Another type system for in-place update

David Aspinall[1] and Martin Hofmann[2]

[1] LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
da@dcs.ed.ac.uk,
WWW: www.dcs.ed.ac.uk/home/da
[2] Institut für Informatik, Oettingenstraße 67, 80538 München, Germany
mhofmann@informatik.uni-muenchen.de,
WWW: www.tcs.informatik.uni-muenchen.de/ mhofmann

Abstract. Linear typing schemes guarantee single-threadedness and so the soundness of in-place update with respect to a functional semantics. But linear schemes are restrictive in practice, and more restrictive than necessary to guarantee soundness of in-place update. This has prompted research into static analysis and more sophisticated typing disciplines, to determine when in-place update may be safely used, or to combine linear and non-linear schemes. Here we contribute to this line of research by defining a new typing scheme which better approximates the semantic property of soundness of in-place update for a functional semantics. Our typing scheme includes two kinds of products ($\otimes$ and $\times$), which allows data structures with or without sharing to be defined. We begin from the observation that some data is used only in a "read-only" context after which it may be safely re-used before being destroyed. Formalizing the in-place update interpretation and giving a machine model semantics allows us to refine this observation. We define three *usage aspects* apparent from the semantics, which are used to annotate function argument types. The aspects are (1) used destructively, (2) used read-only but shared with result, and (3) used read-only and not shared.

1 Introduction

The distinctive advantage of pure functional programming is that program functions may be viewed as ordinary mathematical functions. Powerful proof principles such as equational reasoning with program terms and mathematical induction are sound, without needing to use stores or other auxiliary entities, as is invariably required when reasoning about imperative programs.

Consider the functional implementation of linked list reversal, as shown in Fig. 1 (for the moment, ignore the first argument to `cons`). This definition of reversal is readily verified by induction and equational reasoning over the set of finite lists. On the other hand, implementing reversal imperatively using pointers is (arguably) more cumbersome and error prone and, more seriously, would be harder to verify using complicated reasoning principles for imperative programs.

The advantage of an imperative implementation, of course, is that it modifies its argument in-place whereas in a traditional functional implementation the result must be created from scratch and garbage collection is necessary in order salvage heap space. We are interested in the possibility of having the best of both worlds by using a semantics-preserving translation of functional programs into imperative ones which use in-place update and need no garbage collection. In previous work by the second author, a first-order functional language called

```
def list reverse_aux (list l,list acc) =
      match l with
              nil -> acc
            | cons(d,h,t) -> reverse_aux(t,cons(d,h,acc))

def list reverse (list l) = reverse_aux(l, nil)

def list append(list l,list m) =
      match l with
              nil -> m
            | cons(d,h,t) -> cons(d,h,append(t,m))
```

Fig. 1. LFPL examples

LFPL was defined, together with such a translation. LFPL relies on some user intervention to manage memory but without compromising the functional semantics in any way. This works by augmenting (non-nil) constructors of inductive datatypes such as `cons` with an additional argument of an abstract "diamond" resource type $\diamond$ whose elements can be thought of as heap-space areas, loosely corresponding to Tofte-Talpin's notion of *regions* [TT97].

To construct an element of an inductive type, we must supply an value of the $\diamond$ abstract type. The only way of obtaining values of type $\diamond$ is by deconstructing elements of recursive types in a pattern match. The first argument to each use of `cons` in Fig. 1 is this value of type $\diamond$; the `cons` on the right hand side of the match is "justified" by the preceding `cons`-pattern. The correspondence need not always be as local; in particular, values of type $\diamond$ may be passed as arguments to and returned by functions, as well as appearing in components of data structures.

We can give a compositional translation of LFPL into C by mapping $\diamond$ to the type `void *` (a universal pointer type), and implementing `cons` as

```
list_t cons(void *d, entry_t hd, list_t tl){
   d->head=hd; d->tail=tl; return d;
}
```

Here, `list_t` is the type of pointers to a C struct with appropriately typed entries `head`, `tail`, and we have elided (compulsory) typecasts to reduce clutter. As expected, `nil` is implemented as a function returning a null pointer. When receiving a non-null argument of type `list_t` we can save its entries as local variables and subsequently use the pointer itself as an argument to `cons()`. This implements pattern matching. For details of the translation, see [Hof00].

The main result of [Hof00] was that the semantics of the C translation agrees with the functional semantics of the soure program *provided* the latter admitted a linear typing for inductive types and $\diamond$ types, i.e., bound variables of inductive type are used at most once. In particular, this linearity guarantees that the memory space pointed to by a $\diamond$-value is not needed anywhere else. This prevents function definitions like:

```
def list twice (list l) =
  match l with
    nil -> nil
  | cons(d,h,t) -> cons(d,0,cons(d,0,twice(l)))
```

The functional semantics of `twice` maps a list l to a list twice as long as l with zero entries; on the other hand, the LFPL translation to C of the above code computes a circular list. As one would expect, the translation of `append` in Fig. 1 appends one linked list to another in place; again, the translation of a non-linear phrase like `append(l,l)` results in a circular list, disagreeing with the functional semantics. As an aside: we can implmenent `twice` in LFPL with the typing `list[int*<>] -> list[int]`, where the argument list provides the right amount of extra space. In recent unpublished work with Steffen Jost and Dilsun Kırlı, we have shown that the position of $\diamond$ types and their arguments can be automatically inferred, using integer linear programming.

