or E (with sub- or super-scripts) let us implicitly assume that it stands for a type from BType or EType, respectively. In plain words, $\mathrm{N_D}(B)$ and $\mathrm{N_R}(B)$ are the sets of the names of all the diamonds and recursive type constructors, respectively, within B. For any $\xi \in \mathrm{AddrD}(B) \cup \mathrm{AddrR}(B)$, let ζ_ξ^B denote the name from Nm at the top of B_ξ (i.e. the image of δ or ρ). We will often leave the superscript B out from ζ_ξ^B. Analogously define ζ_ξ^E.

Definition 3 (Syntax of separation assertions).
For any type $B \in \mathrm{BType}$ *let* $\mathrm{BasicSep}(B)$ *be the least set containing the following* basic separation assertions*:*

$$\frac{}{\bot \in \mathrm{BasicSep}(B)} \qquad \frac{\xi_1, \xi_2 \in \mathrm{AddrD}(B)}{\zeta_{\xi_1} \otimes \zeta_{\xi_2} \in \mathrm{BasicSep}(B)}$$

$$\frac{\xi_R \in \mathrm{AddrR}(B), \xi_D \in \mathrm{AddrD}(B),\ \xi_R \sqsubseteq \xi_D}{\otimes\zeta_{\xi_D} \curlywedge \zeta_{\xi_R} \in \mathrm{BasicSep}(B),\ \otimes\zeta_{\xi_D}/\zeta_{\xi_R} \in \mathrm{BasicSep}(B)}$$

Let $\otimes\zeta$ *be a shortcut for* $\otimes\zeta\curlywedge\zeta$*.*

Separation assertions $C \in \mathrm{Sep}(B)$ *are sets of basic separation assertions.*

Sets $\mathrm{BasicSep}(E)$ *and* $\mathrm{Sep}(E)$ *are defined analogously.*

Definition 4 (Meaning of separation assertions).
We say that a valid representation $v \Vdash^\sigma_A a$ *satisfies an assertion* $C \in \mathrm{Sep}(B)$ *(written as* $v \Vdash^\sigma_{B;C} a$*) where* $\alpha(B) = A$*, if:*

- $\bot \notin C$
- *for every* $\zeta_{\xi_1} \otimes \zeta_{\xi_2} \in C$ *it holds* $P_A^{\xi_1}(v,\sigma) \cap P_A^{\xi_2}(v,\sigma) = \emptyset$
- *for every* $\otimes\zeta_\xi \curlywedge \zeta_{\xi_R} \in C$ *with* $A_{\xi_R} = \mu X.A'$ *and* $\xi = \xi_R\xi'$ *(implying* $A_\xi = A'_{\xi'}$*) it holds*

$$\forall \xi_1, \xi_2 \in \mathrm{UddrR}(A),\ {\xi_1}^{\mathrm{F},A} = {\xi_2}^{\mathrm{F},A} = \xi_R,\ \xi_1 \not\sqsubseteq \xi_2,\ \xi_2 \not\sqsubseteq \xi_1 \implies P_A^{\xi_1\xi'/\xi_R}(v,\sigma) \cap P_A^{\xi_2\xi'/\xi_R}(v,\sigma) = \emptyset$$

- *for every* $\otimes\zeta_\xi/\zeta_{\xi_R} \in C$ *with* $A_{\xi_R} = \mu X.A'$ *and* $\xi = \xi_R\xi'$ *it holds*

$$\forall \xi_1, \xi_2 \in \mathrm{UddrR}(A),\ {\xi_1}^{\mathrm{F},A} = {\xi_2}^{\mathrm{F},A} = \xi_R,\ \xi_1 \neq \xi_2 \implies P_A^{\xi_1\xi'/\xi_R}(v,\sigma) \cap P_A^{\xi_2\xi'/\xi_R}(v,\sigma) = \emptyset$$

The definition follows the informal description of separation assertions in the Introduction. The assertions $\otimes\zeta'/\zeta$ and $\otimes\zeta'\curlywedge\zeta$ correspond to internal separation *along* and *across* a recursive type (see Fig. 3).

