

LFPL with Types for Deep Sharing

Michal Konečný

Abstract

First-order LFPL is a functional language for non-size increasing computation with an operational semantics that allows in-place update. The semantics is correct for all well-typed programs thanks to linear restrictions on the typing. Nevertheless, the linear typing is very strict and rejects many correct, natural in-place update algorithms. Aspinall and Hofmann added usage aspects to variables in LFPL terms in order to typecheck more programs while keeping correctness.

We further extend this language to typecheck even more programs, especially those that use data sharing on the heap. We do it by assigning usage aspects to certain type sub-terms instead of whole types thus referring to value portions instead of the whole values. The usage aspects express preconditions of separation between certain argument portions, the guarantees of preservation of certain argument portions and containment of every result portion in some set of argument portions and a rely-precondition for every separation guarantee within the result value. The new typing allows deterministic type-checking with automatic synthesis of the best usage aspects for recursively defined functions.

Contents

Contents	1
1 Introduction	2
2 Underlying language	5
3 Operational semantics	6
3.1 Data representation	6
3.2 Term evaluation	8
3.3 Heap regions	8
4 Annotations	12
4.1 Types	12
4.2 Typing judgements	14
5 Typing rules	18
6 Correctness	19
7 Inference	20
8 Examples	22
9 Conclusion	24
9.1 Related work	24
References	25

1 Introduction

First-order LFPL [Hofmann 2000] 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 [Hofmann 2002]. We focus on the first order version of LFPL in this report.

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

$$\begin{aligned} \text{append}_A(x, y) = & \text{match } x \text{ with nil} \Rightarrow y \\ & | \text{cons}(h, t)@d \Rightarrow \text{cons}(h, \text{append}_A(t, y))@d \end{aligned}$$

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 $\text{append}_A(x, x)$ with a non-empty list x . Such terms might fail to evaluate or evaluate with incorrect results. We will call a term *operationally correct* (OC) 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 operational correctness relative to some *extra condition* on the representation of the arguments on the heap. For example, we would like to declare $\text{append}_A(x, y)$ correct under certain condition on how lists x and y are represented. A necessary and sufficient condition for $\text{append}_A(x, y)$ 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, $\text{append}_A(x, y)$ 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 (Linear Functional Programming Language) uses linearity to achieve operational correctness of its terms. This means that in LFPL a variable cannot be used twice unless the occurrences are in the two components of a cartesian product or in the two branches of an if-then-else statement. Thanks to this restriction, LFPL maintains the “single pointer property”, i.e. to any heap location there exists at most one reference at any time. This has the consequence that different variables always refer to disjoint regions on the heap.

The language from [Aspinall & Hofmann 2002] (which we will call UAPL) relaxes the linearity of LFPL by adding an integer $i \in \{1, 2, 3\}$ (called usage aspect) to every variable in a typing context ($x :^i A$). The aspects indicate whether a variable's content on the heap might be destroyed (aspect 1) or has to be preserved during evaluation and also, in case it is preserved, whether the content of the variable may share with the result (aspect 2) or has to be

separated from it (aspect 3). For example, `append` will get the following typing:

$$x :^1 L(A), y :^2 L(A) \vdash \text{append}_A(x, y) : L(A).$$

In [2002] we described how the UAPL aspects can be translated to conditions and rely-guarantees and how this view of UAPL (or other similar typings of the LFPL underlying language) can help in its formalisation and its inference of annotation.

It is impossible to achieve a full characterisation of semantical correctness of the first-order LFPL evaluation by a simple typing system. Nevertheless, UAPL has improved over LFPL in allowing more of the correct algorithms to be recognised. For example, it type-checks some programs which safely access both components of two structures that potentially share some heap locations. This is impossible in LFPL.

Still, many correct programs will be rejected by UAPL. For example, if `appendA(x, y)` is used in some program where `x` and `y` could not be guaranteed to be fully disjoint (e.g. `appendA(cons(h, tx)@dx, cons(h, ty)@dy)`), then UAPL would not type-check it. UAPL needs this stronger precondition because it cannot distinguish between the cons-cell (shallow) and data (deep) levels of a list. This report 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) in any recursive data-type.

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 or binary tree type constructor $L(-)$, $T_b(-)$. We give a name ζ to each such a place (usually using a bold lower-case letter) and mark it in the type. For example, in a list constructor we mark it like this: $L^{[\zeta]}(-)$. 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).

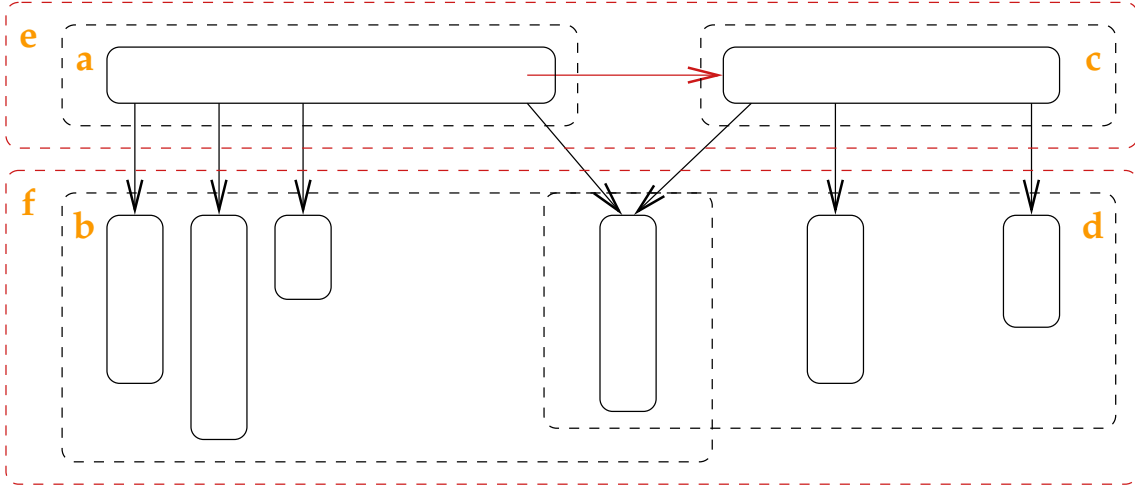
We formulate preconditions and rely-guarantees using the portion names given in the types within a typing judgement. The most obvious atomic condition is the separation (disjointness) of two portions ζ_1, ζ_2 on the heap which is written as $\zeta_1 \otimes \zeta_2$.

We need other atomic separation conditions 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 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 conditions $\otimes \zeta / \zeta_R$ and $\otimes \zeta \wedge \zeta_R$. 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 an sub-unfolding of the other (see Def. 4.4 for details).

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

Figure 1: Heap portions in the *append* program



The typing for $\text{append}_{L(\text{Bool})}$ in our new system is:

$$x : L^{[a]}(L^{[b]}(\text{Bool})), y : L^{[c]}(L^{[d]}(\text{Bool})); \{a \otimes b, a \otimes c, a \otimes d\} \vdash \\ \text{append}_{L(\text{Bool})}(x, y) : L^{[e \subseteq \{a, c\}]}(L^{[f \subseteq \{b, d\}]}(\text{Bool})); \{a\}; \otimes f/e \Leftarrow \{b \otimes d, \otimes b/a, \otimes d/c\}$$

Each argument has two portions indicated in their types. The argument portion names are **a**, **b**, **c**, **d** and are illustrated in Fig. 1. Just before \vdash there is a pre-condition 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 **e** and **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. $\{a\}$ in this case. The last guarantee is a list of implications giving a set of separation pre-conditions necessary for each basic separation guarantee about the result. The expression $\otimes f/e$ indicates internal separation between individual elements of the resulting list which depends on the same property for both argument lists and the separation of **b** from **d**.

The containment and destruction-limitation guarantees in the example above have an analogous function to the UAPL usage aspects.

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 algorithm converting a binary tree with labelled nodes (but not leaves) to the list containing for each leaf the list of labels as read along the path from the leaf to the root (Fig. 2).

The function $\text{pathsaux}_A(t, p)$ does the same as paths_A but appends p to all of the lists in its result list. Actually, in the course of the induction, all of the elements of the result list are constructed within calls to pathsaux_A as the second argument and only at the leaves this argument is added to the result lists. On return from the two inductive calls, the results are appended and returned.

In the operational point of view, the algorithm simply reverses all the pointers in the structure of the tree and turns the leaves into cons-cells of the resulting list. It is crucial here that

Figure 2: Binary Tree Reversal

$$\begin{aligned}
 \text{paths}_A(t) &= \text{pathsaux}_A(x, \text{nil}_A) \\
 \text{pathsaux}_A(t, p) &= \text{match } t \text{ with leaf}_b(-)@d \Rightarrow \text{let } n = \\
 &\quad \text{in cons}(p, \text{nil}_{L(A)})@d \\
 &\quad | \text{node}_b(a, l, r)@d \Rightarrow \text{let } ap = \text{cons}(a, p)@d \\
 &\quad \text{in let } s1 = \text{pathsaux}_A(l, ap) \\
 &\quad \text{in let } s2 = \text{pathsaux}_A(r, ap) \\
 &\quad \text{in append}_{L(A)}(s1, s2)
 \end{aligned}$$

leaves are not heap-free and their locations can be reused for cons-cells of the resulting list. The labels on leaves are ignored.

Notice that in the last append $s1$ and $s2$ are lists whose control structures are disjoint but whose data are lists that share among each other extensively.

Automatically inferred annotation for the above program can be found in Sect. 8.

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 recursive datatypes, sums and products, unit and LFPL-like memory-resource types:

$$A ::= \diamond \mid \diamond A \mid \text{Unit} \mid A_1 \times A_2 \mid A_1 + A_2 \mid X \mid \mu X.A$$

where X ranges over a set TVar of type variables. Let Type denote the set of all types. 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 representation¹.

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

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

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

¹Operationally viewed, the construction $\diamond A$ is analogous to Standard ML reference type $A \text{ ref}$ as well as to the pointer type construction $(A \ *)$ in C. Unlike in SML, our constructor is transparent to the denotational semantics.

