Typing with Conditions and Guarantees in LFPL

Michal Konečný
LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
mkonecny@dcs.ed.ac.uk

October 15, 2002

Abstract

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.

We investigate a general approach to easing the tight linear restrictions of LFPL by generalising the static analysis used by Aspinall and Hofmann in [Aspinall & Hofmann 2002]. It consists in devising new type systems for the core language of LFPL whose judgements express sets of rely-guarantee pairs with assertions about the operational semantics, namely about the heap representation of the arguments and the result. The method can be applied to other functional languages with heap based operational semantics.

As an example and application of the approach we reformulate the language of [Aspinall & Hofmann 2002] from this perspective and prove the correctness of some of its crucial typing rules, show its type inference algorithm and the existence of principal types relative to the simple LFPL typing. We refer to other languages based on LFPL which fit into the approach.

1 Introduction

First order LFPL [Hofmann 2000] is an example of 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 also 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:

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

In the denotational semantics of the program, the argument d of cons is (virtually) ignored in both cases and thus the semantics is list concatenation as expected. Nevertheless, the operational semantics will append the lists in-place, rewriting the first list's last cons-cell if the list is not empty. (The other cons-cells of the first list would be overwritten with the same content). The third argument of cons corresponds to the heap address of the cons-cell in the operational semantics.

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 *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 $append_A(x, y)$ correct under certain condition on how lists x and y are represented. A sufficient condition for $append_A(x, y)$ being OC is that the heap region occupied by x is separated (disjoint) from the 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 to show correctness of a bigger term of which this is a subterm.

LFPL (Linear Function 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. In addition to this restriction, LFPL maintains the invariant that different variables 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 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 or has to be separated from it. This idea is similar to that of *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].

It turned out that it was apparently rather hard to prove the correctness of the typing rules given in [Aspinall & Hofmann 2002]. The paper contains only an outline of a correctness proof. The details were provided by the present author. In this process, more insight into the UAPL typing has been attained via a precise interpretation of the usage aspects as expressing certain conditions and guarantees (i.e. pre- and post-conditions). This development including a part of the correctness proof is contained in Sect. 3.

Another contribution of this section is a simple iterative algorithm deciding for a simply typed program whether it can be correctly annotated with usage aspects (UAPL type-checking) and if so, inferring its strongest correct usage aspects. In particular, every given plainly typed program either has a strongest correct UAPL annotation or has none.

Before defining and studying UAPL, we present the approach in an abstract setting in Sect. 2. After UAPL we refer to two of its extensions in Sect. 4.

2 Rely-Guarantees about Heap Representation

Before we formalise the main ideas, we fix some notation concerning an unspecified *functional* typed language $\mathcal{L}$, following [Aspinall & Hofmann 2002]. In Sect. 3 we will instantiate $\mathcal{L}$ as well as all the concepts below with the example of UAPL.

Let e range over the expressions and A over the types of $\mathcal{L}$ and assume some typing rules that define valid typing judgements of the form $\Gamma \vdash e : A$ where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is called a typing context. A pair Γ, A is called a (term) signature.

A program P over $\mathcal{L}$ is a map from a finite set of function symbols to typing judgements that capture the signature and definition of each symbol. The expressions of $\mathcal{L}$ within a well-typed program may contain calls to the functions of P including recursive calls. A tuple of expressions is given as arguments to a function symbol in accordance with its signature.

Let a denotational semantics $[\![A]\!]$ for each type A be given. Also let a denotation of programs $[\![P]\!]$ and terms $[\![e]\!]_{\eta,[\![P]\!]}$ with a valuation η of variables (including the ones that are free in e) be defined by a least fixed point as usual.

Lastly, assume that an *imperative* operational semantics is defined as a 5-ary relation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ where

- σ, σ': Loc $\rightharpoonup$ HVal are heaps,
- S: Var $\rightharpoonup$ SVal is an environment,
- Loc is a set of heap locations (ranged over by ℓ) and
- SVal is a set of stack values (e.g. numerals, tuples) ranged over by v, Loc $\subseteq$ SVal,
- HVal is a set of heap values h (e.g. cons cells) which contain SVal values
- Var is the set of variables of $\mathcal{L}$.

The intuitive meaning of the above relation is: with the environment S and the heap σ the term e evaluates to v and changes the heap to σ'.

A basic operational value v may refer to a heap σ and together with the heap represent a denotational value $a \in [\![A]\!]$, e.g. a list or a tree. This relationship is formalised by the relation $v \Vdash^{\sigma}_{A} a$ and extended to tuples by the relation $S \Vdash^{\sigma}_{\Gamma} \eta$.

2.1 General CG-Systems

Definition 2.1. *A conditions and guarantees (CG-) system $\mathcal{CG}$ for the language $\mathcal{L}$ is a mapping which assigns to each signature Γ, A a pair of sets $\mathrm{Cond}_{\mathcal{CG}}(\Gamma)$ and $\mathrm{Guar}_{\mathcal{CG}}(A, \Gamma)$ whose elements are of the following form:*

- $C \in \mathrm{Cond}_{\mathcal{CG}}(\Gamma)$—*a subset (understood as a predicate) of* $\{(S, \sigma) \mid (\exists \eta)(S \Vdash^{\sigma}_{\Gamma} \eta)\}$
  (I.e. *a pre-condition on the heap representation of the arguments.)*

- $G \in \mathrm{Guar}_{\mathcal{CG}}(A, \Gamma)$—*a subset of* $\{(\nu, \sigma', S, \sigma) \mid (\exists a, \eta)(\nu \Vdash^{\sigma'}_{A} a \wedge S \Vdash^{\sigma}_{\Gamma} \eta)\}$
  (I.e. *a post-condition on the heap representation of the result and its relationship to the representation of the arguments.)*

We will write $S, \sigma \Vdash C$ *and* $\nu, \sigma', S, \sigma \Vdash G$ *for* $(S, \sigma) \in C$ *and* $(\nu, \sigma', S, \sigma) \in G$*, respectively.*

A pair $(C, G) \in \mathrm{Cond}_{\mathcal{CG}}(\Gamma) \times \mathrm{Guar}_{\mathcal{CG}}(A, \Gamma)$ *is called a* CG-pair *for* Γ, A*.*

The sets of predicates $\mathrm{Cond}_{\mathcal{CG}}(\Gamma)$ and $\mathrm{Guar}_{\mathcal{CG}}(\Gamma, A)$ will be usually considered in the context of an evaluation $S, \sigma \vdash e \rightsquigarrow \nu, \sigma'$ where the term e satisfies $\Gamma \vdash e : A$. The predicates are then applied on the heaps and values from the evaluation relation.

Definition 2.2. *A typing judgement* $\Gamma \vdash e : A$ annotated by a CG-pair C, G *for* Γ, A *(written as* $\Gamma; C \vdash e : A; G$*) is* OC *if*
whenever $S, \sigma \vdash e \rightsquigarrow \nu, \sigma'$ *with* $S \Vdash^{\sigma}_{\Gamma} \eta$ *and* $S, \sigma \Vdash C$
then $\nu \Vdash^{\sigma'}_{A} [\![e]\!]_{\eta, [\![P]\!]}$ *and* $\nu, \sigma', S, \sigma \Vdash G$*.*

The correct evaluation of e relies on the precondition C which typically is separation between (parts of) arguments on the heap. Extra guarantees inside G may be of several kinds:

- *Separation* between parts of the result on the result heap
- *Containment* of (parts of) the result within the union of the regions of certain (parts of the) arguments
- *Non-destruction* of certain (parts of the) arguments

Each of such guarantees may rely on some extra precondition like those in $\mathrm{Cond}_{\mathcal{CG}}(\Gamma)$.