For a list $\mathsf{L}^{[\zeta]}(A)$, the assertions $\otimes\zeta'\curlywedge\zeta$ (and $\otimes\zeta$ in particular) are void (because among any two unfolded addresses of cons-cells one is a prefix of the other) and thus trivially true in any context. We will therefore ignore them from now on. Assertions $\zeta\otimes\zeta'$ for the type $\mathsf{L}^{[\zeta]}(A)$ are not always tautological and will be kept explicitly in the $\mathrm{Sep}(B)$ sets.

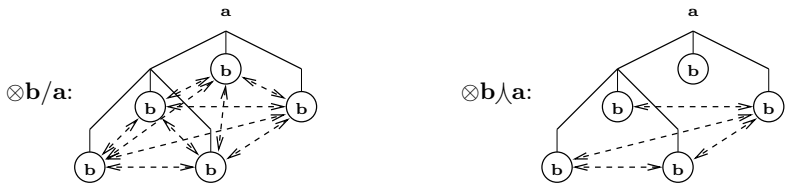

Fig. 3. Separation along and across a recursive type

Typing judgements. Before tackling the annotation of typing judgements, we need to extend some type-related notions to (annotated) typing contexts viewed as tuples of (annotated) types. We put $\mathrm{Addr}(\Gamma) = \big\{x\xi \;\big|\; \xi \in \mathrm{Addr}(\Gamma(x))\big\}$ and also define regions, name sets, separation assertions and a heap representation of value tuples and their portions and subportions related to typing contexts in a straightforward manner.

Definition 5 (Syntax of typing judgements).
The set of annotated typing judgements ZJudg *is defined as follows:*

$$\frac{\begin{array}{c}\Gamma = x_1 : B_1, \ldots, x_n : B_n \qquad C \in \mathrm{Sep}(\Gamma) \\ E \in \mathrm{EType},\ \mathrm{N_C}(E) \subseteq \mathrm{N_D}(\Gamma) \qquad D \subseteq \mathrm{N_D}(\Gamma) \qquad G \in \mathrm{BasicSep}(E) \to \mathrm{Sep}(\Gamma)\end{array}}{(\Gamma; C \vdash e : E; D; G) \in \mathrm{ZJudg}}$$

We will represent a concrete guarantee function G by a series of statements $s \Longleftarrow G(s)$ one for each $s \in \mathrm{Dom}(G) \setminus \{\bot\}$ with $G(s) \neq \emptyset$. We will always assume that a guarantee function G maps $\bot$ (i.e. false) to itself.

Definition 6 (Meaning of typing judgements).
A typing judgement $\Gamma; C \vdash e : E; D; G$ *is* sound *if whenever it holds* $S \Vdash^{\sigma}_{\Gamma;C} \eta$ *(separation preconditions) and* $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ *then it holds*

- *(correctness and internal separation guarantees)* $v \Vdash^{\sigma'}_{E;C_G} [\![e]\!]_{\eta,[\![P]\!]}$ *where* $C_G = \big\{s \in \mathrm{BasicSep}(E) \;\big|\; S \Vdash^{\sigma}_{\Gamma;G(s)} \eta\big\}$
- *(containment guarantees) for each* $\xi \in \mathrm{AddrD}(E)$ *with* $[\zeta_\xi \subseteq \gamma(\xi)]$ $P^{\xi}_{E}(v, \sigma') \subseteq \bigcup_{\zeta_{x\xi'} \in \gamma(\xi)} P^{x\xi'}_{\Gamma}(S, \sigma)$
- *(preservation guarantees) for each* $x\xi \in \mathrm{AddrD}(\Gamma)$ *it holds* $\sigma|_P = \sigma'|_P$ *where* $P = P^{x\xi}_{\Gamma}(S, \sigma) \setminus \bigcup_{\zeta_{x'\xi'} \in D} P^{x'\xi'}_{\Gamma}(S, \sigma)$