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:

$$\begin{array}{lll} \text{[PUT]} & \text{[GET]} & \text{[LOC]} \\ x : A, d : \diamond \vdash x@d : \diamond A & x : \diamond A \vdash \text{get}(x) : A & x : \diamond A \vdash \text{loc}(x) : \diamond \end{array}$$

All types in a typing judgement have to be closed (without free type variables).

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

We consider the straightforward denotational semantics of types as sets:

- $\llbracket \text{Unit} \rrbracket = \{\text{unit}\}$
- $\llbracket A_1 \times A_2 \rrbracket = \llbracket A_1 \rrbracket \times \llbracket A_2 \rrbracket$
- $\llbracket \mu X.A \rrbracket = \llbracket A[X \mapsto \mu X.A] \rrbracket$
- $\llbracket \diamond \rrbracket = \{\diamond\}$
- $\llbracket A_L + A_R \rrbracket = \{L\} \times \llbracket A_L \rrbracket \cup \{R\} \times \llbracket A_R \rrbracket$
- $\llbracket \diamond A \rrbracket = \llbracket A \rrbracket$

Notice that $\diamond(_)$ is ignored and \diamond does not play any significant role here.

Denotation of programs $\llbracket P \rrbracket$ and terms $\llbracket e \rrbracket_{\eta, \llbracket P \rrbracket}$ with valuation η of free variables in e is defined as usual apart from the pointer-related constructs which are virtually ignored:

- $\llbracket \text{loc}(x) \rrbracket_{\eta, \llbracket P \rrbracket} = \diamond$
- $\llbracket x@d \rrbracket_{\eta, \llbracket P \rrbracket} = \llbracket \text{get}(x) \rrbracket_{\eta, \llbracket P \rrbracket} = \llbracket x \rrbracket_{\eta, \llbracket P \rrbracket} = \eta(x)$

In examples, we will use a straightforward *shortcut notation* for expressions with booleans, lists and a version of labelled binary trees as shown in Fig. 3. Finitely-branching labelled trees are also included for illustration.

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.

3.1 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 (provided it is laid out without sharing). 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. 3 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

Figure 3: Shortcut notation for booleans, lists and trees.

$$\begin{array}{ll}
\text{Bool} = \text{Unit} + \text{Unit} & \text{tt} = \text{inL}(\text{unit}), \text{ff} = \text{inR}(\text{unit}) \\
\text{L}(A) = \mu X. \text{Unit} + \diamond(A \times X) & \text{nil}_A = \text{fold}_{\text{L}(A)}(\text{inL}(\text{unit})) \\
& \text{cons}(h, t)@d = \text{fold}_{\text{L}(A)}(\text{inR}((h, t)@d)) \\
& \text{match } x \text{ with nil} \Rightarrow e_1 | \text{cons}(h, t)@d \Rightarrow e_2 = \dots \\
\text{T}_b(A) = \mu X. \diamond A + \diamond(A \times (X \times X)) & \text{leaf}_b(a)@d = \text{fold}_{\text{T}_b(A)}(\text{inL}(a@d)) \\
& \text{node}_b(a, l, r)@d = \text{fold}_{\text{T}_b(A)}(\text{inR}((a, (l, r))@d)) \\
& \text{match } x \text{ with leaf}_b(a)@d \Rightarrow e_1 = \dots \\
& \quad | \text{node}_b(a, l, r)@d \Rightarrow e_2 \\
\text{T}(A) = \mu X. A \times \text{L}_N(X) & \text{tree}(a, l) = \text{fold}_{\text{T}(A)}((a, l)) \\
& \text{match } t \text{ with tree}(a, l) \Rightarrow e = \dots \\
\text{L}_N(A) = \mu X. \diamond \text{Unit} + \diamond(A \times X) & \text{nil}_A@d = \text{fold}_{\text{L}_N(A)}(\text{inL}(\text{unit}@d)) \\
& \dots
\end{array}$$

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. Nevertheless, we will not make use of this possibility in this report as we will consider the two portions as one.

We will now make the heap-based representation of values precise. 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:

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

An *environment* is a partial mapping $S: \text{Var} \rightarrow \text{Val}$ from variables to operational values.

A *heap* $\sigma: \text{Loc} \rightarrow \text{Val}$ 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 $v \Vdash_A^\sigma a$ defined as follows:

- $\text{unit} \Vdash_{\text{Unit}}^\sigma \text{unit}$
- $\ell \Vdash_{\diamond}^\sigma \diamond,$
- $(v_1, v_2) \Vdash_{A_1 \times A_2}^\sigma (a_1, a_2)$ if $v_k \Vdash_{A_k}^\sigma a_k$ for $k = 1, 2$
- $\text{inl}(v) \Vdash_{A_L + A_R}^\sigma (L, a)$ if $v \Vdash_{A_L}^\sigma a$
- $\text{inr}(v) \Vdash_{A_L + A_R}^\sigma (R, a)$ if $v \Vdash_{A_R}^\sigma a$
- $v \Vdash_{\mu X. A}^\sigma a$ if $v \Vdash_{A[\mu X. A/X]}^\sigma a,$
- $\ell \Vdash_{\diamond A}^\sigma a$ if $\sigma(\ell) \Vdash_A^\sigma a.$

Whenever $v \Vdash_A^\sigma a$ cannot be derived by a finite derivation from the rules above, the quadruple is not in the relation.

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 \mathcal{P} are bounded. A recursive type $\mu X.A$ is *bounded* if all occurrences of X in A are within some $\diamond(-)$.

3.2 Term evaluation

Using the heap representation of values, define the operational semantics by an evaluation relation $\mathcal{S}, \sigma \vdash e \rightsquigarrow v, \sigma'$ given in Fig. 4.

This evaluation allows *in-place update* and is *non-size-increasing* in the sense that there is no means of allocating new space on the heap. All heap operations either read the heap or overwrite some of the previously referenced locations.

3.3 Heap regions

The complete heap region $\mathcal{R}_A(v, \sigma)$ taken by an operational value v with $v \Vdash_A^\sigma a$ is defined as follows:

- $\mathcal{R}_{\text{Unit}}(\text{unit}, \sigma) = \emptyset$
- $\mathcal{R}_\diamond(\ell, \sigma) = \{\ell\}$
- $\mathcal{R}_{A_1 \times A_2}((v_1, v_2), \sigma) = \mathcal{R}_{A_1}(v_1, \sigma) \cup \mathcal{R}_{A_2}(v_2, \sigma)$
- $\mathcal{R}_{A_L + A_R}(\text{inl}(v), \sigma) = \mathcal{R}_{A_L}(v, \sigma)$
- $\mathcal{R}_{A_L + A_R}(\text{inr}(v), \sigma) = \mathcal{R}_{A_R}(v, \sigma)$
- $\mathcal{R}_{\mu X.A}(v, \sigma) = \mathcal{R}_{A[\mu X.A/X]}(v, \sigma)$
- $\mathcal{R}_\diamond A(\ell, \sigma) = \{\ell\} \cup \mathcal{R}_A(\sigma(\ell), \sigma)$

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 “control structure” 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 list value of type

$$L(L(\text{Bool})) = \mu X.\text{Unit} + \diamond((\mu Y.\text{Unit} + \diamond(\text{Bool} \times Y)) \times X)$$

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 $\diamond(-)$.

In order to be able to make this distinction, we need to define the *portions* of a region $\mathcal{R}_A(v, \sigma)$ corresponding to each \diamond in A .

In order to be able to talk about occurrences of \diamond in a type A , as well as occurrences of recursive types and type variables within them, let us define formal addresses of these kinds of subterms of a type.

Figure 4: Definition of Evaluation Relation

<p>[UNIT]</p> $\frac{}{S, \sigma \vdash \text{unit} \rightsquigarrow \text{unit}, \sigma}$	<p>[VAR]</p> $\frac{}{S, \sigma \vdash x \rightsquigarrow S(x), \sigma}$
<p>[FUNC]</p> $\frac{S(x_i) = v_i \quad [x_i \mapsto v_i], \sigma \vdash e_f \rightsquigarrow v, \sigma'}{S, \sigma \vdash f(x_1, \dots, x_n) \rightsquigarrow v, \sigma'}$	
<p>[LET]</p> $\frac{S, \sigma \vdash e_1 \rightsquigarrow v, \sigma' \quad S[x \mapsto v], \sigma' \vdash e_2 \rightsquigarrow v', \sigma''}{S, \sigma \vdash \text{let } x = e_1 \text{ in } e_2 \rightsquigarrow v', \sigma''}$	
<p>[PAIR]</p> $\frac{}{S, \sigma \vdash (x_1, x_2) \rightsquigarrow (S(x_1), S(x_2)), \sigma}$	
<p>[PAIR-ELIM]</p> $\frac{S(x) = (v_1, v_2) \quad S[x_1 \mapsto v_1][x_2 \mapsto v_2], \sigma \vdash e \rightsquigarrow v, \sigma'}{S, \sigma \vdash \text{match } x \text{ with } (x_1, x_2) \Rightarrow e \rightsquigarrow v, \sigma'}$	
<p>[INL]</p> $\frac{}{S, \sigma \vdash \text{inL}(x) \rightsquigarrow \text{inl}(S(x)), \sigma}$	<p>[INR]</p> $\frac{}{S, \sigma \vdash \text{inR}(x) \rightsquigarrow \text{inr}(S(x)), \sigma}$
<p>[SUM-ELIM-LEFT]</p> $\frac{S(x) = \text{inl}(v_L) \quad S[x_L \mapsto v_L], \sigma \vdash e_L \rightsquigarrow v, \sigma'}{S, \sigma \vdash \text{case } x \text{ of inL}(x_L) \Rightarrow e_L \text{inR}(x_R) \Rightarrow e_R \rightsquigarrow v, \sigma'}$	
<p>[SUM-ELIM-RIGHT]</p> $\frac{S(x) = \text{inr}(v_R) \quad S[x_R \mapsto v_R], \sigma \vdash e_R \rightsquigarrow v, \sigma'}{S, \sigma \vdash \text{case } x \text{ of inL}(x_L) \Rightarrow e_L \text{inR}(x_R) \Rightarrow e_R \rightsquigarrow v, \sigma'}$	
<p>[FOLD]</p> $\frac{}{S, \sigma \vdash \text{fold}_{\mu X.A}(x) \rightsquigarrow S(x), \sigma}$	<p>[UNFOLD]</p> $\frac{}{S, \sigma \vdash \text{unfold}_{\mu X.A}(x) \rightsquigarrow S(x), \sigma}$
<p>[PUT]</p> $\frac{}{S, \sigma \vdash x@d \rightsquigarrow S(d), \sigma[S(d) \mapsto S(x)]}$	
<p>[GET]</p> $\frac{}{S, \sigma \vdash \text{get}(x) \rightsquigarrow \sigma(S(x)), \sigma}$	<p>[LOC]</p> $\frac{}{S, \sigma \vdash \text{loc}(x) \rightsquigarrow S(x), \sigma}$

Definition 3.1 (Type subterm addresses).