Each of the predicate sets $\mathrm{Cond}_{\mathcal{CG}}(\Gamma)$ or $\mathrm{Guar}_{\mathcal{CG}}(\Gamma, A)$ is partially ordered by logical implication, e.g. $C \leqslant C' \iff (\forall S, \sigma)(S, \sigma \Vdash C \iff S, \sigma \Vdash C')$.

Definition 2.3. *Define an order on the set* $\mathrm{Cond}_{\mathcal{CG}}(\Gamma) \times \mathrm{Guar}_{\mathcal{CG}}(A, \Gamma)$ *contra-variantly element-wise:* $(C, G) \leqslant (C', G') \iff C \geqslant C'$ *and* $G \leqslant G'$

Obviously, if a typing judgement annotated with (C', G') is OC then it is also OC if it is annotated with $(C, G) \leqslant (C', G')$ instead.

2.2 CG-Typing

Fix a CG-system $\mathcal{CG}$ for $\mathcal{L}$ and use it implicitly for the rest of this section. Having defined the semantics of CG-pairs, we now outline the way CG-pairs can be used as a part of the typing, obtaining *annotated typing rules* which will yield the annotated typing judgements from the previous section. We will denote by $\mathcal{L}'$ the language $\mathcal{L}$ with the new annotated typing rules.

Definition 2.4. *A* CG-typing *of* $\mathcal{L}$ *consists of*

- *an assignment* $(\Gamma, A) \mapsto \mathcal{W}(\Gamma, A) \subseteq \mathrm{Cond}(\Gamma) \times \mathrm{Guar}(A, \Gamma)$ *of* valid *CG-pairs for each signature of* $\mathfrak{L}$
- *the typing rules of* $\mathfrak{L}$ *each of which is* annotated *in the following way:*
  - *Each typing judgement* $\Gamma_i \vdash e_i : A_i$ *in the premises is associated with an unspecified pair* $(C_i, G_i) \in \mathcal{W}(\Gamma_i, A_i)$
  - *A side condition on the pairs* (C_i, G_i) *might be added.*
  - *A procedure is indicated how to derive a pair* $(C, G) \in \mathcal{W}(\Gamma, A)$ *for the concluded typing judgement* solely *from the pairs* (C_i, G_i) *provided that the side condition holds (if any).*
- *extra trivial typing rules with* $\Gamma \vdash e : A$ *on both sides annotated as above.*

Thus rules take the form:

$$[\textsc{gen}]\quad \frac{\Gamma_1; C_1 \vdash e_1 : A_1; G_1 \quad \cdots \quad \Gamma_n; C_n \vdash e_n : A_n; G_n}{\Gamma; C(C_1, G_1, \ldots, C_n, G_n) \vdash e[e_1, \ldots, e_n] : A; G(C_1, G_1, \ldots, C_n, G_n)}$$

The actual syntax of adding CG-pairs to the rules is left open at the moment. In particular systems, there may be a more concise way to represent the CG-pairs. For example, see Sect. 3 and 4.

Definition 2.5. *A typing rule in a CG-typing is* correct *if in any instantiation of the rule it holds: all annotated typing judgements from the premises are OC*
$\Longrightarrow$ *the concluded annotated typing judgement is OC.*

A CG-typing is correct if all its rules are correct.

Let us outline a typical proof that a CG-typing rule of the form [GEN] is correct:

1. Assume that $S_0, \sigma_0 \vdash e[e_1, \ldots, e_n] \rightsquigarrow \nu_0, \sigma'$, $S_0 \Vdash^{\sigma_0}_{\Gamma} \eta_0$ and $S_0, \sigma_0 \Vdash C$ hold.
2. Prove $S_1, \sigma_0 \vdash e_{j_1} \rightsquigarrow \nu_1, \sigma_1$, $S_1 \Vdash^{\sigma_0}_{\Gamma_{j_1}} \eta_1$ and $S_1, \sigma_0 \Vdash C_{j_1}$ for some $j_1 \in \{1, \ldots, n\}$ (given by the evaluation strategy for e, i.e. e_{j_1} is the first subterm to be evaluated).
3. Deduce $\nu_1 \Vdash^{\sigma_1}_{A_{j_1}} [\![e_{j_1}]\!]_{\eta_1, [\![P]\!]}$ and $\nu_1, \sigma_1, S_1, \sigma_0 \Vdash G_{j_1}$.
4. Proceed similarly to steps 2 and 3 with other typing judgements in the assumptions (led by the evaluation strategy for e) introducing a sequence $\{(j_i, S_i, \sigma_i, \eta_i, \nu_i)\}_{0<i\leqslant m}$ with $\sigma_m = \sigma'$.
5. Prove $\nu \Vdash^{\sigma'}_{A} [\![e[e_1, \ldots, e_n]]\!]_{\eta, [\![P]\!]}$ and $\nu_0, \sigma', S_0, \sigma_0 \Vdash G$—Q.E.D.

2.3 Inference of Annotation

Let us now turn to the type-checking problem which is now a combination of the ordinary type-checking of $\mathcal{L}$ with the inference of annotation. Annotation is inferred in the course of the type-checking and sometimes a certain branch of the type-checking is barred by a side condition.

When there is more than one sequence of typing rule applications for a certain term (resulting in the same typing judgement) then there may be more than one CG-pair derivable for the judgement. In such a case, we should be able to take the resulting CG-pair to be the supremum of the derived pairs in the implication order. This will happen in UAPL because there is a single strongest derivation despite some non-determinism in the typing rules.

Special attention has to be paid to the typing of *recursion* via function calls. An *annotated program* P would now associate an OC typing judgement $\Gamma_f; C_f \vdash e_f : A_f; G_f$ to every function symbol f. The problem is now obvious that (C_f, G_f) cannot be derived by the typing rules directly due to their potential dependency on (C_f, G_f). These have to be derived as some fixpoint of their circular definition. Under some conditions, they can be derived iteratively as the *strongest fixpoint* as follows: Associate the strongest CG-pair (C_f^0, G_f^0) in $\mathcal{W}(\Gamma_f, A_f)$ with each function symbol f and then derive possibly weaker pairs (C_f^1, G_f^1) for them via the typing of their terms e_f. Repeat this with (C_f^1, G_f^1) to derive (C_f^2, G_f^2) and so on until $(C_f^k, G_f^k) = (C_f^{k+1}, G_f^{k+1})$ for all f.

For this iteration to work, the derivation of typing judgements has to be *monotone*; i.e. whenever the function signatures' annotation is weakened, also the derived annotation is weakened. This is equivalent to all typing rules being monotone in the following sense: In any pair of instances of the rule differing only in their CG-pairs ($W_1, \ldots, W_n, W$ versus $W'_1, \ldots, W'_n, W'$) it holds:

$$\big((\forall i)(W_i \leqslant W'_i)\big) \Longrightarrow W \leqslant W'.$$

By the term *"inferring strongest annotation"* we mean deciding whether a given well-typed program or term in $\mathcal{L}$ can be annotated within a given CG-typing and if affirmative also computing its strongest correct annotation.