Consider the typing judgement for a predictably defined function which negates the first boolean (if existent) of every element in a list of lists of booleans:

$$\begin{array}{l} x : \mathsf{L}^{[\mathbf{a}]}(\mathsf{L}^{[\mathbf{b}]}(\mathsf{Bool})); \{\mathbf{a}{\otimes}\mathbf{b}, {\otimes}\mathbf{b}/\mathbf{a}\} \vdash \\ \quad \mathit{negheadlist}(x) : \mathsf{L}^{[\mathbf{c}\subseteq\{\mathbf{a}\}]}(\mathsf{L}^{[\mathbf{d}\subseteq\{\mathbf{b}\}]}(\mathsf{Bool})); \{\mathbf{a}, \mathbf{b}\};\ \ \mathbf{c}{\otimes}\mathbf{d} \Longleftarrow \{\mathbf{a}{\otimes}\mathbf{b}\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {\otimes}\mathbf{d}/\mathbf{c} \Longleftarrow \{{\otimes}\mathbf{b}/\mathbf{a}\} \end{array}$$

The precondition means that the elements of the list cannot overlap on the heap with each other. With our system, this is the only way to guarantee the correctness, i.e. that no boolean on the heap would be negated more than once.

One might argue that, in fact, only portion **b** gets modified and portion **a** does not change at all. This is true and follows from the fact that lists in portion **b** are modified in-place and moreover their resulting cons-cells occupy exactly the same positions as the original ones. Nevertheless, we have no means of expressing such a condition in our assertion language and when deriving the annotation in a purely compositional manner we have to assume that the lists could have been modified in a way which would change their reference location and thus modify portion **a** too.

5 Typing rules

For the complete set of annotated typing rules see [5]. In this paper we show only some representative cases which are listed in Fig. 4. For illustration, consider the following instantiation of the fairly complex typing rule [FOLD] for binary trees labelled with diamonds:

$$\begin{array}{r} x : {}^{\Diamond[\mathbf{a}]}\big(\Diamond^{[\mathbf{b}]}\big) + {}^{\Diamond[\mathbf{c}]}\big(\Diamond^{[\mathbf{d}]} \times \mathsf{T}_{\mathsf{b}}^{[\mathbf{e}]}(\Diamond^{[\mathbf{f}]}) \times \mathsf{T}_{\mathsf{b}}^{[\mathbf{g}]}(\Diamond^{[\mathbf{h}]})\big); \emptyset \vdash \\ \mathsf{fold}_{\mathsf{T}_{\mathsf{b}}(\Diamond)}\,(x) : \mathsf{T}_{\mathsf{b}}^{[\mathbf{i}\subseteq\{\mathbf{a},\mathbf{c},\mathbf{e},\mathbf{g}\}]}(\Diamond^{[\mathbf{j}\subseteq\{\mathbf{b},\mathbf{d},\mathbf{f},\mathbf{h}\}]}); \emptyset; \\ \mathbf{i}\otimes\mathbf{j} \Longleftarrow \{\mathbf{a},\mathbf{c},\mathbf{e},\mathbf{g}\}\otimes\{\mathbf{b},\mathbf{d},\mathbf{f},\mathbf{h}\} \;\; \otimes\mathbf{j}\not\downarrow\mathbf{i} \Longleftarrow \{\mathbf{f}\otimes\mathbf{h}\} \\ \otimes\mathbf{j}/\mathbf{i} \Longleftarrow \{\mathbf{f}\otimes\mathbf{h}\}\cup\{\mathbf{b},\mathbf{d}\}\otimes\{\mathbf{f},\mathbf{h}\} \quad \otimes\mathbf{i} \Longleftarrow \{\mathbf{e}\otimes\mathbf{g}\} \end{array}$$

In the [FUNC] rule we assume an annotated program P with $P(f) = \Gamma_f; C_f \vdash e_f : B_f; D_f; G_f$ where Γ_f contains exactly the variables $y_1, \ldots, y_{n_f}$ in this order.