Let us define the set of **type subterm addresses** $\text{AddrXRD}(\mathbf{A})$ of a type \mathbf{A} as the set of those sequences ξ , for which \mathbf{A}_ξ (see below) is defined. The sequences are made of $\diamond, \mathbf{L}, \mathbf{R}, 1, 2$ and $\bar{\mu}_X, \bar{X}$ for all $X \in \text{TVar}$. These addresses correspond to certain subterms of \mathbf{A} as follows:

$$\begin{aligned} \diamond_{\diamond} &= \diamond & (\diamond \mathbf{A})_{\diamond} &= \diamond \mathbf{A} & (\diamond \mathbf{A})_{\diamond \xi} &= \mathbf{A}_\xi \\ (\mathbf{A}_L + \mathbf{A}_R)_{\mathbf{L}\xi} &= (\mathbf{A}_L)_\xi & (\mathbf{A}_L + \mathbf{A}_R)_{\mathbf{R}\xi} &= (\mathbf{A}_R)_\xi \\ (\mathbf{A}_1 \times \mathbf{A}_2)_{1\xi} &= (\mathbf{A}_1)_\xi & (\mathbf{A}_1 \times \mathbf{A}_2)_{2\xi} &= (\mathbf{A}_2)_\xi \\ (\mu X. \mathbf{A})_{\bar{\mu}_X} &= \mu X. \mathbf{A} & (\mu X. \mathbf{A})_{\bar{\mu}_X \xi} &= \mathbf{A}_\xi \\ X_{\bar{X}} &= X \end{aligned}$$

Let $\text{AddrD}(\mathbf{A})$ be the subset of $\text{AddrXRD}(\mathbf{A})$ consisting of the addresses which end with \diamond . A type \mathbf{A} is called **heap-free** iff $\text{AddrD}(\mathbf{A}) = \emptyset$.

Similarly, for a given type variable X , let $\text{AddrX}_X(\mathbf{A})$ be the subset of $\text{AddrXRD}(\mathbf{A}')$ consisting of the addresses ending with \bar{X} where \mathbf{A}' is \mathbf{A} in which all bound type variables have been renamed not to coincide with X . (Thus $\text{AddrX}_X(\mathbf{A})$ is the set of addresses of the free occurrences of X in \mathbf{A} .)

Finally, let $\text{AddrR}(\mathbf{A})$ be the subset of $\text{AddrXRD}(\mathbf{A})$ of addresses ending with $\bar{\mu}_X$ for some X .

For example, using the type $\mathbf{A} = \diamond \times (\mu Y. \text{Unit} + \diamond Y)$ we get $\mathbf{A}_{1\diamond} = \diamond$, $\mathbf{A}_{2\bar{\mu}_Y} = \mu Y. \text{Unit} + \diamond Y$ and $\mathbf{A}_{2\bar{\mu}_Y \mathbf{R}\bar{Y}} = Y$.

When it holds $v \Vdash_{\mathbf{A}}^\sigma a$, we may unfold the recursive types within \mathbf{A} arbitrarily and still yielding a valid type \mathbf{A}' for which it holds $v \Vdash_{\mathbf{A}'}^\sigma a$. Moreover, when imagining the fully unfolded (infinite) version of \mathbf{A} , we can capture all the possible shapes that the values might take on the heap. Thus we will formalise addresses into this imaginary infinite type to help us with the definition of heap portions later.

Definition 3.2 (Addresses within fully unfolded types).

For a closed type \mathbf{A} , let $\text{UddrRD}(\mathbf{A})$ denote the set of **unfolded type subterm addresses** which is defined analogously to $\text{AddrXRD}(\mathbf{A})$ as the set of addresses ξ , for which \mathbf{A}_ξ^ξ is defined. The unfolded type subterm \mathbf{A}_ξ^ν is defined for some types \mathbf{A} , address ξ and type variable valuation $\nu: \text{TVar} \rightarrow \text{Type}$ as follows:

$$\begin{aligned} (\mu X. \mathbf{A})_{\bar{\mu}_X}^\nu &= \mu X. \mathbf{A} & (\mu X. \mathbf{A})_{\bar{\mu}_X \xi}^\nu &= \mathbf{A}_\xi^{\nu[X \mapsto \mathbf{A}]} \\ X_{\bar{\mu}_X}^\nu &= \mu X. \nu(X) & X_{\bar{\mu}_X \xi}^\nu &= \nu(X)_\xi^\nu \end{aligned}$$

plus equations for $\diamond, \diamond -, - + -, - \times -$ which are analogous to those for \mathbf{A}_ξ .

Moreover, we associate to every unfolded address $\xi \in \text{UddrRD}(\mathbf{A})$ its corresponding non-unfolded address $\xi^F \in \text{AddrXRD}(\mathbf{A})$ as follows:

- If $\xi \in \text{AddrXRD}(\mathbf{A})$ then $\xi^F = \xi$,
- otherwise, let $\xi_p \bar{\mu}_X$ be the only prefix of ξ (denoting the rest by ξ_r : $\xi = \xi_p \bar{\mu}_X \xi_r$) such that $\xi_p \bar{X} \in \text{AddrXRD}(\mathbf{A})$ and let ξ_μ be the longest prefix of ξ_p ending with $\bar{\mu}_X$. Then define $\xi^F = \xi_\mu (\xi_r)^F$ whenever ξ_r is not empty and $\xi^F = \xi_\mu$ otherwise.

Consider a representation of a value of the list type $\mu X. \text{Unit} + \diamond(A \times X)$. All the unfolded addresses $\bar{\mu}_X R (\diamond 2 \bar{\mu}_X)^n \bar{\diamond}$ 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 cannot be associated with any heap location.

In the following definition, we will formalise this observation in the general case: we will associate one heap location to each one among a certain subset of unfolded addresses of a diamond within a type, given a particular representation of a value of that type.

Definition 3.3 (Locations for unfolded D-addresses).

Assume $v \Vdash_{\bar{A}}^{\sigma} a$ and pick a type address $\xi \in \text{UddrD}(A)$. Let $v_{\xi}(A, v, \sigma) \in \text{Dom}(\sigma)$ denote the heap location $v_{\xi}^{\diamond}(A, v, \sigma)$ (see below) if it is defined and let $v_{\xi}(A, v, \sigma)$ be undefined otherwise.

A heap location $v_{\xi}^{\vee}(A, v, \sigma)$ is defined in some cases, as follows:

$$\begin{aligned} v_{\bar{\diamond}}^{\vee}(\bar{\diamond}, \ell, \sigma) &= v_{\bar{\diamond}}^{\vee}(\bar{\diamond}A, \ell, \sigma) = \ell & v_{\bar{\diamond}\xi}^{\vee}(\bar{\diamond}A, \ell, \sigma) &= v_{\xi}^{\vee}(A, \sigma(\ell), \sigma) \\ v_{L\xi}^{\vee}(A_L + A_R, \text{inl}(v), \sigma) &= v_{\xi}^{\vee}(A_L, v, \sigma) & v_{R\xi}^{\vee}(A_L + A_R, \text{inr}(v), \sigma) &= v_{\xi}^{\vee}(A_R, v, \sigma) \\ v_{1\xi}^{\vee}(A_1 \times A_2, (v_1, v_2), \sigma) &= v_{\xi}^{\vee}(A_1, v_1, \sigma) & v_{2\xi}^{\vee}(A_1 \times A_2, (v_1, v_2), \sigma) &= v_{\xi}^{\vee}(A_2, v_2, \sigma) \\ v_{\bar{\mu}_X\xi}^{\vee}(\mu X.A, v, \sigma) &= v_{\xi}^{\vee[X \rightarrow A]}(A, v, \sigma) & v_{\bar{\mu}_X\xi}^{\vee}(X, v, \sigma) &= v_{\xi}^{\vee}(v(X), v, \sigma) \end{aligned}$$

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 3.4 (Heap portions and subportions for diamonds).

Whenever $v \Vdash_{\bar{A}}^{\sigma} a$ and $\xi \in \text{AddrD}(A)$, we define the **portion** of the region $R_A(v, \sigma)$ corresponding to the unfolded type address ξ as the set

$$P_A^{\xi}(v, \sigma) = \left\{ v_{\xi'}(A, v, \sigma) \mid \xi' \in \text{UddrD}(A), (\xi')^F = \xi, v_{\xi'}(A, v, \sigma) \text{ defined} \right\}$$

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

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

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

Notice that different heap portions of a value may overlap with each other in general. For example, a list $[(R, [(L, ff)]), (L, ff)]$ of type

$$L(\text{Bool} + L(\text{Bool} + \text{Bool})) = \mu X. \text{Unit} + \diamond((\text{Bool} + (\mu Y. \text{Unit} + \diamond(\text{Bool} + \text{Bool} \times Y))) \times X)$$

represented by ℓ_1 on the heap:

$$\begin{aligned} \ell_1 &\mapsto \text{cons}(\text{inr}(\ell_2), \ell_2), \\ \ell_2 &\mapsto \text{cons}(\text{inl}(ff), \text{nil}) \end{aligned}$$

has two portions corresponding to the type addresses $\bar{\mu}_X R \bar{\diamond}$ and $\bar{\mu}_X R \bar{\diamond} 1 R \bar{\mu}_Y R \bar{\diamond}$. The first portion equals to $\{\ell_1, \ell_2\}$ and the second one to $\{\ell_2\}$.

4 Annotations

We will add several kinds of annotations to every typing judgement $\Gamma \vdash e : A$.