Theorem 2.6. *If in a CG-typing*

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

then the iterative process described above infers the strongest annotation for any program P.

3 UAPL

We present the language from [Aspinall & Hofmann 2002] as an instance of the framework from the previous section.

3.1 The Language

The types and terms are in essence those of first-order LFPL [Hofmann 2000]:

$$
\begin{array}{rcl}
A & ::= & \Diamond \mid \mathsf{Bool} \mid A_1 \times A_2 \mid A_1 \otimes A_2 \mid \mathsf{L}(A) \\
e & ::= & x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid f(x_1, \ldots, x_n) \\
 & \mid & \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & \mid & (e_1, e_2) \mid \mathsf{fst}(x) \mid \mathsf{snd}(x) \\
 & \mid & x_1 \otimes x_2 \mid \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \Rightarrow e \\
 & \mid & \mathsf{nil}_A \mid \mathsf{cons}_A(x_h, x_t, x_d) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil}_A \Rightarrow e_1 | \mathsf{cons}_A(x_h, x_t, x_d) \Rightarrow e_2
\end{array}
$$

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

The language term constructors $\mathsf{tt}, \mathsf{ff}, \mathsf{fst}, \mathsf{snd}, \_\otimes\_, \mathsf{nil}_A, \mathsf{cons}_A$ are treated as special pre-defined function symbols and could, therefore, be omitted above. Nevertheless, we keep them in the grammar to make it more comprehensible.

The plain typing of the terms is standard apart from the rules for lists:

$$
\frac{}{\vdash \mathsf{nil}_A : \mathsf{L}(A)}\ [\textsc{nil}] \qquad \frac{}{x_h : A, x_t : \mathsf{L}(A), x_d : \Diamond \vdash \mathsf{cons}(x_h, x_t, x_d) : \mathsf{L}(A)}\ [\textsc{cons}]
$$

$$
[\textsc{match-list}]\quad \frac{\Gamma \vdash e_1 : A' \quad \Gamma, x_h : A, x_t : \mathsf{L}(A), x_d : \Diamond \vdash e_2 : A'}{\Gamma, x : \mathsf{L}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil}_A \Rightarrow e_1 | \mathsf{cons}_A(x_h, x_t, x_d) \Rightarrow e_2 : A'}
$$

in which the extra argument of cons is given the LFPL type $\Diamond$. This argument does not play any role in the denotational semantics but is crucially used in the imperative operational semantics as a heap address.

The denotational semantics is given as hinted in Sect. 2; in particular:

$$
[\![\mathsf{nil}_A]\!]_{\eta,[\![P]\!]} = [] \qquad [\![\mathsf{cons}_A(x_h, x_t, x_d)]\!]_{\eta,[\![P]\!]} = \eta(x_h) :: \eta(x_t)
$$

The values stored in environments and in heap locations are the following:

$$
v ::= \ell \mid \mathsf{tt} \mid \mathsf{ff} \mid (v_1, v_2) \mid \mathsf{nil} \qquad h ::= \{\mathsf{hd} = v_1, \mathsf{tl} = v_2\}
$$

Figure 1: Definition of Evaluation Relation

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

[FUNC]
$$\frac{S(x_i) = v_i \quad [x_i \mapsto v_i], \sigma \vdash e_f \rightsquigarrow v, \sigma'}{S, \sigma \vdash f(x_1, \ldots, x_n) \rightsquigarrow v, \sigma'}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where ℓ ranges over the heap locations from Loc. The definition of the evaluation relation is in Fig. 1. The rules for the heap representation relation $v \Vdash^{\sigma}_{A} a$ are more or less derivable from the evaluation rules, we quote only the ones for lists here:

$$\frac{}{\mathrm{nil} \Vdash^{\sigma}_{L(A)} []} \qquad \frac{\sigma(\ell) = \{\mathrm{hd} = v_1, \mathrm{tl} = v_2\} \quad v_1 \Vdash^{\sigma}_{A} h \quad v_2 \Vdash^{\sigma}_{L(A)} t}{\ell \Vdash^{\sigma}_{L(A)} h :: t}$$

An important aspect of the heap representation definition is whether the two parts of a tensor product are required to be separated from each other on the heap or not. We will not require it in the default representation relation because there are valid situations when a tensor product may not be represented as a separated product.

Let $v \mathbin{^{\otimes}\Vdash^{\sigma}_{A}} a$ be a relation which differs from $v \Vdash^{\sigma}_{A} a$ only by adding a condition to the tensor product and cons-cell rules requiring that the components are separated:

$$\frac{v_1 \mathbin{^{\otimes}\Vdash^{\sigma}_{A_1}} a_1 \quad v_2 \mathbin{^{\otimes}\Vdash^{\sigma}_{A_2}} a_2 \quad R_{A_1}(v_1, \sigma) \cap R_{A_2}(v_2, \sigma) = \varnothing}{(v_1, v_2) \mathbin{^{\otimes}\Vdash^{\sigma}_{A_1 \otimes A_2}} (a_1, a_2)}$$

$$\frac{\sigma(\ell) = \{\mathrm{hd} = v_1, \mathrm{tl} = v_2\} \quad v_1 \mathbin{^{\otimes}\Vdash^{\sigma}_{A}} h \quad v_2 \mathbin{^{\otimes}\Vdash^{\sigma}_{L(A)}} t \quad R_A(v_1, \sigma) \cap R_{L(A)}(v_2, \sigma) = \varnothing}{\ell \mathbin{^{\otimes}\Vdash^{\sigma}_{L(A)}} h :: t}$$

where $R_A(v, \sigma)$ is the *region* of the value of type A represented by v on σ, i.e. the set of locations in σ "reachable" from v.

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

Lemma 3.1. *For every evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ with the representation of arguments $S \Vdash^{\sigma}_{\Gamma} \eta$ and result $v \Vdash^{\sigma'}_{A} a$, it holds that:*

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

This lemma can be proved easily by induction on the structure of e from the definition of evaluation relation and region.

3.2 Usage Aspects

The syntax of annotated typing judgements is:

$$x_1 :^{i_1} A_1, \ldots, x_n :^{i_n} A_n \vdash e : A$$

where each i_j, called *usage aspect*, is from the set $\{1, 2, 3\}$. The usage aspects can be informally interpreted as follows:

Figure 2: UAPL typing rules

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

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

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

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

[IF]
$$\frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2 \vdash e_2 : A}{\Gamma_1 \wedge \Gamma_2, x :^{3} \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A}$$

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

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

[LIST-ELIM]
$$\frac{\Gamma_1 \vdash e_1 : A' \quad \Gamma_2, h :^{i_h} A, t :^{i_t} L(A), d :^{i_d} \Diamond \vdash e_2 : A'}{\Gamma_1 \wedge \Gamma_2, x :^{\min(i_h, i_t, i_d)} L(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil}_A \Rightarrow e_1 | \mathsf{cons}_A(h, t, d) \Rightarrow e_2 : A'}$$

- 1: *condition*: argument is separated from all the others
  condition: arguments' tensor products are separated products
  guarantee: none (argument could be even destroyed)

- 2: *condition*: argument separated from others with aspect 1 or 2
  condition: arguments' tensor products are separated products
  guarantee: argument preserved during evaluation

- 3: *condition*: argument separated from others with aspect 1
  guarantee: argument preserved and separated from the result apart from its portions shared with arguments of aspect 2

This is not the full story as some rules will require an alternative interpretation of the usage aspects to be OC. We will formalise their meaning as a CG-pair in Subsect. 3.3.