Given a result type E, we define the minimal default guarantee $\mathrm{G_M}(E)$ for E as $[\zeta_1\otimes\zeta_2 \Longleftarrow Z_1\otimes_{\mathrm{D}} Z_2]_{\zeta_1\otimes\zeta_2\in\mathrm{BasicSep}(E)}$ where $[\zeta_1\subseteq Z_1]$ and $[\zeta_2\subseteq Z_2]$ are annotations in E and $\otimes_{\mathrm{D}}$ is one of the following two similar operations:

$$\begin{aligned} Z_1\otimes Z_2 &= \{\zeta_1\otimes\zeta_2 \,|\, \zeta_1 \in Z_1, \zeta_2 \in Z_2\} \\ Z_1\otimes_{\mathrm{D}} Z_2 &= \{\zeta_1\otimes\zeta_2 \,|\, \zeta_1 \in Z_1, \zeta_2 \in Z_2, \zeta_1 \neq \zeta_2\} \end{aligned}$$

The difference is significant only in the [UNFOLD] rule where every portion is being split into several pieces some pairs of which might be disjoint depending on the internal separation preconditions.

Several rules use a "copied" type $E \in \mathrm{E_Y}(B)$ for (part of) the result type which declares the one-to-one containment of E portions within corresponding B portions. With it go its default guarantees $\mathrm{G_Y}(E, B)$ which state that every basic separation assertion of E relies on the corresponding assertion for B.

In [FOLD], we construct the result type by a point-wise union of types which differ only in their containment sets. In several places also various rely-guarantee functions are merged via a point-wise union of images.

The notation $\Gamma_1 \wedge \Gamma_2$ stands for the merge of the context Γ_1, Γ_2. The use of this notation contains the implicit condition that every variable that appears in

[FUNC]
$$\frac{\Gamma = \Gamma_f[x_1/y_1,\ldots,x_{n_f}/y_{n_f}]}{\Gamma; C_f \vdash f(x_1,\ldots,x_{n_f}) : E_f; D_f; G_f}$$

[RENAME]
$$\frac{\Gamma; C \vdash e : E; D; G \quad \tau\,\mathrm{N}(\Gamma) \to \mathrm{Nm}}{\Gamma[\tau]; C[\tau] \vdash e : E[\tau]; D[\tau]; G[\tau]}$$

[LET]
$$\frac{\Gamma_1; C_1 \vdash e_1 : A\left[\frac{\delta,\gamma}{\rho}\right]; D_1; G_1 \quad \Gamma_2, x : A\left[\frac{\delta}{\rho}\right]; C_2 \vdash e_2 : E; D_2; G_2}{\Gamma_1 \wedge \Gamma_2; C_1 \cup T(C_2) \cup \big(D_1 \otimes \mathrm{N_D}(\Gamma_2)\big) \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : E[\gamma/\delta]; D_1 \cup D_2[\gamma/\delta]; T \circ G_2}$$

where $T = C \mapsto \big((C \setminus \mathrm{BSx})[\gamma/\delta]\big) \cup G_1\big(C \cap \mathrm{BSx}\big)$ and $\mathrm{BSx} = \mathrm{BasicSep}(A\left[\frac{\delta}{\rho}\right])$.

[PUT]
$$\frac{E \in \mathrm{E_Y}(^{\Diamond[\zeta]}B)}{x : B, d : \Diamond^{[\zeta]}; \{\zeta\} \otimes \mathrm{N_D}(B) \vdash x@d : E; \{\zeta\}; \mathrm{G_Y}(E, ^{\Diamond[\zeta]}B)}$$

[GET]
$$\frac{E \in \mathrm{E_Y}(B)}{x : {}^{\Diamond[\zeta]}B; \emptyset \vdash \mathsf{get}\,(x) : E; \emptyset; \mathrm{G_Y}(E, B)}$$