First, we will assign a name ζ from a set \mathbf{Nm} to each portion (diamond) address and each recursive type address within the parameters in the context Γ . (In the examples, we will use bold letters $\mathbf{a}, \mathbf{b}, \dots$ as names.) There is a bijection between these names and classes of a certain equivalence relation on pairs consisting of a variable and an address within its type. In a machine implementation of the present system, it might be more convenient to work with type addresses and argument names modulo an equivalence relation instead of portion names. The main reason for introducing portion names is to make typing rules simpler and typing judgements shorter and easier to read.

Further, the typing judgement will contain a separation precondition predicate represented by an element of a set $\mathbf{Sep}(\Gamma)$ which will be defined later. Each element \mathbf{C} of $\mathbf{Sep}(\Gamma)$ induces a binary relation on the argument portion names indicating which portions have to be disjoint on the heap. The rest of \mathbf{C} may indicate internal separation within some portions modulo the unfolding of some recursive types. For example, the elements of some list might need to be separated from each other.

The result type will also be annotated at each portion and recursive type address with a portion name ζ from \mathbf{Nm} . Moreover, portion addresses will be annotated with a set \mathbf{Z} of argument portion names. This set indicates the guarantee that ζ will be fully contained within the space of the argument portions whose names are in \mathbf{Z} .

Another guarantee will be expressed by a distinct subset \mathbf{D} of the argument portion names. Portions in \mathbf{D} could be modified during the evaluation of the term e . All other portions are guaranteed to be preserved wherever they are not overlapping with the portions in \mathbf{D} .

Finally, there will be a separation condition predicate from $\mathbf{Sep}(\Gamma)$ for each basic separation predicate of the result type showing a sufficient condition for its validity.

4.1 Types

We first need to formalise a type labelled at its diamond 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.

Definition 4.1 (Generic annotated type).

Given a type A and functions $f_D: \mathbf{AddrD}(A) \rightarrow \mathbf{T}_D$, $f_R: \mathbf{AddrR}(A) \rightarrow \mathbf{T}_R$ for some set \mathbf{T} , the expression $A \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right]$ is an **annotated type**. Another syntax for the same annotated type is defined inductively as follows:

$$\begin{aligned}
 \diamond \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= \diamond^{[f_D(\overline{\diamond})]} & (\diamond A) \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= \diamond^{[f_D(\overline{\diamond})]} \left(A \left[\begin{smallmatrix} f_D \circ [\xi \mapsto \overline{\diamond} \xi] \\ f_R \circ [\xi \mapsto \overline{\diamond} \xi] \end{smallmatrix} \right] \right) & \mathbf{Unit} \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= \mathbf{Unit} \\
 (A_L + A_R) \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= A_L \left[\begin{smallmatrix} f_D \circ [\xi \mapsto L\xi] \\ f_R \circ [\xi \mapsto L\xi] \end{smallmatrix} \right] + A_R \left[\begin{smallmatrix} f_D \circ [\xi \mapsto R\xi] \\ f_R \circ [\xi \mapsto R\xi] \end{smallmatrix} \right] \\
 (A_1 \times A_2) \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= A_1 \left[\begin{smallmatrix} f_D \circ [\xi \mapsto 1\xi] \\ f_R \circ [\xi \mapsto 1\xi] \end{smallmatrix} \right] \times A_2 \left[\begin{smallmatrix} f_D \circ [\xi \mapsto 2\xi] \\ f_R \circ [\xi \mapsto 2\xi] \end{smallmatrix} \right] \\
 \mu X. A \left[\begin{smallmatrix} f_D \\ f_R \end{smallmatrix} \right] &= \mu X^{(f_R(\overline{\mu}_X))}. A \left[\begin{smallmatrix} f_D \circ [\xi \mapsto \overline{\mu}_X \xi] \\ f_R \circ [\xi \mapsto \overline{\mu}_X \xi] \end{smallmatrix} \right]
 \end{aligned}$$

Let $\alpha(_)$ be the obvious polymorphic forgetful mapping from annotated to plain types.

As examples of the syntax introduced above, 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:

$$\begin{aligned} L^{[\zeta]}(A) &= \mu X^{(\zeta)}. \text{Unit} + \diamond^{[\zeta]}(A \times X) \\ L_n^{[\zeta]}(A) &= \mu X^{(\zeta)}. \diamond^{[\zeta]} \text{Unit} + \diamond^{[\zeta]}(A \times X) \\ T_b^{[\zeta]}(A) &= \mu X^{(\zeta)}. \diamond^{[\zeta]} A + \diamond^{[\zeta]}(A \times X \times X) \\ T^{(\zeta_R)[\zeta]}(A) &= \mu X^{(\zeta_R)}. A \times L_n^{[\zeta]}(X) \end{aligned}$$

where A stands for an annotated type in this case.

Definition 4.2 (Annotated types B and E).

Let $BType$ be the set of all annotated types $A[\frac{\delta}{\rho}]$ where $\delta: \text{AddrD}(A) \rightarrow \text{Nm}$, $\rho: \text{AddrR}(A) \rightarrow \text{Nm}$.

Let $EType$ be the set of all annotated types $A[\frac{\delta, \gamma}{\rho}]$ with $\delta, \gamma: \text{AddrD}(A) \rightarrow \text{Nm} \times \wp(\text{Nm})$ where ρ and δ fulfil the same conditions as before.

In the other syntax for $A[\frac{\delta, \gamma}{\rho}]$, a pair $(\zeta, \{\zeta_1, \zeta_2, \dots\})$ from the image of δ, γ will be usually written as $[\zeta \sqsubseteq \{\zeta_1, \zeta_2, \dots\}]$.

For $B = A[\frac{\delta}{\rho}] \in BType$ and $E = A[\frac{\delta, \gamma}{\rho}] \in EType$ let

$$\begin{aligned} N_D(B) &= N_D(E) = \text{Ran}(\delta) \\ N_R(B) &= N_R(E) = \text{Ran}(\rho) \\ N_C(E) &= \bigcup_{\xi \in \text{AddrR}(A)} \gamma(\xi) \end{aligned}$$

and also $N(B) = N_D(B) \cup N_R(B)$ and $N(E) = N_D(E) \cup N_R(E)$.

Thus $N_D(B)$ and $N_R(B)$ are the sets of the names of all the diamonds and recursive type constructors, respectively, within B .

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.

All the notions defined for plain types extend by analogy to annotated types. In particular, $\text{AddrXRD}(B)$ and B_ξ are well defined for a $B \in BType$ as well as for an $E \in EType$.

For any $\xi \in \text{AddrD}(B) \cup \text{AddrR}(B)$, 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 4.3 (Syntax of separation conditions).

For any type $B \in BType$ let $\text{BasicSep}(B)$ be the least set containing the following basic separation conditions:

$$\begin{array}{c} \frac{}{\perp \in \text{BasicSep}(B)} \quad \frac{\xi_1, \xi_2 \in \text{AddrD}(B)}{\zeta_{\xi_1} \otimes \zeta_{\xi_2} \in \text{BasicSep}(B)} \\ \frac{\xi_R \in \text{AddrR}(B), \xi_D \in \text{AddrD}(B), \xi_R \sqsubseteq \xi_D}{\otimes \zeta_{\xi_D} \wedge \zeta_{\xi_R} \in \text{BasicSep}(B), \otimes \zeta_{\xi_D} / \zeta_{\xi_R} \in \text{BasicSep}(B)} \end{array}$$

Let $\otimes \zeta$ be a shortcut for $\otimes \zeta \wedge \zeta$.

Separation conditions are sets of basic separation conditions: $\text{Sep}(\mathbf{B}) = \wp(\text{BasicSep}(\mathbf{B}))$.

For $\mathbf{E} \in \mathbf{EType}$, separation conditions $\text{BasicSep}(\mathbf{E})$ and $\text{Sep}(\mathbf{E})$ are defined analogously.

Definition 4.4 (Meaning of separation conditions).

We say that a valid representation $\nu \Vdash_{\mathbf{A}}^{\sigma} \mathbf{a}$ satisfies a condition $\mathbf{C} \in \text{Sep}(\mathbf{B})$ (written as $\nu \Vdash_{\mathbf{B}; \mathbf{C}}^{\sigma} \mathbf{a}$) where $\alpha(\mathbf{B}) = \mathbf{A}$, if:

- $\perp \notin \mathbf{C}$
- for every $\zeta_{\xi_1} \otimes \zeta_{\xi_2} \in \mathbf{C}$ it holds $\mathbf{P}_{\mathbf{A}}^{\xi_1}(\nu, \sigma) \cap \mathbf{P}_{\mathbf{A}}^{\xi_2}(\nu, \sigma) = \emptyset$
- for every $\otimes \zeta_{\xi} \wedge \zeta_{\xi_R} \in \mathbf{C}$ with $\mathbf{A}_{\xi_R} = \mu X. \mathbf{A}'$ and $\xi = \xi_R \xi'$ (implying $\mathbf{A}_{\xi} = \mathbf{A}'_{\xi'}$) it holds

$$\begin{aligned} \forall \xi_1, \xi_2 \in \text{UddrR}(\mathbf{A}), \xi_1^F = \xi_2^F = \xi_R, \xi_1 \not\sqsubseteq \xi_2, \xi_2 \not\sqsubseteq \xi_1 \\ \implies \mathbf{P}_{\mathbf{A}}^{\xi_1 \xi' / \xi_R}(\nu, \sigma) \cap \mathbf{P}_{\mathbf{A}}^{\xi_2 \xi' / \xi_R}(\nu, \sigma) = \emptyset \end{aligned}$$

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

$$\begin{aligned} \forall \xi_1, \xi_2 \in \text{UddrR}(\mathbf{A}), \xi_1^F = \xi_2^F = \xi_R, \xi_1 \neq \xi_2 \\ \implies \mathbf{P}_{\mathbf{A}}^{\xi_1 \xi' / \xi_R}(\nu, \sigma) \cap \mathbf{P}_{\mathbf{A}}^{\xi_2 \xi' / \xi_R}(\nu, \sigma) = \emptyset \end{aligned}$$

For example, for a type $\mathbf{T}_b^{[a]}(\mathbf{L}^{[b]}(\mathbf{L}^{[c]}(\mathbf{Bool})))$, the condition $\otimes \mathbf{a}$ means that the nodes and leaves of the tree are laid out in the heap without any sharing and $\otimes \mathbf{b}/\mathbf{a}$ means that the lists labelling the tree do not share with each other. The condition $\otimes \mathbf{c}/\mathbf{b}$ means that each of the labelling list has elements separated from each other and is independent from $\otimes \mathbf{b}/\mathbf{a}$ as well as $\otimes \mathbf{c}/\mathbf{a}$ which stands for the separation of the contents between different labels of the tree. Notice that $\otimes \mathbf{c}/\mathbf{a}$ usually implies $\otimes \mathbf{b}/\mathbf{a}$.