UAPL typing rules (apart from the obvious rule for function calls) are shown in Fig. 2 and the annotated typing judgements of the predefined functions in Fig. 3.

The notation $\Gamma[x]$ stands for the aspect of x in Γ and Γ^{i} stands for the context which arises from Γ by changing any usage aspect 2 in Γ to i. The joined context $\Gamma_1 \wedge \Gamma_2$ is defined when

Figure 3: Usage aspects of predefined functions

$\vdash \mathsf{tt}, \mathsf{ff} : \mathsf{Bool}$ $\qquad x :^2 A_1 \times A_2 \vdash \mathsf{fst}(x) : A_1$ $\qquad x :^2 A_1 \times A_2 \vdash \mathsf{snd}(x) : A_2$

$x_1 :^2 A_1, x_2 :^2 A_2 \vdash x_1 \otimes x_2 : A_1 \otimes A_2$

$\vdash \mathsf{nil}_A : \mathsf{L}(A)$ $\qquad h :^2 A, t :^2 \mathsf{L}(A), d :^1 \Diamond \vdash \mathsf{cons}_A(h, t, d) : \mathsf{L}(A)$

Table 1: Usage Aspects as a CG-pair

Condition	Guarantee $\Longleftarrow$ Rely
	$\mathrm{pres}(\lvert\Gamma\rvert_{2,3}) \Longleftarrow$
$\mathrm{sep}(\lvert\Gamma\rvert_1, \lvert\Gamma\rvert) \wedge \mathrm{tens}(\lvert\Gamma\rvert_1)$	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma\rvert_{1,2}) \Longleftarrow$
	$\mathrm{tens}(r) \Longleftarrow \mathrm{sep}(\lvert\Gamma\rvert_2, \lvert\Gamma\rvert_2) \wedge \mathrm{tens}(\lvert\Gamma\rvert_2)$

the contexts are compatible (i.e. $\Gamma_1(x) = \Gamma_2(x)$ for every $x \in |\Gamma_1| \cap |\Gamma_2|$ where $|\Gamma|$ is the set of all variables in Γ) and annotations are calculated as the minimum where the contexts Γ_1, Γ_2 overlap:

$$\forall x \in |\Gamma_1| \cap |\Gamma_2|.(\Gamma_1 \wedge \Gamma_2)[x] = \min(\Gamma_1[x], \Gamma_2[x]).$$

The rules [DROP] and [RAISE] would be trivial without the usage aspects. These rules introduce nondeterminism to the type-checking by allowing the annotation to be weakened at any time and strengthened at certain situations. As we will show that all the rules are monotone (Prop. 3.3) and stable under strengthening of premises (Prop. 3.4), there is a deterministic type-checking strategy yielding the strongest annotation for each OC typing judgement. It consists of giving [RAISE] priority over all other rules and the opposite for [DROP]: use it only if needed to make premises match each other. Thus Thm. 2.6 will apply to UAPL.

3.3 Correctness

Let us now formalise the meaning of usage aspects in typing judgements as sets of CG-pairs. Fix an annotated typing judgement $\Gamma \vdash e : A$ and a matching operational judgement $S, \sigma \vdash e \rightsquigarrow v, \sigma'$.

The CG-pair (denoted $W_\Gamma = (C_\Gamma, G_\Gamma)$) that the usage aspects represent is summarised in Tab. 1 using the following notation (sep stands for *separation*, pres for *preservation* and tens for *tensor product separation*):

$$
\begin{aligned}
|\Gamma|_{i,j} &= |\Gamma|_i \cup |\Gamma|_j\,, \quad |\Gamma|_i = \{x \in |\Gamma| \mid \Gamma[x] = i\} \\
\mathrm{sep}(x,y) &\equiv \mathrm{rg}(x) \cap \mathrm{rg}(y) = \varnothing, \quad \mathrm{sep}(T,T') \equiv \textstyle\bigwedge_{x\in T, x'\in T', x\neq x'} \mathrm{sep}(x,x') \\
\mathrm{rg}(x) &= R_{\Gamma(x)}(S(x),\sigma), \\
\mathrm{tens}(x) &\equiv (\exists a)\big(S(x) \mathrel{{}^{\otimes}\!\Vdash^{\sigma}_{\Gamma(x)}} a\big), \quad \mathrm{tens}(T) = \textstyle\bigwedge_{x\in T} \mathrm{tens}(x) \\
\mathrm{pres}(x) &\equiv (\forall \ell \in \mathrm{rg}(x))\,\big(\sigma(\ell) = \sigma'(\ell)\big), \quad \mathrm{pres}(T) = \textstyle\bigwedge_{x\in T} \mathrm{pres}(x)
\end{aligned}
$$

The symbol r is treated as a special variable representing the result on the heap σ', i.e. $S(r) = v$, $\Gamma(r) = A$ and in relation to r the heap σ' is used instead of σ.

This CG-pair can be decomposed into two weaker but simpler CG-pairs that are equivalent together to the original CG-pair. In the first one, the rely precondition of the third guarantee moved to the condition while in the second one the third guarantee is removed.

We shall call the first simpler CG-pair the primary interpretation and the other one the secondary interpretation of the usage aspects. Notice that the primary interpretation corresponds to the intuitive interpretation of the usage aspects as given earlier.

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

Lemma 3.2. *If $\Gamma \leqslant \Gamma'$ then $W_\Gamma \leqslant W_{\Gamma'}$* (i.e. $C_\Gamma \geqslant C_{\Gamma'}$ *and* $G_\Gamma \leqslant G_{\Gamma'}$).

As a result of this lemma, there is a strongest annotation for each typing judgement, namely the one with aspect 3 at all the arguments. The last ingredients to be able to apply Theorem 1 to UAPL are monotonicity, stability under strengthening of premises and correctness of the rules and of the annotation of the predefined functions in Fig 3. These will be the content of the following four propositions.

Proposition 3.3. *The UAPL typing rules are monotone.*

Proof. For each rule, assume its OC instance using the notation from Fig. 2. Then take the same premises with some collection of usage aspects which are not higher than the original ones. Use primed notation for new usage aspects and annotated contexts. Then construct the new conclusion to make a new instance of the rule (if possible). Since the rules are just annotated versions of valid ordinary typing rules, the conclusions will differ only in the usage aspects. For each rule we have to show that the new conclusion is weaker or equal to the old one. We show it here for a few rules only:

[DROP]. $\Gamma', x :^{\min(i',j)} A$ is weaker than $\Gamma, x :^{j} A$.

[RAISE]. $(\Gamma')^3 \leqslant \Gamma^3$

[LET]. $(\Gamma_1')^{i'} \wedge \Gamma_2'$ is weaker than $\Gamma_1^i \wedge \Gamma_2$ because $(\Gamma_1')^{i'} \leqslant (\Gamma_1')^i \leqslant \Gamma_1^i$.