[LOC]
$$\frac{}{x : {}^{\Diamond[\zeta]}B; \emptyset \vdash \mathsf{loc}\,(x) : \Diamond^{[\zeta' \subseteq \{\zeta\}]}; \emptyset; \emptyset}$$

[PAIR]
$$\frac{E \in \mathrm{E_Y}(B_1 \times B_2)}{x_1 : B_1, x_2 : B_2; \emptyset \vdash (x_1, x_2) : E; \emptyset; \mathrm{G_Y}(E, B_1 \times B_2)}$$

[PAIR-ELIM]
$$\frac{\Gamma, x_1 : B_1, x_2 : B_2; C \vdash e : E; D; G}{\Gamma, x : B_1 \times B_2; C \vdash \mathsf{match}\ x\ \mathsf{with}\ \ (x_1, x_2) \Rightarrow e : E; D; G}$$

[FOLD]
$$\frac{B = (A[\mu X.A/X])\left[\frac{\delta}{\rho}\right] \quad B' = A\left[\frac{\delta|_{\mathrm{AddrD}(A)}}{\rho|_{\mathrm{AddrR}(A)}}\right] \quad E = E' \cup \bigcup_{\xi\overline{X} \in \mathrm{AddrX}_X(A)} E_{\xi\overline{\mu}_X} \quad E' \in \mathrm{E_Y}(B') \quad E_\xi \in \mathrm{E_Y}(B_\xi)}{x : B; \emptyset \vdash \mathsf{fold}_{\mu X.A}\,(x) : E; \emptyset; \mathrm{G_M}(E) \cup G \cup G' \cup G''}$$

where
$G = \mathrm{G_Y}(E', B') \cup \bigcup_{\xi\overline{X} \in \mathrm{AddrX}_X(A)} \mathrm{G_Y}(E_{\xi\overline{\mu}_X}, B_{\xi\overline{\mu}_X})$
$G' = \big[\otimes\zeta_\xi \curlywedge \zeta_{\overline{\mu}_X} \Longleftarrow \big\{\zeta_{x\xi'\xi} \otimes \zeta_{x\xi''\xi} \,\big|\, \xi'\overline{X}, \xi''\overline{X} \in \mathrm{AddrX}_X(A)\big\}\big]_{\xi \in \mathrm{AddrD}(A)}$ and
$G'' = \big[\otimes\zeta_\xi / \zeta_{\overline{\mu}_X} \Longleftarrow \mathrm{ditto} \cup \big\{\zeta_{x\xi'\xi} \otimes \zeta_{x\xi} \,\big|\, \xi'\overline{X} \in \mathrm{AddrX}_X(A)\big\}\big]_{\xi \in \mathrm{AddrD}(A)}$

[UNFOLD]
$$\frac{E \in \mathrm{E_Y}(B[\mu X^{(\zeta)}.B/X])}{x : \mu X^{(\zeta)}.B; \emptyset \vdash \mathsf{unfold}_{\mu X.\alpha(B)}\,(x) : E; \emptyset; \mathrm{G_Y}(E, B[\mu X^{(\zeta)}.B/X]) \cup G' \cup G''}$$

where
$G' = [\zeta_{x\xi'\xi} \otimes \zeta_{x\xi''\xi} \Longleftarrow \{\otimes\zeta_\xi \curlywedge \zeta_{\overline{\mu}_X}\}]_{\xi \in \mathrm{AddrD}(A), \xi'\overline{X}, \xi''\overline{X} \in \mathrm{AddrX}_X(A)}$ and
$G'' = \Big[\zeta_{x\xi'\overline{\mu}_\xi} \otimes \zeta_{x\xi} \Longleftarrow \{\otimes\zeta_\xi / \zeta_{\overline{\mu}_X}\}\Big]_{\xi \in \mathrm{AddrD}(A), \xi'\overline{X} \in \mathrm{AddrX}_X(A)}$

Fig. 4. Example annotated typing rules

both contexts has the same type and portion name annotations in both of them. To be able to apply rules that contain $\Gamma_1 \wedge \Gamma_2$ as intended, we might need to apply the [RENAME] rule first.