Several basic conditions for the above type are always true: $\otimes \mathbf{b}$, $\otimes \mathbf{c}$, $\otimes \mathbf{c} \wedge \mathbf{b}$, $\mathbf{a} \otimes \mathbf{b}$, $\mathbf{a} \otimes \mathbf{c}$, $\mathbf{b} \otimes \mathbf{c}$. As for a list $\mathbf{L}^{[c]}(\mathbf{A})$ the conditions $\otimes \zeta' \wedge \zeta$ (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 ignore them from now on. Conditions like $\mathbf{a} \otimes \mathbf{b}$, $\mathbf{b} \otimes \mathbf{c}$ are false in some contexts and will be kept explicitly in the $\text{Sep}(\mathbf{B})$ sets.

4.2 Typing judgements

Before tackling the annotation of typing judgements, we need to extend some type-related notions to typing contexts viewed as tuples of types. We will define type subterm addresses, separation conditions and a heap representation of value tuples and their portions and sub-portions related to typing contexts in a straightforward manner.

Definition 4.5 (Contexts as tuples of types).

Given a typing context Γ , define the set of its addresses

$$\text{AddrXRD}(\Gamma) = \{x\xi \mid \xi \in \text{AddrXRD}(\Gamma(x))\}.$$

For any $x\xi \in \text{AddrXRD}(\Gamma)$, let $\Gamma_{x\xi} = \Gamma(x)_\xi$ and if the context is annotated with portion names, let also $\zeta_{x\xi}^\Gamma = \zeta_\xi^{\Gamma(x)}$.

Also separation conditions $\text{BasicSep}(\Gamma)$ and $\text{Sep}(\Gamma)$ are defined analogously to $\text{BasicSep}(\mathbf{B})$ and $\text{Sep}(\mathbf{B})$ using the context addresses instead of type addresses.

We say that an environment S correctly represents a valuation η on a heap σ according to a typing context Γ (written $S \Vdash_\Gamma^\sigma \eta$) if it holds $S(x) \Vdash_{\Gamma(x)}^\sigma \eta(x)$ for every x in Γ .

Whenever this is so, we can use the following extended notation for regions, portions and subportions:

$$\begin{aligned} R_\Gamma(S, \sigma) &= R_{\Gamma(x)}(S(x), \sigma) \\ P_\Gamma^{x\xi}(S, \sigma) &= P_{\Gamma(x)}^\xi(S(x), \sigma) \\ P_\Gamma^{x\xi/x\xi_R}(S, \sigma) &= P_{\Gamma(x)}^{\xi/\xi_R}(S(x), \sigma) \end{aligned}$$

A representation $S \Vdash_{\Gamma; \mathbf{C}}^\sigma \eta$ satisfying a separation condition $\mathbf{C} \in \text{Sep}(\Gamma)$ has a meaning analogous to the meaning of $v \Vdash_{\mathbf{E}; \mathbf{C}}^\sigma \alpha$.

The following definitions of annotated typing judgements and their meaning follows the informal description given in the Introduction and in the beginning of this section.

Definition 4.6 (Syntax of typing judgements).

The sets of **annotated typing judgements** ZJudg are defined as follows:

$$\frac{\Gamma = x_1 : B_1, \dots, x_n : B_n \quad \mathbf{C} \in \text{Sep}(\Gamma) \quad \mathbf{E} \in \text{EType}, N(\mathbf{E}) \cap N(\Gamma) = \emptyset, N_{\mathbf{C}}(\mathbf{E}) \subseteq N_{\mathbf{D}}(\Gamma) \quad \mathbf{D} \subseteq N_{\mathbf{D}}(\Gamma) \quad \mathbf{G} \in \text{BasicSep}(\mathbf{E}) \rightarrow \text{Sep}(\Gamma)}{(\Gamma; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G}) \in \text{ZJudg}}$$

where $N_{\mathbf{D}}(\Gamma)$ and $N(\Gamma)$ are the unions of $N_{\mathbf{D}}(B_i)$ and $N(B_i)$, respectively, for all $i = 1, \dots, n$.

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

Definition 4.7 (Meaning of typing judgements).

A typing judgement $\Gamma; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G}$ is **operationally correct (OC)** if

- whenever it holds $S \Vdash_{\Gamma; \mathbf{C}}^\sigma \eta$ (separation preconditions) and $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ then it holds
 - (correctness and internal separation guarantees) $v \Vdash_{\mathbf{E}; \mathbf{C}_G}^{\sigma'} \llbracket e \rrbracket_{\eta, [P]}$ where

$$\mathbf{C}_G = \{s \in \text{BasicSep}(\mathbf{E}) \mid S \Vdash_{\Gamma; \mathbf{G}(s)}^\sigma \eta\}$$

– (containment guarantees) for each $\xi \in \text{AddrD}(\mathbb{E})$ with annotation $[\zeta_{\xi} \subseteq \gamma(\xi)]$

$$P_{\mathbb{E}}^{\xi}(\nu, \sigma') \subseteq \bigcup_{\zeta_{x\xi'} \in \gamma(\xi)} P_{\Gamma}^{x\xi'}(S, \sigma)$$

– (preservation guarantees) for each $x\xi \in \text{AddrD}(\Gamma)$ it holds

$$\sigma|_P = \sigma'|_P \text{ where } P = P_{\Gamma}^{x\xi}(S, \sigma) \setminus \bigcup_{\zeta_{x'\xi'} \in \mathbb{D}} P_{\Gamma}^{x'\xi'}(S, \sigma)$$

Consider the typing judgement for a predictably defined function which negates the first element of the argument list (if not empty):

$$x : L^{[a]}(\text{Bool}); \emptyset \vdash \text{neghead}(x) : L^{[b \subseteq \{a\}]}(\text{Bool}); \{a\}; \emptyset$$

Both the separation precondition and guarantee are empty because `BasicSep()` is empty for both the context and the result type. The only portion of the argument gets potentially destroyed.

The following function does the same as `neghead` to every element of a list of lists:

$$\begin{aligned} x : L^{[a]}(L^{[b]}(\text{Bool})); \{a \otimes b, \otimes b/a\} \vdash \\ \text{negheadlist}(x) : L^{[c \subseteq \{a\}]}(L^{[d \subseteq \{b\}]}(\text{Bool})); \{a, b\}; \quad c \otimes d \Leftarrow \{a \otimes b\} \\ \otimes d/c \Leftarrow \{\otimes b/a\} \end{aligned}$$

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 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.

The last simple example is binary tree mirroring:

$$\begin{aligned} x : T_b^{[a]}(L^{[b]}(\text{Bool})); \{a \otimes b, \otimes a\} \vdash \text{treflex}(x) : T_b^{[c \subseteq \{a\}]}(L^{[d \subseteq \{b\}]}(\text{Bool})); \{a\}; \\ \quad \otimes c \Leftarrow \{\otimes a\} \\ \quad \otimes d \wedge c \Leftarrow \{\otimes b \wedge a\} \\ \quad \otimes d/c \Leftarrow \{\otimes b/a\} \end{aligned}$$

which requires `\otimes a`, i.e. the skeleton of the tree must not overlap with itself.