[$\times$-INTRO]. $\Gamma_1' \wedge \Gamma_2' \leqslant \Gamma_1 \wedge \Gamma_2$

[$\otimes$-ELIM]. $\Gamma', x :^{\min(i_1',i_2')} A_1 \otimes A_2 \leqslant \Gamma, x :^{\min(i_1,i_2)} A_1 \otimes A_2$ □

Proposition 3.4. *The UAPL typing rules are stable under strengthening of premises,* i.e. *if certain premises enable a rule, then the same premises with stronger usage aspects also enable the rule.*

Proof. This is trivial for all rules but [DROP], [×-INTRO] and [LET] as these are the only ones having side conditions that depend on the annotation of premises. Raising the i in [DROP] keeps the side condition valid. The [×-INTRO] side-condition is clearly stable under raising the usage aspects and so is the [LET] side condition as 3 is the highest usage aspect. □

Proposition 3.5. *The UAPL typing rules are OC.*

Proof. [DROP]. The correctness of this rule follows from the earlier observation that lowering a usage aspect results in weakening the corresponding CG-pair.

[RAISE]. Assume $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ with $S \Vdash^{\sigma}_{\Gamma} \eta$ and that each region $\mathrm{rg}(x)$ is separated from all others and the tensor products of x are separated whenever $\Gamma[x] = 1$. Thus the condition of the premise is met and using its correctness we get $v \Vdash^{\sigma'}_{A} [\![e]\!]_{\eta,[\![P]\!]}$ and $\mathrm{pres}(|\Gamma|_{2,3})$. The latter implies $\mathrm{pres}(|\Gamma^3|_{2,3})$ and as the result is heap-free, we have $\mathrm{rg}(r) = \varnothing$ and consequently it trivially holds $\mathrm{tens}(r)$ and $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma^3|_{1,2})$.

[WEAK]. Assume $S_x, \sigma \vdash e \rightsquigarrow v, \sigma'$, $S_x \Vdash^{\sigma}_{\Gamma_x} \eta_x$ and $S_x, \sigma \Vdash C_{\Gamma_x}$ where $S_x = S[x \mapsto v_x]$, $\eta_x = \eta[x \mapsto a_x]$ and $\Gamma_x = \Gamma, x :^3 A$. Then clearly it holds $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, $S \Vdash^{\sigma}_{\Gamma} \eta$ and $S, \sigma \Vdash C_{\Gamma}$ which allows us to use the premise. The guarantee $v, \sigma', S, \sigma \Vdash G_{\Gamma}$ implies $v, \sigma', S_x, \sigma \Vdash G_{\Gamma_x}$ because its only part involving x is $\mathrm{pres}(x)$ which holds due to Lemma 3.2 (3) and $\mathrm{sep}(|\Gamma|_1, \{x\})$.

[LET]. The skeleton of this proof for the primary interpretation is in Tab. 2. Each line of the table contains a statement preceded by its name and followed by a list of references to statements and definitions from which it has been derived. The statements whose names end with *i1* are valid under the condition that $i = 1$, etc. A proof of the secondary interpretation can be obtained from the first proof by minor modifications as indicated in the table.

[IF]. Put $\Gamma := \Gamma_1 \wedge \Gamma_2, x :^3 \mathsf{Bool}$ and assume $S \Vdash^{\sigma}_{\Gamma} \eta$ as well as $S, \sigma \Vdash C_{\Gamma}$. Also assume $S_i, \sigma \vdash e_i \rightsquigarrow v_i, \sigma_i$ where $S_i := S|_{|\Gamma_i|}$ for $i = 1, 2$. Clearly $S_i \Vdash^{\sigma}_{\Gamma_i} \eta_i$ where $\eta_i := \eta|_{|\Gamma_i|}$ and by Lemma 3.2 we get also $S_i, \sigma \Vdash C_{\Gamma_i}$.

If $S(x) = \mathsf{tt}$, set $j = 1$, otherwise $j = 2$, so that $S, \sigma \vdash e \rightsquigarrow v_j, \sigma_j$ where $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Using the correctness of the premise for e_j we get $v_j \Vdash^{\sigma_j}_{A} [\![e]\!]_{\eta,[\![P]\!]}$ and $v_j, \sigma_j, S, \sigma \Vdash G_{\Gamma_j}$. The first guarantee $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma|_{1,2})$ follows from $|\Gamma_j|_{1,2} \subseteq |\Gamma|_{1,2}$. The second one, $\mathrm{pres}(|\Gamma|_{2,3})$, follows from $\mathrm{pres}(|\Gamma_j|_{2,3})$, $\mathrm{sep}(|\Gamma_j|_1, |\Gamma|_{2,3})$ and Lemma 3.1 (3). Finally, if $\mathrm{sep}(|\Gamma|_2, |\Gamma|_2)$ and $\mathrm{tens}(|\Gamma|_2)$, then the same precondition holds for Γ_j because $|\Gamma_j|_2 \subseteq |\Gamma|_{1,2}$ and thus we get $\mathrm{tens}(r)$ from G_{Γ_j}.

[×-INTRO]. Assume $S, \sigma \vdash (e_1, e_2) \rightsquigarrow v, \sigma''$ which implies $S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma'$, $S, \sigma' \vdash e_2 \rightsquigarrow v_2, \sigma''$ and $v = (v_1, v_2)$. Further assume $S \Vdash^{\sigma}_{\Gamma} \eta$ and $S, \sigma \Vdash C_{\Gamma}$ where $\Gamma = \Gamma_1 \wedge \Gamma_2$. We get $S_i \Vdash^{\sigma}_{\Gamma_i} \eta_i$ where where $S_i := S|_{|\Gamma_i|}$ and $\eta_i := \eta|_{|\Gamma_i|}$ and from Lemma 3.2 also $S_i, \sigma \Vdash C_{\Gamma_i}$ for $i = 1, 2$. Applying the hypothesis that the first premise is OC, we get $v_1 \Vdash^{\sigma'}_{A_1} [\![e_1]\!]_{\eta,[\![P]\!]}$ and $v_1, \sigma', S_1, \sigma \Vdash G_{\Gamma_1}$, especially $\mathrm{pres}(|\Gamma_1|_{2,3})$.

Now we can conclude that $S_2 \Vdash^{\sigma'}_{\Gamma_2} \eta_2$ using $\mathrm{sep}(|\Gamma_1|_1, |\Gamma|)$, Lemma 3.1 (3) and the side condition that Γ_2 does not contain any variable which has aspect 1 in Γ_1. This and $S, \sigma \Vdash C_{\Gamma_2}$ yields $S, \sigma' \Vdash C_{\Gamma_2}$. Thus we can use the hypothesis of correctness for the second premise to obtain $v_2 \Vdash^{\sigma''}_{A_2} [\![e_2]\!]_{\eta,[\![P]\!]}$ and $v_2, \sigma'', S_2, \sigma' \Vdash G_{\Gamma_2}$.

Table 2: Correctness of [LET]

Implicit conditions: $(\dots i1)\ {\sim}(i = 1)$, $(\dots i12)\ {\sim}(i \in \{1,2\})$, etc.
Secondary interpretation: remove text in $\square$ and rename:
$(C1i12) \mapsto (C1i1)$, $(C1i3) \mapsto (C1i23)$, $(G1i12) \mapsto (G1i1)$, $(G1i3) \mapsto (G1i23)$.