When $\delta\ \mathrm{AddrD}(A) \to \mathrm{Nm}$ and $\gamma\ \mathrm{AddrD}(A) \to \wp(\mathrm{Nm})$, then we derive a set-valued substitution $\gamma/\delta\ \mathrm{Nm} \rightharpoonup \wp(\mathrm{Nm})$ as $\zeta \mapsto \gamma(\delta^{-1}(\zeta))$. This is used in the [LET] rule.

Notice that, unlike in other extensions of LFPL, there is no side condition in [LET]. Instead, the illegal cases yield an inconsistent assertion set C, i.e. one containing $\bot$ or $\zeta{\otimes}\zeta$ for some portion name ζ. Similarly, when an additional separation guarantee s is unachievable, its rely-assertion $G(s)$ would contain an unsatisfiable basic separation assertion.

6 Correctness and inference

A typing rule is sound if whenever all the premises of an instance of the rule are sound, then so is the conclusion. Assuming that all the typing judgements $P(f)$ are sound, it is possible to prove that all the typing rules in our system as listed in [5] are correct. The proofs are conceptually simple but technical due to the complexity of the annotations. See the report for a sample proof of correctness for [LET].

Given a well-typed and well-annotated program P, for every term e over P we can perform the inference of its unannotated type deterministically as usual but we can also infer its annotation at the same time. Moreover, we can derive an annotation which does not reuse any name for two different portions or recursive type constructors. Such a derivable annotation of e is unique modulo renaming of portions and recursive type constructors.

Given an unannotated well-typed program P, we can infer its strongest derivable annotation by iteratively inferring weaker and weaker annotations starting from the strongest annotation: $C_f = \emptyset$, $E_f = A_f[{}^{\to\xi\mapsto\emptyset}_{\ -}]$, $D_f = \emptyset$ and $G_f = [s \Longleftarrow \emptyset]$ for all f in the program.

We have studied this process in a more abstract setting in [6] and showed that it will always terminate with the strongest derivable annotation for P provided that all the typing rules are monotone. In [5] we have outlined the proof that our rules are monotone.

Example The following is the strongest correct annotated judgement for the binary tree reversal algorithm mentioned in the Introduction:

$$
\begin{array}{l}
x : \mathsf{T}^{[\mathbf{a}]}_{\mathsf{b}}(\mathsf{L}^{[\mathbf{b}]}(\mathsf{Bool})); \{\mathbf{a}{\otimes}\mathbf{b}, {\otimes}\mathbf{a}\} \vdash \\
\quad \mathit{paths}_{\mathsf{L(Bool)}}(x) : \mathsf{L}^{[\mathbf{c}\subseteq\{\mathbf{a}\}]}(\mathsf{L}^{[\mathbf{d}\subseteq\{\mathbf{a}\}]}(\mathsf{L}^{[\mathbf{e}\subseteq\{\mathbf{b}\}]}(\mathsf{Bool}))); \{\mathbf{a}\}; \\
\mathbf{c}{\otimes}\mathbf{d} \Longleftarrow \{{\otimes}\mathbf{a}\} \qquad \mathbf{c}{\otimes}\mathbf{e} \Longleftarrow \{\mathbf{a}{\otimes}\mathbf{b}\} \qquad \mathbf{d}{\otimes}\mathbf{e} \Longleftarrow \{\mathbf{a}{\otimes}\mathbf{b}\} \\
{\otimes}\mathbf{d}/\mathbf{c} \Longleftarrow \{\bot, {\otimes}\mathbf{a}\}\ {\otimes}\mathbf{e}/\mathbf{c} \Longleftarrow \{\bot, {\otimes}\mathbf{b}/\mathbf{a}\}\ {\otimes}\mathbf{e}/\mathbf{d} \Longleftarrow \{{\otimes}\mathbf{b}/\mathbf{a}\}
\end{array}
$$

Due to the lack of parametric polymorphism, the example cannot involve types with an unbound variable. Therefore we have substituted a simple non-heap-free type $\mathsf{L(Bool)}$ in place of the usual type meta-variable A.