Figure 5: Basic typing rules

<p>[UNIT]</p> $\frac{}{; \emptyset \vdash \mathbf{unit} : \mathbf{Unit}; \emptyset; \emptyset}$	<p>[WEAK]</p> $\frac{\Gamma; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G}}{\Gamma, x : \mathbf{B}; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G}}$	<p>[VAR]</p> $\frac{E \in \mathbf{E}_Y(\mathbf{B})}{x : \mathbf{B}; \emptyset \vdash x : \mathbf{E}; \emptyset; \mathbf{G}_Y(\mathbf{E}, \mathbf{B})}$
<p>[FUNC]</p> $\frac{\Gamma = \Gamma_f[\vec{x}/\vec{y}]}{\Gamma; \mathbf{C}_f \vdash f(x_1, \dots, x_{n_f}) : \mathbf{E}_f; \mathbf{D}_f; \mathbf{G}_f}$	<p>[RENAME]</p> $\frac{\Gamma; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G} \quad \tau : \mathbf{N}(\Gamma) \rightarrow \mathbf{Nm}}{\Gamma[\tau]; \mathbf{C}[\tau] \vdash e : \mathbf{E}[\tau]; \mathbf{D}[\tau]; \mathbf{G}[\tau]}$	
<p>[LET]</p> $\frac{\Gamma_1; \mathbf{C}_1 \vdash e_1 : \mathbf{A}[\frac{\delta_1, \gamma}{\rho_1}]; \mathbf{D}_1; \mathbf{G}_1 \quad \Gamma_2, x : \mathbf{A}[\frac{\delta_2}{\rho_2}]; \mathbf{C}_2 \vdash e_2 : \mathbf{E}; \mathbf{D}_2; \mathbf{G}_2}{\Gamma_1 \wedge \Gamma_2; \mathbf{C}_1 \cup \mathbf{T}(\mathbf{C}_2) \cup (\mathbf{D}_1 \otimes \mathbf{N}_D(\Gamma_2)) \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \mathbf{E}[\gamma/\delta_2]; \mathbf{D}_1 \cup \mathbf{D}_2[\gamma/\delta_2]; \mathbf{T} \circ \mathbf{G}_2}$ <p>where $\mathbf{T} = \mathbf{C} \mapsto ((\mathbf{C} \setminus \mathbf{Sep}x)[\gamma/\delta_2]) \cup \mathbf{G}_1((\mathbf{C} \cap \mathbf{Sep}x)[\delta_1/\delta_2, \rho_1/\rho_2])$ and $\mathbf{Sep}x = \mathbf{BasicSep}(\mathbf{A}[\frac{\delta_2}{\rho_2}])$.</p>		
<p>[PAIR]</p> $\frac{E \in \mathbf{E}_Y(\mathbf{B}_1 \times \mathbf{B}_2)}{x_1 : \mathbf{B}_1, x_2 : \mathbf{B}_2; \emptyset \vdash (x_1, x_2) : \mathbf{E}; \emptyset; \mathbf{G}_Y(\mathbf{E}, \mathbf{B}_1 \times \mathbf{B}_2)}$		
<p>[PAIR-ELIM]</p> $\frac{\Gamma, x_1 : \mathbf{B}_1, x_2 : \mathbf{B}_2; \mathbf{C} \vdash e : \mathbf{E}; \mathbf{D}; \mathbf{G}}{\Gamma, x : \mathbf{B}_1 \times \mathbf{B}_2; \mathbf{C} \vdash \mathbf{match} \ x \ \mathbf{with} \ (x_1, x_2) \Rightarrow e : \mathbf{E}; \mathbf{D}; \mathbf{G}}$		
<p>[INL]</p> $\frac{E_L \in \mathbf{E}_Y(\mathbf{B}_L) \quad E_R = \mathbf{A}[\frac{\delta, \xi \mapsto \emptyset}{\rho}]}{x : \mathbf{B}_L; \emptyset \vdash \mathbf{inL}(x) : E_L + E_R; \emptyset; \mathbf{G}_Y(E_L, \mathbf{B}_L)}$	<p>[INR]</p> $\frac{E_L = \mathbf{A}[\frac{\delta, \xi \mapsto \emptyset}{\rho}] \quad E_R \in \mathbf{E}_Y(\mathbf{B}_R)}{x : \mathbf{B}_R; \emptyset \vdash \mathbf{inR}(x) : E_L + E_R; \emptyset; \mathbf{G}_Y(E_R, \mathbf{B}_R)}$	
<p>[SUM-ELIM]</p> $\frac{\Gamma, x_L : \mathbf{B}_L; \mathbf{C}_L \vdash e_L : \mathbf{E}_L; \mathbf{D}_L; \mathbf{G}_L \quad \Gamma, x_R : \mathbf{B}_R; \mathbf{C}_R \vdash e_R : \mathbf{E}_R; \mathbf{D}_R; \mathbf{G}_R}{\Gamma, x : \mathbf{B}_L + \mathbf{B}_R; \mathbf{C}_L \cup \mathbf{C}_R \vdash \mathbf{case} \ x \ \mathbf{of} \ \mathbf{inL}(x_L) \Rightarrow e_L \mathbf{inR}(x_R) \Rightarrow e_R : \mathbf{E}_L \cup \mathbf{E}_R; \mathbf{D}_L \cup \mathbf{D}_R; \mathbf{G}_L \cup \mathbf{G}_R}$		
<p>[PUT]</p> $\frac{E \in \mathbf{E}_Y(\diamond^{[\zeta]} \mathbf{B})}{x : \mathbf{B}, d : \diamond^{[\zeta]}; \{\zeta\} \otimes \mathbf{N}_D(\mathbf{B}) \vdash x@d : \mathbf{E}; \{\zeta\}; \mathbf{G}_Y(\mathbf{E}, \diamond^{[\zeta]} \mathbf{B})}$		
<p>[GET]</p> $\frac{E \in \mathbf{E}_Y(\mathbf{B})}{x : \diamond^{[\zeta]} \mathbf{B}; \emptyset \vdash \mathbf{get}(x) : \mathbf{E}; \emptyset; \mathbf{G}_Y(\mathbf{E}, \mathbf{B})}$	<p>[LOC]</p> $\frac{}{x : \diamond^{[\zeta]} \mathbf{B}; \emptyset \vdash \mathbf{loc}(x) : \diamond^{[\zeta' \subseteq \{\zeta\}]}; \emptyset; \emptyset}$	

Figure 6: Typing of fold and unfold

[FOLD]

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

where

$$G' = \left[\otimes \zeta_{\xi} \wedge \zeta_{\bar{\mu}_X} \leftarrow \left\{ \zeta_{x\xi'\xi} \otimes \zeta_{x\xi''\xi} \mid \xi' \bar{X}, \xi'' \bar{X} \in \text{Addr}X_X(A) \right\} \right]_{\xi \in \text{AddrD}(A)} \text{ and}$$

$$G'' = \left[\otimes \zeta_{\xi} / \zeta_{\bar{\mu}_X} \leftarrow \text{ditto} \cup \left\{ \zeta_{x\xi'\xi} \otimes \zeta_{x\xi} \mid \xi' \bar{X} \in \text{Addr}X_X(A) \right\} \right]_{\xi \in \text{AddrD}(A)}$$

[UNFOLD]

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

where

$$G' = \left[\zeta_{x\xi'\xi} \otimes \zeta_{x\xi''\xi} \leftarrow \left\{ \otimes \zeta_{\xi} \wedge \zeta_{\bar{\mu}_X} \right\} \right]_{\xi \in \text{AddrD}(A), \xi' \bar{X}, \xi'' \bar{X} \in \text{Addr}X_X(A)} \text{ and}$$

$$G'' = \left[\zeta_{x\xi'\bar{\mu}_\xi} \otimes \zeta_{x\xi} \leftarrow \left\{ \otimes \zeta_{\xi} / \zeta_{\bar{\mu}_X} \right\} \right]_{\xi \in \text{AddrD}(A), \xi' \bar{X} \in \text{Addr}X_X(A)}$$

5 Typing rules

The rules are summarised in Figs. 5 and 6. The new notation used there will be described next.

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, \dots, y_{n_f} in this order.

Given a result type E , we define the minimal default guarantee for E as follows:

$$G_M(E) = [\zeta_1 \otimes \zeta_2 \leftarrow Z_1 \otimes_D Z_2]_{\zeta_1 \otimes \zeta_2 \in \text{BasicSep}(E)}$$

where Z_1, Z_2 appear in the following annotations in E : $[\zeta_1 \subseteq Z_1], [\zeta_2 \subseteq Z_2]$; and \otimes_D is one of the following two similar operations:

$$Z_1 \otimes Z_2 = \{ \zeta_1 \otimes \zeta_2 \mid \zeta_1 \in Z_1, \zeta_2 \in Z_2 \}$$

$$Z_1 \otimes_D Z_2 = \{ \zeta_1 \otimes \zeta_2 \mid \zeta_1 \in Z_1, \zeta_2 \in Z_2, \zeta_1 \neq \zeta_2 \}$$

The second one differs from the first one by assuming that whenever the same portion name is used in two different containment sets, then this portion could be divided into two disjoint portions which could replace the two occurrences of the original portion while keeping the containment guarantee. The fact that $Z_1 \otimes_D Z_2$ is used instead of $Z_1 \otimes Z_2$ in $G_M(E)$ 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 pre-conditions.

Several rules use a “copied” type $E \in E_Y(B)$ for (part of) the result type. If $B = A \left[\frac{\delta}{\rho} \right] \in \text{BType}$, then

$$E_Y(B) = \left\{ A \left[\frac{\delta', \xi \mapsto \{\delta(\xi)\}}{\rho'} \right] \mid \text{any valid } \delta', \rho' \right\} \subset \text{EType}.$$

With a copied type $E \in E_Y(B)$ go its default guarantees $G_Y(E, B)$ given by the following formula:

$$G_Y(E, B) = G_M(E) \cup \left[\otimes \zeta_{\xi}^E / \zeta_{\xi_R}^E \leftarrow \{ \otimes \zeta_{\xi}^B / \zeta_{\xi_R}^B \} \right]_{\otimes \zeta_{\xi}^E / \zeta_{\xi_R}^E \in \text{BasicSep}(E)} \\ \cup \left[\otimes \zeta_{\xi}^E \wedge \zeta_{\xi_R}^E \leftarrow \{ \otimes \zeta_{\xi}^B \wedge \zeta_{\xi_R}^B \} \right]_{\otimes \zeta_{\xi}^E \wedge \zeta_{\xi_R}^E \in \text{BasicSep}(E)}$$

In [FOLD] and [SUM-ELIM], we construct the result type by a E-type union. If $E_1, E_2 \in E\text{Type}$, the union $E_1 \cup E_2$ is defined if the types differ only in the γ -part of the annotation ($E_1 = A\left[\frac{\delta, \gamma_1}{\rho}\right], E_2 = A\left[\frac{\delta, \gamma_2}{\rho}\right]$) as the type $A\left[\frac{\delta, \gamma_1 \cup \gamma_2}{\rho}\right]$ where $\gamma_1 \cup \gamma_2$ is the point-wise union of images. In several places also various guarantee-condition functions are merged (e.g. $G_L \cup G_R$) 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 x that appears in 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.

Given a substitution $\tau: N(\Gamma) \rightarrow Nm$, we can perform it on an annotated type E , a separation condition C , a guarantee-condition mapping G as well as a set D in the obvious way. The results are denoted $E[\tau]$, $C[\tau]$, $G[\tau]$ and $D[\tau]$, respectively. Analogous substitutions can be done for $\tau: N(\Gamma) \rightarrow \wp(Nm)$ using set union where necessary.

When $\delta: \text{AddrD}(A) \rightarrow Nm$ is injective and $\gamma: \text{AddrD}(A) \rightarrow \wp(Nm)$, then we derive a set-valued substitution $\gamma/\delta: Nm \rightarrow \wp(Nm)$ as $\gamma \circ \delta^{-1}$. 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 condition set C , i.e. one containing \perp or $\zeta \otimes \zeta$ for some portion name ζ . Similarly, when an additional separation guarantee s is unachievable, its precondition set $G(s)$ would contain an unsatisfiable basic separation condition.

6 Correctness

A typing rule is OC if whenever all the premises of an instance of the rule are OC then so is the conclusion. Let us assume that all the typing judgements $P(f)$ are OC.

Proposition 6.1. *All the rules in Figs. 5 and 6 are OC.*

Proof outline. The proof is conceptually simple but looks complicated due to the complexity of the annotations. We will therefore show the proof in a greater detail only for the rule [LET] which is among the most intricate ones.