(*OC1,2*)	$\Gamma_1 \vdash e_1 : A$ is OC, $\quad \Gamma_2, x :^i A \vdash e_2 : A'$ is OC	assume
(*Op*)	$S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v, \sigma''$	assume
(*Op1,2*)	$S, \sigma \vdash e_1 \rightsquigarrow v_x, \sigma' \quad S[x \mapsto v_x], \sigma' \vdash e_2 \rightsquigarrow v, \sigma''$	*Op*,$\rightsquigarrow$
(*HA*)	$S \Vdash^{\sigma}_{\Gamma} \eta \quad (\Gamma := \Gamma_1^i \wedge \Gamma_2)$	assume
(*1-23*)	$\lvert\Gamma_1\rvert_1 \cup \lvert\Gamma_2\rvert_1 \subseteq \lvert\Gamma\rvert_1 \quad \lvert\Gamma\rvert_{2,3} \subseteq \lvert\Gamma_1\rvert_{2,3} \cup \lvert\Gamma_2\rvert_{2,3}$	$\Gamma = \Gamma_1^i \wedge \Gamma_2$
(*C*)	$S, \sigma \Vdash \mathrm{sep}(\lvert\Gamma\rvert_1, \lvert\Gamma\rvert) \wedge \mathrm{sep}(\lvert\Gamma\rvert_2, \lvert\Gamma\rvert_2) \wedge \mathrm{tens}(\lvert\Gamma\rvert_{1,2})$	assume
(*HA1*)	$S_1 \Vdash^{\sigma}_{\Gamma_1} \eta_1 \quad (S_1 := S\lvert_{\lvert\Gamma_1\rvert}, \eta_1 := \eta\lvert_{\lvert\Gamma_1\rvert})$	*HA*
(*C1i12*)	$S_1, \sigma \Vdash \mathrm{sep}(\lvert\Gamma_1\rvert_1, \lvert\Gamma_1\rvert) \wedge \mathrm{sep}(\lvert\Gamma_1\rvert_2, \lvert\Gamma_1\rvert_2) \wedge \mathrm{tens}(\lvert\Gamma_1\rvert_{1,2})$	*C*,Γ_1^i
(*C1i3*)	$S_1, \sigma \Vdash \mathrm{sep}(\lvert\Gamma_1\rvert_1, \lvert\Gamma_1\rvert) \wedge \mathrm{tens}(\lvert\Gamma_1\rvert_1)$	*C*,Γ_1^i
(*HR1*)	$v_x \Vdash^{\sigma'}_{A} a \quad (a := [\![e_1]\!]_{\eta_1, [\![P]\!]})$	*OC1*
(*G1i12*)	$v_x, \sigma', S_1, \sigma \Vdash \mathrm{pres}(\lvert\Gamma_1\rvert_{2,3}) \wedge \mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_1\rvert_{1,2}) \wedge \mathrm{tens}(r)$	*OC1*
(*G1i3*)	$v_x, \sigma', S_1, \sigma \Vdash \mathrm{pres}(\lvert\Gamma_1\rvert_{2,3}) \wedge \mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_1\rvert_{1,2})$	*OC1*
(*HA2p*)	$S_2 \Vdash^{\sigma}_{\Gamma_2} \eta_2 \quad (S_2 := S\lvert_{\lvert\Gamma_2\rvert}, \eta_2 := \eta\lvert_{\lvert\Gamma_2\rvert})$	*HA*
(*HA2g*)	$S_2 \Vdash^{\sigma'}_{\Gamma_2} \eta_2$	*HA2p,C,1-23,G1*
(*HA2*)	$S_x \Vdash^{\sigma'}_{\Gamma_x} \eta_x \quad (S_x := S_2[x \mapsto v_x], \eta_x := \eta_2[x \mapsto a], \Gamma_x := \Gamma_2, x :^i A)$	*HA2g,HR1*
(*C2g*)	$S_2, \sigma' \Vdash \mathrm{sep}(\lvert\Gamma_2\rvert_1, \lvert\Gamma_2\rvert) \wedge \mathrm{sep}(\lvert\Gamma_2\rvert_2, \lvert\Gamma_2\rvert_2) \wedge \mathrm{tens}(\lvert\Gamma_2\rvert_{1,2})$	*C,G1*
(*C2xi1*)	$S_x, \sigma' \Vdash \mathrm{sep}(x, \lvert\Gamma_2\rvert) \wedge \mathrm{tens}(x)$	*C,G1i12*,Γ_1^i,sidecond
(*C2xi2*)	$S_x, \sigma' \Vdash \mathrm{sep}(x, \lvert\Gamma_2\rvert_{1,2}) \wedge \mathrm{tens}(x)$	*C,G1i12*,$\Gamma_1^i \wedge \Gamma_2$,sidecond
(*C2xi3*)	$S_x, \sigma' \Vdash \mathrm{sep}(x, \lvert\Gamma_2\rvert_1)$	*C*,sidecond
(*C2*)	$S_x, \sigma' \Vdash \mathrm{sep}(\lvert\Gamma_x\rvert_1, \lvert\Gamma_x\rvert) \wedge \mathrm{sep}(\lvert\Gamma_x\rvert_2, \lvert\Gamma_x\rvert_2) \wedge \mathrm{tens}(\lvert\Gamma_x\rvert_{1,2})$	*C2g,C2x*
(*HR2*)	$v \Vdash^{\sigma''}_{A'} [\![e_2]\!]_{\eta_x, [\![P]\!]}$	*OC2*
(*G2*)	$v, \sigma'', S_x, \sigma' \Vdash \mathrm{pres}(\lvert\Gamma_x\rvert_{2,3}) \wedge \mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_x\rvert_{1,2}) \wedge \mathrm{tens}(r)$	*OC2*
(*HR*)	$v \Vdash^{\sigma''}_{A'} [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]_{\eta, [\![P]\!]}$	*HR2*,η_x,denot.sem.
(*S23*)	$S, \sigma \Vdash \mathrm{sep}(\lvert\Gamma_2\rvert_{2,3}, \lvert\Gamma_1\rvert_1) \quad S, \sigma' \Vdash \mathrm{sep}(\lvert\Gamma_1\rvert_{2,3}, \lvert\Gamma_2\rvert_1)$	*1-23,C*
(*Si1*)	$\lvert\Gamma\rvert_{2,3} \cap \lvert\Gamma_1\rvert \subseteq \lvert\Gamma_1\rvert_3, \quad S_1, \sigma \Vdash \mathrm{sep}(\lvert\Gamma_1\rvert_3, \lvert\Gamma_1\rvert_2)$	$\Gamma = \Gamma_1^i \wedge \Gamma_2$,*C*
(*P*)	$v, \sigma'', S, \sigma \Vdash \mathrm{pres}(\lvert\Gamma\rvert_{2,3})$	*G1,G2,S23,Si1*
(*Ri12*)	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_x\rvert_{1,2}) \subseteq \mathrm{rg}(\lvert\Gamma_2\rvert_{1,2}) \cup \mathrm{rg}(\lvert\Gamma_1\rvert_{1,2}) = \mathrm{rg}(\lvert\Gamma\rvert_{1,2})$	*G2,G1*
(*Ri3*)	$\mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma_x\rvert_{1,2}) = \mathrm{rg}(\lvert\Gamma_2\rvert_{1,2}) \subseteq \mathrm{rg}(\lvert\Gamma\rvert_{1,2})$	*G2*
(*G*)	$v, \sigma'', S, \sigma \Vdash \mathrm{pres}(\lvert\Gamma\rvert_{2,3}) \wedge \mathrm{rg}(r) \subseteq \mathrm{rg}(\lvert\Gamma\rvert_{1,2}) \wedge \mathrm{tens}(r)$	*P,R,G2*
(*OC*)	$\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : A'$ is OC	*Op,HA,C,HR,G*