The annotation of *paths* tells us about the function that in order to work correctly, it needs the input tree to have non-overlapping skeleton ($\otimes\mathbf{a}$) disjoint from the labels ($\mathbf{a}\otimes\mathbf{b}$). The cons-cells of the result list ($\mathbf{c}$) as well as the cons-cells of the lists in it ($\mathbf{d}$) are constructed solely from the skeleton of the argument tree ($\mathbf{c}\subseteq\{\mathbf{a}\}$, $\mathbf{d}\subseteq\{\mathbf{a}\}$) which is also the only argument portion being destroyed. Thus the labels are preserved.

Notice that the elements of the result list cannot be guaranteed to be separated from each other ($\otimes\mathbf{d}/\mathbf{c}$) under any preconditions (indicated by $\bot$). Consequently, the same holds for the labels when viewed from the result lists' cons-cells ($\otimes\mathbf{e}/\mathbf{c}$). Labels viewed from the individual elements' cons-cells can be guaranteed to be separated from each other ($\otimes\mathbf{e}/\mathbf{d}$) provided that the labels of the argument tree are separated from each other ($\otimes\mathbf{b}/\mathbf{a}$).

7 Conclusion

We have enhanced the typing system of [3] following the natural idea that data and skeleton should be treated separately. This lead to a, perhaps surprisingly, complicated system of annotations. Despite using complex notation, the system provides efficient annotation inference.

A possible direct application of various extensions of LFPL including ours is to generate complex efficient imperative algorithms automatically from easy-to-verify functional programs. At the same time, a formal proof of correctness or permissible resource consumption could be generated fairly easily. For some problem areas the present language might not be sufficiently expressive. It should be possible, though, to adapt the annotation system in this paper to other language features including higher order functions and other kinds of resource-aware typings than that of LFPL.

For example, it could be used for functional languages without the $\Diamond$ types that have a fairly straightforward heap-based evaluation strategy. Recently Hofmann, Jost [7, 8] and Kirlı developed a statical analysis of memory consumption in a $\Diamond$-free version of LFPL with implicit memory allocation and deallocation. Combining the present system with the above is a subject of current research.

There is an implementation of the present typing system adapted to a language with ML-style datatypes as an add-on to an LFPL compiler by Robert Atkey. The compiler can be tested through a web interface [9]. It was used to generate the examples in this paper. To make the typing more practicable, a faily sophisticted system for generating helpful error messages is under development.

Related work. The theory of *Shapely types* by Jay [10] treats certain types as containers explicitly. Thus the idea to treat differently the skeleton (i.e. shape) and data layers of data types is common to both this and Jay's work. Nevertheless, the tools and the purpose of shape theory are very different from ours and do not seem to be applicable to the present system. Shape theory is formulated in categorical terms and studies mainly shape polymorphism and statical shape analysis on the semantical level. In contrast, we develop a statical analysis of non-interference in a particular operational semantics with in-place update.

The idea to indicate the level of boxing for representing values of recursive types using a "boxing" type constructor ($^{\Diamond}(\_)$ in our case) has been also used by Shao [11] using the notation `Boxed`$(\_)$. This idea appears often in the context of the strongly-typed intermediate language for compilation of functional languages *FLINT* [12].

The use of pre- and postconditions for certifying in-place update with sharing is not new. Recent work includes *Alias types* [13, 14] 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 [15] may serve a similar purpose as alias types but is also applicable to higher-level languages. The logic is in Hoare style where postconditions cannot refer to the original state and cannot therefore capture effects. If the logic is adapted so that it can refer to the original state, we believe that it is expressive enough to encode our assertions. To find a natural way of doing this is a promising ongoing research.