Assume $S \Vdash_{\Gamma_1 \wedge \Gamma_2; C_1 \cup T(C_2) \cup (D_1 \otimes N_D(\Gamma_2))}^{\sigma} \eta$ and $S, \sigma \vdash \text{let } x = e_1 \text{ in } e_2 \rightsquigarrow v, \sigma''$. It follows from definition of \rightsquigarrow that $S, \sigma \vdash e_1 \rightsquigarrow v_x, \sigma'$ and $S[x \mapsto v_x], \sigma' \vdash e_2 \rightsquigarrow v, \sigma''$. By simple restriction we get $S_1 \Vdash_{\Gamma_1; C_1}^{\sigma} \eta_1$ where $S_1 := S|_{\text{Dom}(\Gamma_1)}$ and $\eta_1 := \eta|_{\text{Dom}(\Gamma_1)}$. Now we can use the correctness of the first premise and obtain the separation guarantees G_1 as well as the containment γ for portions of x named by δ_1 and the destruction limitation D_1 in addition to $v_x \Vdash_A^{\sigma} a$ where $a := \llbracket e_1 \rrbracket_{\eta_1, \llbracket P \rrbracket}$.

Relative to the second premise, we can again derive $S_2 \Vdash_{\Gamma_2; C_{2\bar{x}}}^{\sigma} \eta_2$ where $S_2 := S|_{\text{Dom}(\Gamma_2)}$, $\eta_2 := \eta|_{\text{Dom}(\Gamma_2)}$ and

$$C_{2\bar{x}} := \{\perp, \zeta_1 \otimes \zeta_2, \otimes \zeta_1 / \zeta, \otimes \zeta_1 \wedge \zeta \in C_2 \mid \zeta \notin \text{Ran}(\rho_2), \zeta_1, \zeta_2 \notin \text{Ran}(\delta_2)\}.$$

Nevertheless, to be able to use the second premise, we need to show $S_{2x} \Vdash_{\Gamma_{2x}; C_2}^{\sigma'} \eta_{2x}$ where $S_{2x} := S_2[x \mapsto v_x]$, $\Gamma_{2x} := \Gamma_2, x : A \left[\frac{\delta_2}{\rho_2} \right]$ and $\eta_{2x} := \eta_2[x \mapsto a]$. For this, it remains to prove $S_{2x} \Vdash_{\Gamma_{2x}; C_{2x}}^{\sigma'} \eta_{2x}$ and $S_{2x} \Vdash_{\Gamma_{2x}; C_{2xx}}^{\sigma'} \eta_{2x}$ where

$$C_{2x} := \{\zeta_1 \otimes \zeta_2 \in C_2 \mid \zeta_1 \in \text{Ran}(\delta_2), \zeta_2 \notin \text{Ran}(\delta_2)\}$$

and $C_{2xx} := \{\zeta_1 \otimes \zeta_2, \otimes \zeta_1 / \zeta, \otimes \zeta_1 \wedge \zeta \in C_2 \mid \zeta \in \text{Ran}(\rho_2), \zeta_1, \zeta_2 \in \text{Ran}(\delta_2)\}$

because $C_2 = C_{2\bar{x}} \cup C_{2x} \cup C_{2xx}$. The former of these two statements follows from the first part of $T(C_2)$ for $\Gamma_1 \wedge \Gamma_2$ on σ and the containment of each portion $\delta_2(\xi)$ within the portions $\gamma(\xi)$. The latter statement follows from the second part of $T(C_2)$ using the validity of the rely-guarantees G_1 .

We have proved $S_{2x} \Vdash_{\Gamma_{2x}; C_2}^{\sigma'} \eta_{2x}$ and conclude $v \Vdash_E^{\sigma''} \llbracket e_2 \rrbracket_{\eta_{2x}, \llbracket P \rrbracket}$ as well as the containment, preservation and separation guarantees expressed in E , D_2 and G_2 , respectively. It remains to prove that also the guarantees $E[\gamma/\delta_2]$, $D_1 \cup D_2[\gamma/\delta_2]$ and $T \circ G_2$ are valid with respect to the concluding typing judgement. The proofs of containment and preservation guarantees are straightforward. For every basic separation predicate s , we assume $S \Vdash_{\Gamma_1 \wedge \Gamma_2; (T \circ G_2)(s)}^{\sigma} \eta$ and prove $S_{2x} \Vdash_{\Gamma_{2x}; G_2(s)}^{\sigma'} \eta_{2x}$ in exactly the same way as we did for C_2 from $T(C_2)$. Using G_2 we finish by concluding $v \Vdash_{E; s}^{\sigma''} \llbracket e_2 \rrbracket_{\eta_{2x}, \llbracket P \rrbracket}$. \square

7 Inference

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 an iterative process as follows.

- Annotate every unannotated signature Γ_f, A_f with portion names and the strongest conditions and rely-guarantees: $C_f := \emptyset$, $E_f := A_f \left[\frac{-, \xi \mapsto \emptyset}{-} \right]$, $D_f := \emptyset$ and $G_f := [s \longleftarrow \emptyset]$ getting an annotated program P_0 .
- Infer the annotation for all e_f using P_0 and translate the derived conditions and rely-guarantees to annotations of P forming another annotated program P_1 .
- Repeat the process, obtaining a sequence $\{P_i\}$ of annotated versions of the program P until a fixpoint $P_k = P_{k+1}$ is found and return P_k .

We have studied this process in a more abstract setting in [Konečný 2002] and showed that it will always terminate with the strongest derivable annotation for \mathbb{P} provided that all the typing rules are monotone in the following sense:

Definition 7.1. A rule is **monotone** if in any instance by weakening the annotation of the premises we get an instance whose conclusion is equivalent or weaker than the original one.

A typing judgement

$$\Gamma; C \vdash e : A[\frac{\delta, \gamma}{\rho}]; D; G$$

is **equivalent or weaker** than

$$\Gamma'; C' \vdash e' : A'[\frac{\delta', \gamma'}{\rho'}]; D'; G'$$

if $\Gamma = \Gamma'$, $e = e'$, $A = A'$, $\delta = \delta'$, $\rho = \rho'$, $\forall \xi. \gamma'(\xi) \subseteq \gamma(\xi)$, $D' \subseteq D$ and whenever $S \Vdash_{\Gamma; C}^{\sigma} \eta$ then $S \Vdash_{\Gamma'; C'}^{\sigma} \eta$ and whenever $S \Vdash_{\Gamma; G(s)}^{\sigma} \eta$ then $S \Vdash_{\Gamma'; G'(s)}^{\sigma} \eta$ for every basic condition s .

The notion of being **syntactically equivalent or weaker** is defined for typing judgements analogously like above but replacing the conditions on C, C', G, G' with the following: $C' \subseteq C$ and $\forall s. G'(s) \subseteq G(s)$.

Analogously we define when a rule is **syntactically monotone**.

Proposition 7.2. All the rules in Figs. 5 and 6 are syntactically monotone.

Proof outline. The rules without premises are monotone trivially. Only rules [WEAK], [LET], [PAIR-ELIM] and [SUM-ELIM] have any premises. All the constructors of annotation C , γ , D and $G(s)$ in the conclusions of the rules are monotone with respect to subset inclusion. \square

Lemma 7.3. If a judgement J is syntactically equivalent or weaker than J' then J is equivalent or weaker than J' .

The opposite of this lemma does not hold because for some types (e.g. $L^{[a]}(L^{[b]}(\text{Bool}))$) there are tautological separation conditions (e.g. $\mathbf{a} \otimes \mathbf{b}$). Another reason for the failure is that some conditions are consequences of others, in particular of \perp . We ruled out some of these tautological conditions (e.g. $\otimes \zeta$ for lists) but not all because their characterisation is not trivial and would further complicate the system. Nevertheless, the following weaker opposite of the above lemma holds and is sufficient for us:

Lemma 7.4. If

$$J \equiv \Gamma; C \vdash e : A[\frac{\delta, \gamma}{\rho}]; D; G$$

is equivalent or weaker than

$$J' \equiv \Gamma'; C' \vdash e' : A'[\frac{\delta', \gamma'}{\rho'}]; D'; G'$$

then there exist C'' and G'' such that $J'' \equiv \Gamma; C'' \vdash e : A[\frac{\delta, \gamma}{\rho}]; D'; G''$ is equivalent to J' and J is syntactically equivalent or weaker than J'' .

Lemma 7.5. All the rules preserve semantical equivalence of typing judgements.

Corollary 7.6. All the rules in Figs. 5 and 6 are monotone.

Figure 7: Derived rules for list term constructors

$$\begin{array}{c}
 \text{[NIL]} \\
 \frac{E = L^{[\zeta \subseteq \emptyset]}(A[\frac{\delta, \xi \mapsto \emptyset}{\rho}])}{; \emptyset \vdash \text{nil}_A : E; \emptyset; G_M(E)} \\
 \\
 \text{[CONS]} \\
 \frac{B_h = A[\frac{\delta}{\rho}] \quad B_t = L^{[\zeta]}(A[\frac{\delta'}{\rho'}]) \quad E = L^{[\zeta' \subseteq \{\zeta_d\}]}(E_h) \cup E_t \quad E_h \in E_Y(B_h) \quad E_t \in E_Y(B_t)}{h : B_h, t : B_t, d : \diamond^{[\zeta_d]}; \{\zeta_d\} \otimes (N_D(B_h) \cup N_D(B_t)) \vdash \text{cons}(h, t)@d : E; \{\zeta_d\}; G} \\
 \text{where } G = G_M(E) \cup G_Y(E_h, B_h) \cup G_Y(E_t, B_t) \\
 \quad \cup [\otimes \zeta_\xi / \zeta' \leftarrow \{\delta(\xi) \otimes \delta'(\xi)\}]_{\xi \in \text{AddrD}(A)} \\
 \\
 \text{[MATCH-LIST]} \\
 \frac{\Gamma; C_1 \vdash e_1 : E_1; D_1; G_1 \quad \Gamma, h : A[\frac{\delta}{\rho}], t : L^{[\zeta]}(A[\frac{\delta'}{\rho'}]), d : \diamond^{[\zeta_d]}; C_2 \vdash e_2 : E_2; D_2; G_2}{\Gamma, x : L^{[\zeta]}(A[\frac{\delta}{\rho}]); C \vdash \text{match } x \text{ with nil} \Rightarrow e_1 | \text{cons}(h, t)@d \Rightarrow e_2 : E; D; G} \\
 \text{where } C = C_1 \cup T(C_2)[\zeta/\zeta_d][\delta/\delta'][\rho/\rho'], \\
 T(C) = (C \setminus \{\zeta \otimes \zeta_d\}) [(\otimes \delta(\xi)/\zeta) / (\delta(\xi) \otimes \delta'(\xi))]_{\xi \in \text{AddrD}(A)}, \\
 E = E_1 \cup (E_2[\zeta/\zeta_d][\delta/\delta']), \\
 G = G_1 \cup (T \circ G_2) \\
 \text{and } D = D_1 \cup D_2[\zeta/\zeta_d][\delta/\delta'].
 \end{array}$$

8 Examples

To make type checking of programs with lists easier, we put down the typing rules for the list term constructors in Fig. 7. Rules for trees could be derived in a similar way.