Since $R_{A_1}(v_1, \sigma') \subseteq R_{\Gamma_1}(S|_{|\Gamma_1|}, \sigma)$ (Lemma 3.1, point 2), the region $R_{A_1}(v_1, \sigma')$ has not been modified by the evaluation of e_2 thanks to the condition that all variables with aspect 1 in Γ_2 have regions disjoint from all others. Thus we also have $v_1 \Vdash^{\sigma''}_{A_1} [\![e_1]\!]_{\eta,[\![P]\!]}$ and $v_1, \sigma'', S_1, \sigma \Vdash G_{\Gamma_1}$. Now we can deduce $(v_1, v_2) = v \Vdash^{\sigma''}_{A_1 \times A_2} [\![(e_1, e_2)]\!]_{\eta,[\![P]\!]}$.

From $v_i, \sigma'', S_i, \sigma \Vdash G_{\Gamma_i}$ it is easy to deduce $\mathrm{pres}(|\Gamma|_{2,3})$ and $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma|_{1,2})$ using $|\Gamma|_{2,3} \subseteq |\Gamma_1|_{2,3} \cup |\Gamma_2|_{2,3}$ and $|\Gamma_1|_{1,2} \cup |\Gamma_2|_{1,2} \subseteq |\Gamma|_{1,2}$.

When $\mathrm{sep}(|\Gamma|_2, |\Gamma|_2)$ and $\mathrm{tens}(|\Gamma|_2)$ hold on σ then the same holds for Γ_1 and Γ_2 on their respective heaps σ and σ'. Thus v_1 and v_2 are represented on σ'' with separated tensor products. As there is no tensor product between v_1 and v_2, we get $\mathrm{tens}(r)$.

[$\otimes$-ELIM]. Assume $S, \sigma \vdash e' \rightsquigarrow v, \sigma'$ where $e' := \mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \Rightarrow e$ which implies $S(x) = (v_1, v_2)$ and $S'', \sigma \vdash e \rightsquigarrow v, \sigma'$ where $S'' := S[x_1 \mapsto v_1][x_2 \mapsto v_2]$. Further assume $S \Vdash^{\sigma}_{\Gamma'} \eta$ where $\Gamma' := \Gamma, x :^i A_1 \otimes A_2$. Put $\Gamma'' := \Gamma, x_1 :^{i_1} A_1, x_2 :^{i_2} A_2$.

Assuming $S, \sigma \Vdash C_{\Gamma'}$, we aim at proving $S'', \sigma \Vdash C_{\Gamma''}$. Basic conditions confined within $|\Gamma|$ translate directly from $C_{\Gamma'}$ to $C_{\Gamma''}$. Since $i \leqslant i_1, i_2$, any separation required in $C_{\Gamma''}$ between x_1 or x_2 and some variable $y \in |\Gamma|$ follows from the separation between x and y in $C_{\Gamma'}$. Also, $\mathrm{tens}(x_1)$ or $\mathrm{tens}(x_2)$ in $C_{\Gamma''}$ implies that $\mathrm{tens}(x)$ is in $C_{\Gamma'}$. Finally, if $\mathrm{sep}(x_1, x_2)$ is in $C_{\Gamma''}$ then $i = 1$ which implies that $\mathrm{tens}(x)$ is in $C_{\Gamma'}$ which is sufficient for $\mathrm{sep}(x_1, x_2)$ to hold.

By the correctness of the premise we get $v \Vdash^{\sigma'}_{A'} [\![e]\!]_{\eta[x_1 \mapsto a_1, x_2 \mapsto a_2],[\![P]\!]}$ and $v, \sigma', S'', \sigma \Vdash G_{\Gamma''}$ where $(a_1, a_2) = \eta(x)$. By the definition of denotational semantics we get

$$v \Vdash^{\sigma'}_{A'} [\![\mathsf{match}\ x\ \mathsf{with}\ x_1 \otimes x_2 \Rightarrow e]\!]_{\eta,[\![P]\!]}.$$

It remains to prove $v, \sigma', S, \sigma \Vdash G_{\Gamma'}$. Again, conditions confined to Γ and the result translate from $G_{\Gamma''}$ trivially. If $G_{\Gamma'}$ contains $\mathrm{pres}(x)$ then $G_{\Gamma''}$ has to contain both $\mathrm{pres}(x_1)$ and $\mathrm{pres}(x_2)$ by $i_1, i_2 \geqslant i$. Similarly, we get $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma''|_{1,2}) \subseteq \mathrm{rg}(|\Gamma'|_{1,2})$.

The guarantee $\mathrm{tens}(r)$ in $G_{\Gamma'}$ is equivalent to $\mathrm{tens}(r)$ in $G_{\Gamma''}$. From the rely-precondition of $\mathrm{tens}(r)$ in $G_{\Gamma'}$ we can deduce the same for $G_{\Gamma''}$ analogously to deducing $C_{\Gamma''}$ from $C_{\Gamma'}$ above.

[LIST-ELIM]. A proof of correctness for this rule can be obtained by a straightforward combination and adaptation of the proves for [IF] and [$\otimes$-ELIM] treating $\mathsf{cons}_A(h, t, d)$ as a ternary tensor product. □

Proposition 3.6. *The predefined functions are correctly annotated in Fig. 3.*

Proof. All the functions but cons_A clearly evaluate according to their denotational semantics even without the need to consider the preconditions. $\mathsf{cons}_A(h, t, d)$ also evaluates correctly thanks to the precondition $\mathrm{sep}(\{d\}, \{h, t\})$ (h, t are preserved while overwriting d).

All the functions but cons_A trivially satisfy the guarantee $\mathrm{pres}(|\Gamma|_{2,3})$ since their evaluation does not change the heap at all. $\mathsf{cons}_A(h, t, d)$ overwrites only the location $S(d)$ which is by the precondition separated from $|\Gamma|_{2,3} = \{h, t\}$.

All the functions construct their arguments only using the regions of arguments with usage aspects 1 or 2, hence $\mathrm{rg}(r) \subseteq \mathrm{rg}(|\Gamma|_{1,2})$.

The $\mathrm{tens}(r)$ guarantee is valid trivially for the constants $\mathsf{tt}, \mathsf{ff}, \mathsf{nil}_A$ as they are heap-free. For $\mathsf{fst}, \mathsf{snd}, \_\otimes\_, \mathsf{cons}_A$, it holds $\mathrm{tens}(r)$ if all their arguments with aspect 2 have separated tensor

products and the components of the tensor product and the cons cell are separated which is exactly what the rely-precondition demands. □

4 Other Languages

4.1 Explicit Sharing Programming Language (ESPL)

During the research that led to this report other typings for LFPL extending that of UAPL were designed. Namely, Robert Atkey formulated a typing for LFPL, called ESPL [Atkey 2002*a*], that adds *explicit* information about *sharing* among arguments in addition to the usage aspects.