Our system bears some resemblance to the *Region inference* of Tofte and Talpin [16]. They also associate regions with types and infer certain effects made on these regions. One major difference is that they also infer where deallocation of regions should take place. This means that they derive special annotation in *terms* which influences their evaluation. An analogy to this in the context of LFPL might be the inference of diamond typed arguments ($@d$) within programs mentioned above. Another major difference is that the regions of Tofte and Talpin are mutually disjoint and cannot be explicitly overwritten.

The usage aspects of UAPL which are subsumed in the present system are similar to *use types* in linear logic [17] and also to *passivity* [18] within syntactic control of interference [19, 20]. These aspects can be also viewed as *effects* (in the sense of [21]) somewhat similar to `get` and `put` of [16]. Our system can be viewed as inferring a combination of effects on finely specified regions and separation assertions in the style of Reynolds.

Acknowledgements. This research has been supported by the EPSRC grant GR/N28436/01 and by the EC Fifth Framework Programme project Mobile Resource Guarantees. The author is grateful to David Aspinall, Robert Atkey and Martin Hofmann for discussion and comments on this work.

References

1. Hofmann, M.: A type system for bounded space and functional in-place update. Nordic Journal of Computing **7** (2000) 258–289
2. Hofmann, M.: The strength of non size-increasing computation. In: 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02). (2002) 260–269
3. Aspinall, D., Hofmann, M.: Another type system for in-place update. In Métayer, D.L., ed.: Programming Languages and Systems, Proceedings of 11th European

Symposium on Programming, Springer-Verlag (2002) 36–52 Lecture Notes in Computer Science 2305.
4. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (1998)
5. Konečný, M.: LFPL with types for deep sharing. Technical Report EDI-INF-RR-157, LFCS, Division of Informatics, University of Edinburgh (2002)
6. Konečný, M.: Typing with conditions and guarantees for functional in-place update. In: TYPES 2002 Workshop, Nijmegen, Proceedings. (2003) to appear.
7. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03). (2003) 185–197
8. Jost, S.: Static prediction of dynamic space usage of linear functional programs. Master's thesis, Technische Universität Darmstadt, Fachbereich Mathematik (2002)
9. Atkey, R., Konečný, M.: A prototype LFPL compiler (frontend DEEL). An interface available at: http://www.dcs.ed.ac.uk/home/resbnd/prototypes/by_Robert_Atkey/deel/ (2003)
10. Jay, C.B.: A semantics for shape. Science of Computer Programming **25** (1995) 251–283
11. Shao, Z.: Flexible representation analysis. In: Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97), Amsterdam, The Netherlands (1997) 85–98
12. League, C., Shao, Z.: Formal semantics of the FLINT intermediate language. Technical Report Yale-CS-TR-1171, Department of Computer Science, Yale University (1998)
13. Smith, F., Walker, D., Morrisett, G.: Alias types. In Smolka, G., ed.: 9th European Symposium on Programming (ESOP'2000), Springer-Verlag (2000) 366–381 Lecture Notes in Computer Science 1782.
14. Walker, D., Morrisett, G.: Alias types for recursive data structures. In: Types in Compilation 2000. (2001) 177–206 Lecture Notes in Computer Science 2071.
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02), Copenhagen, Denmark (2002) 55–74
16. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation **132** (1997) 109–176
17. Guzmán, J.C., Hudak, P.: Single-threaded polymorphic lambda calculus. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science. (1990) 333–343
18. O'Hearn, P.W., Power, A.J., Takeyama, M., Tennent, R.D.: Syntactic control of interference revisited. Theoretical Computer Science **228** (1999) 211–252
19. Reynolds, J.C.: Syntactic control of interference. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), ACM Press (1978) 39–46
20. Reynolds, J.C.: Syntactic control of interference, part 2. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: Automata, Languages and Programming, 16th International Colloquium, Springer-Verlag (1989) 704–722 Lecture Notes in Computer Science 372.
21. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1988) 47–57