The following typing judgements have been generated by a Haskell implementation of the typing rules in Fig. 5 and the derived rules for binary trees and the rules for lists in Fig. 7. The annotations have been automatically deduced by the program using the iterative process described in Sect. 7.

The judgements in Figs. 8 and 9 are the components of the in-place binary tree reversal algorithm as described in the Introduction. Due to the lack of parametric polymorphism, the example cannot involve types with an unbound variable like it did in the Introduction. Therefore we substitute a simple non-heap-free type $L(\text{Bool})$ in place of the type meta-variable A .

Although we defined binary trees with labels on both nodes and leaves, we ignore the labels on leaves in this algorithm. It is crucial here that leaves are not heap-free and their locations can be reused for cons-cells of the resulting list.

The annotation of *paths* tells us about the function that in order to work correctly, it needs the input tree to have non-overlapping control structure disjoint from the labels. The cons-cells of the result list as well as the cons-cells of the lists in it are constructed solely from the control structure of the argument tree which is also the only argument portion being

Figure 8: Tree reversal top level

$$\begin{aligned}
 \text{paths} = & x : T_b^{[a]}(L^{[b]}(\text{Bool})); \{a \otimes b, \otimes a\} \vdash \\
 & \text{let } n = \text{nil}_{L^\square(\text{Bool})} \text{ in } \text{pathsaux}(x, n) \\
 & : L^{[c \subseteq \{a\}]}(L^{[d \subseteq \{a\}]}(L^{[e \subseteq \{b\}]}(\text{Bool}))); \{a\}; \\
 & c \otimes d \leftarrow \{\otimes a\} \\
 & c \otimes e \leftarrow \{a \otimes b\} \\
 & d \otimes e \leftarrow \{a \otimes b\} \\
 & \otimes d / c \leftarrow \{\perp, \otimes a\} \\
 & \otimes e / c \leftarrow \{\perp, \otimes b / a\} \\
 & \otimes e / d \leftarrow \{\otimes b / a\}
 \end{aligned}$$

Figure 9: Tree reversal recursion

$$\begin{aligned}
 \text{pathsaux} = & t : T_b^{[a]}(L^{[b]}(\text{Bool})), p : L^{[c]}(L^{[d]}(\text{Bool})); \\
 & \{a \otimes b, a \otimes c, a \otimes d, \otimes a\} \vdash \\
 & \text{match } t \text{ with } \text{leaf}_b(a) @ d \Rightarrow \text{let } n = \text{nil}_{L^\square(L^\square(\text{Bool}))} \\
 & \quad \text{in } \text{cons}(p, n) @ d \\
 & \quad | \text{node}_b(v, l, r) @ d \Rightarrow \text{let } vp = \text{cons}(v, p) @ d \\
 & \quad \quad \text{in } \text{let } s1 = \text{pathsaux}(l, vp) \\
 & \quad \quad \quad \text{in } \text{let } s2 = \text{pathsaux}(r, vp) \\
 & \quad \quad \quad \text{in } \text{append}(s1, s2) \\
 & : L^{[e \subseteq \{a\}]}(L^{[f \subseteq \{a, c\}]}(L^{[g \subseteq \{b, d\}]}(\text{Bool}))); \{a\}; \\
 & e \otimes f \leftarrow \{a \otimes c, \otimes a\} \\
 & e \otimes g \leftarrow \{a \otimes b, a \otimes d\} \\
 & f \otimes g \leftarrow \{a \otimes b, a \otimes d, b \otimes c, c \otimes d\} \\
 & \otimes f / e \leftarrow \{\perp, a \otimes c, \otimes a\} \\
 & \otimes g / e \leftarrow \{\perp, b \otimes d, \otimes b / a\} \\
 & \otimes g / f \leftarrow \{b \otimes d, \otimes b / a, \otimes d / c\}
 \end{aligned}$$

destroyed. Thus the label data is not modified on the heap at all.

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

9 Conclusion

We have enhanced the typing system of Aspinall & Hofmann [2002] following the natural idea that data and control structure should be treated separately. This led to a, perhaps surprisingly, complicated system of annotations. Despite using complex notation, the system provides efficient annotation inference which is, in fact, an efficient static analysis of memory interference.

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. For some problem areas the present language might not be sufficiently expressive. It should be possible, though, to adapt the annotation system in this report to other language features, including arrays, directed graphs, higher order functions and various resource types.

Also, it could be used for functional languages without the \diamond type that have a fairly straightforward heap-based evaluation strategy. For example, recently Hofmann, Jost [2002] and Kirli developed a static analysis of memory consumption in a \diamond -free version of LFPL with implicit memory allocation and deallocation. Combining the present system with the above would contribute to another area of further research which is to extend the present system with a resource usage analysis.

9.1 Related work

The theory of *Shapely types* by Jay [1995] treats inductive and other types as containers explicitly. Thus the idea to treat differently the control (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 static shape analysis on the semantical level. In contrast, we develop a static 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 [1997a] using the notation `Boxed(-)`. This idea appears often in the context of the strongly-typed intermediate language for compilation of functional languages *FLINT* [Shao 1997b, League & Shao 1998].

The use of pre- and post-conditions for certifying in-place update with sharing is not new. Recent work includes *Alias types* [Walker & Morrisett 2000, Walker & Morrisett 2001] 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 [Reynolds 2002] may serve a similar purpose as alias types but for higher level imperative 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 pre-state, we believe that it is expressive enough to encode our assertions. Nevertheless, such an encoding might not be very natural.

Our system bears some resemblance to the *Region inference* of Tofte & Talpin [1997]. 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 difference is that the regions of Tofte & 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 [Guzmán & Hudak 1990] and also to *passivity* [O’Hearn et al. 1999] within syntactic control of interference [Reynolds 1978, Reynolds 1989]. These aspects can be also viewed as *effects* (in the sense of [Lucassen & Gifford 1988]) somewhat similar to *get* and *put* of Tofte & Talpin [1997]. 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. The author is grateful to David Aspinall, Robert Atkey and Martin Hofmann for discussion and comments on this work.

References

- Aspinall, D. & Hofmann, M. [2002], Another type system for in-place update, *in* D. L. Métyayer, ed., ‘Programming Languages and Systems, Proceedings of 11th European Symposium on Programming’, Springer-Verlag, pp. 36–52. Lecture Notes in Computer Science 2305. 2, 24
- Guzmán, J. C. & Hudak, P. [1990], Single-threaded polymorphic lambda calculus, *in* ‘Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science’, pp. 333–343. 25
- Hofmann, M. [2000], ‘A type system for bounded space and functional in-place update’, *Nordic Journal of Computing* 7(4), 258–289.
URL: citeseer.nj.nec.com/hofmann00type.html 2
- Hofmann, M. [2002], The strength of non size-increasing computation, *in* ‘Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science’. 2

- Jay, C. B. [1995], 'A semantics for shape', *Science of Computer Programming* **25**(2–3), 251–283.
URL: citeseer.nj.nec.com/jay95semantics.html 24
- Jost, S. [2002], Static prediction of dynamic space usage of linear functional programs, Master's thesis, Technische Universität Darmstadt, Fachbereich Mathematik. 24
- Konečný, M. [2002], Typing with conditions and guarantees in LFPL, Technical Report EDI-INF-RR-0151, LFCS, Division of Informatics, University of Edinburgh. 3, 21
- League, C. & Shao, Z. [1998], Formal semantics of the FLINT intermediate language, Technical Report Yale-CS-TR-1171, Department of Computer Science, Yale University. 24
- Lucassen, J. M. & Gifford, D. K. [1988], Polymorphic effect systems, in 'Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages', ACM Press, pp. 47–57. 25
- O'Hearn, P. W., Power, A. J., Takeyama, M. & Tennent, R. D. [1999], 'Syntactic control of interference revisited', *Theoretical Computer Science* **228**, 211–252. 25
- Reynolds, J. C. [1978], Syntactic control of interference, in 'Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, pp. 39–46. 25
- Reynolds, J. C. [1989], Syntactic control of interference, part 2, in G. Ausiello, M. Dezani-Ciancaglini & S. R. D. Rocca, eds, 'Automata, Languages and Programming, 16th International Colloquium', Springer-Verlag, pp. 704–722. Lecture Notes in Computer Science 372. 25
- Reynolds, J. C. [2002], Separation logic: A logic for shared mutable data structures, in 'Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science'. 25
- Shao, Z. [1997a], Flexible representation analysis, in 'Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)', Amsterdam, The Netherlands, pp. 85–98. 24
- Shao, Z. [1997b], An overview of the FLINT/ML compiler, in 'Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)', Amsterdam, The Netherlands. 24
- Tofte, M. & Talpin, J.-P. [1997], 'Region-based memory management', *Information and Computation* **132**(2), 109–176. 25
- Walker, D. & Morrisett, G. [2000], Alias types, in 'ESOP 2000', pp. 366–381. Lecture Notes in Computer Science 1782. 24
- Walker, D. & Morrisett, G. [2001], Alias types for recursive data structures, in 'Types in Compilation 2000', pp. 177–206. Lecture Notes in Computer Science 2071. 24