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

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

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

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

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

$$(1) \qquad y \in S_x \Longrightarrow ((x \in S \Longrightarrow y \in S) \wedge (x \in D \Longrightarrow y \in D))$$

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

Using this intuitive explanation, the annotation can be expressed as the following CG-pair:

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

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

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

which is much simpler and easier to prove correct than UAPL's [LET].

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

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

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

4.2 Language with Deep Sharing (DEEL)

Another first-order LFPL typing that follows the trend is the one of [Konečný 2002] which we shall call DEEL. It is based on a more complex annotation method in which not an argument as a whole is annotated but each argument contains none to many annotations depending on the complexity of its type. Thus assertions can be made about portions of each argument corresponding to certain type subterms within a typing judgement. This allows the type-checking to recognise as correct, among other things, an $append(x, y)$ in which lists x and y share on the level of its deeper data but are separated on the level of the top cons-cells:

$$\begin{aligned}
&x : \mathsf{L}^{[1]}(\mathsf{L}^{[2]}(\mathsf{Bool})), y : \mathsf{L}^{[3]}(\mathsf{L}^{[4]}(\mathsf{Bool}));\\
&\{1{\otimes}2, 1{\otimes}3, 1{\otimes}4\} \vdash\\
&\ append(x, y)\\
&: \mathsf{L}^{[5\subseteq\{1,3\}]}(\mathsf{L}^{[6\subseteq\{2,4\}]}(\mathsf{Bool}));\{1\};\\
&\ 5{\otimes}6 \Longleftarrow \{1{\otimes}2, 1{\otimes}4, 2{\otimes}3, 3{\otimes}4\}\\
&{\otimes}6/5 \Longleftarrow \{2{\otimes}4, {\otimes}2/1, {\otimes}4/3\}
\end{aligned}$$

Each argument has two portions according to their types. The argument portion names are $1, 2, 3, 4$. Just before $\vdash$ there is a pre-condition demanding that three pairs of argument portions be separated. The result type has also two portions, named 5 and 6. 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 in the style of ESPL. The last guarantee is a list of implications giving a set of separation pre-conditions necessary for each basic separation guarantee about the result[1]. The expression ${\otimes}6/5$ indicates internal separation between individual elements of the resulting list which depends on the same property for both argument lists and the separation of 2 from 4.

The language extends UAPL and ESPL in that it type-checks more of the correct LFPL terms. At the same time it has a very simple inference algorithm due to having no annotation-only rules which would make the type-checking non-deterministic (like [DROP] and [RAISE] in

[1] The system is not aware of the fact that 5 cannot share with 6 because a more complex list type might allow sharing between its control structure and elements.

UAPL). This algorithm has been implemented and tested by the author. The above example has been generated by the program too.

5 Conclusion

We have extracted and made explicit the abstract ideas that were behind the original design of UAPL and thus managed to reformulate the usage aspects in a way in which it is manageable to prove its correctness[2] and to work out an annotation inference algorithm. We have also recalled ESPL from this perspective and given some hints for its further study inspired by this view. Furthermore, we previewed the new language DEEL which has been formed as a result of this study.

We conjecture that also $\alpha\lambda$-calculus [O'Hearn 2002] which arises from the logic of Bunched Implications [O'Hearn & Pym 1999] can be interpreted in the present approach similarly to ESPL but without considering preservation guarantees. This is very interesting among other reasons because $\alpha\lambda$-calculus is a higher order language.

The present approach might be beneficial in extending also UAPL to higher order. A suitable extension of the imperative operational semantics with explicit allocation of closures has been suggested in [Atkey 2002*a*] and [Atkey 2002*b*]. The leading idea is that every function type constructor in the judgement has to be treated like a typing judgement (featuring its captured context) and thus should be annotated with a subset of CG-pairs for this context.

For example, one could perhaps abstract away from the captured context and treat it as a single unit with one usage aspect, resulting in having two usage aspects per function type constructor, e.g.:

$$x :^{j} A \ {}_{i}\!\!\rightarrow_{j} A', y :^{i} A \vdash xy : A'$$

This is an area of further research. Nevertheless, it seems that the complex annotation resulting from higher order types will have so intricate dependencies that type-checking will become very hard. For example, the typing judgement above would be valid for any $i, j \in \{1, 2, 3\}$.

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 (TAL). 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. We believe that it is expressive enough to encode the UAPL, ESPL and DEEL assertions but the encoding might not be very natural.

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

[2] Admittedly, the proof is still fairly complex and it might be worth formalising it in a proof assistant to be able to keep careful track of all the details.

cover here.

The novelty of LFPL and its extensions is that one can write in them certified in-place update algorithms that evaluate in accordance with a simple functional denotational semantics. The new resource type $\Diamond$ makes in-place update explicit in a functional setting. In UAPL, ESPL and DEEL any *constructor* argument of type $\Diamond$ is given a "destroyed" aspect. Thus by distinguishing different ways in which the resources are used the typing is made more flexible. UAPL and DEEL moreover keep a deterministic type-checking via usage aspect annotation inference thus liberating a programmer from writing any usage aspects in their program signatures. A complementary aid is given to an LFPL programmer by an automatic inference of constructor arguments of type $\Diamond$ [Jost 2002].

Acknowledgements. This research has been supported by the EPSRC grant GR/N28436/01. The author is grateful to David Aspinall and Robert Atkey for discussion and comments on this work.

References

Aspinall, D. & Hofmann, M. [2002], Another type system for in-place update, *in* D. L. Métayer, ed., 'Programming Languages and Systems, Proceedings of 11th European Symposium on Programming', Springer-Verlag, pp. 36–52. Lecture Notes in Computer Science 2305.

Atkey, R. [2002*a*], LFPL with explicit sharing and destruction. An unpublished draft.

Atkey, R. [2002*b*], Type systems with explicit sharing. Thesis Proposal.

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.

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*

Hofmann, M. [2002], The strength of non size-increasing computation, *in* 'Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science'.

Jost, S. [2002], Static prediction of dynamic space usage of linear functional programs, Master's thesis, Technische Universität Darmstadt, Fachbereich Mathematik.

Konečný, M. [2002], LFPL with types for deep sharing, Technical Report EDI-INF-RR-?, LFCS, Division of Informatics, University of Edinburgh.

O'Hearn, P. & Pym, D. [1999], 'The logic of bunched implications', *Bulletin of Symbolic Logic* **5**(2), 215–243.

O'Hearn, P. W. [2002], On bunched typing. To Appear in the Journal of Functional Programming.

O'Hearn, P. W., Power, A. J., Takeyama, M. & Tennent, R. D. [1999], 'Syntactic control of interference revisited', *Theoretical Computer Science* **228**, 211–252.

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.

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.

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'.

Walker, D. & Morrisett, G. [2000], Alias types, *in* 'ESOP 2000', pp. 366–381. Lecture Notes in Computer Science 1782.

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.