Linear typing together with the resource type $\diamond$ seems restrictive at first sight. In particular, without dynamic creation of memory in the translation, no function can be written that increases the size of its input. Yet surprisingly, a great many standard examples from functional programming fit very naturally into the LFPL typing discipline, among them, insertion sort, quick sort, tree sort, breadth-first traversal of a tree and Huffman's algorithm. Moreover, in [Hof00] it was shown that every non size-increasing function on lists over booleans in the complexity class ETIME can be represented.

In spite of this positive outlook, the linear typing discipline, as any other typing scheme, rejects many semantically valid programs. In our context a program is semantically valid if its translation to imperative code computes its functional semantics. We cannot hope to catch all semantically valid programs by a typing discipline, of course, but we can try to refine the type system to reduce the "slack", i.e., the discrepancy between the semantically valid programs and those which pass the typing discipline.

In this paper we address one particular source for slack, namely the implicit assumption that every access to a variable is potentially destructive, i.e., changes the memory region pointed to or affected by this variable. This is overly conservative: multiple uses of a variable need not compromise semantic validity, as long as only the last in a series of multiple uses is destructive (and moreover the results of the earlier accesses do not interfere with the ultimate destructive access). A safe static approximation of this idea in the context of LFPL is the goal of this paper. We present a type system which is more general than linear typing in that it permits multiple uses of variables in certain cases, yet is sound in the sense that for well-typed LFPL programs the imperative translation computes the functional semantics.

1.1 Usage aspects for variables

Some examples will help to motivate our type system. A first example is the function `sumlist : list[int] -> int` which computes the sum of the elements in an integer list.

```
def int sumlist(list[int] l) =
     match l with
             nil -> 0
           | cons(d,h,t) -> h + sumlist(t)
```

With the destructive pattern matching scheme of LFPL, we must consider that `l` is destroyed after evaluating `sumlist(l)`, although the list would actually remain intact under any reasonable implementation. We can avoid losing the list by returning it along with the result, rebuilding it as we compute the sum of the elements. But this leads to a cumbersome programming style, and one has to remember that `sumlist' : list[int] -> int * list[int]` returns its argument unmodified. A better solution is to say that from the definition above, we see that the list is not destroyed (because the $\diamond$-value `d` is not used), so we would like to assign `sumlist` a type which expresses that it does not destroy its argument.

Not only should the `sumlist` function inspect its argument list without modifying it, but the result it returns no longer refers to the list. This means that an expression like

```
cons(d,sumlist(l),reverse(l))
```

where d is of type $\diamond$, should also be soundly implemented, if we assume that evaluation occurs from left to right. In other words, we can freely use the value of `sumlist(l)` even after `l` is destroyed.

This is not the case for functions which inspect data structures without modifying them, but return a result which contains some part of the argument. An example is the function `nth_tail` which returns the nth tail of a list:

```
def list nth_tail(int n, list l) =
  if n<=0 then l else match l with
                          nil -> nil
                        | cons(d,h,t) -> nth_tail(n-1, t)
```

Unlike `sumlist`, the result of `nth_tail` may be *shared* (aliased) with the argument. This means an expression like

```
cons(d,nth_tail(2,l),nil)
```

will be sound, but

```
cons(d,nth_tail(2,l),cons(d',reverse(l),nil))
```

will *not* be soundly implemented by the in-place update version, so the second expression should not be allowed in the language. (If `l=[1,2,3]`, the expression should evaluate to the list `[[3],[3,2,1]]` but the in-place version would yield `[[1],[3,2,1]]`). Simpler example functions in the same category as `nth_tail` include projection functions and the identity function.

As a final example, consider again the `append` function in Fig. 1. The imperative implementation physically appends the second list to the first one and returns the so modified first argument. Thus, the first list `l` has been *destroyed* so we should treat that in the same way as arguments to `reverse`. But the second list `m` is *shared* with the result, and so should be treated in the same way as arguments to `nth_tail`. This suggests that we should consider the way a function operates on each of its arguments.

These observations lead us to distinguish three *usage aspects* of variables, which are the central innovation in our type system. The usage aspects are:

- Aspect 1: modifying use, e.g., `l` in `reverse(l)`
- Aspect 2: non-modifying use, but shared with result, e.g., `m` in `append(l,m)`
- Aspect 3: non-modifying use, not shared with result, e.g., `l` in `sumlist(l)`.

The numbers are in increasing order of "safety" or "permissiveness" in the type system. Variables may only be accessed once with aspect 1. Variables can be used many times with aspect 2, but this prevents an aspect 1 usage later if intermediate results are retained. Finally, variables can be freely used with aspect 3, the pure "read-only" usage. Perhaps surprisingly, these exact distinctions appear to be novel, but they are closely connected to several other analyses appearing in related work [Wad90,Ode92,Kob99] — see Section 5 for precise comparisons.

Our type system decorates function arguments with usage aspects, and then tracks the way that variables are used. For example, we have the following types:

```
  reverse : list[t]^1 -> list[t]
  sumlist : list[t]^3 -> int
 nth_tail : list[t]^2 * int^3 -> list[t]
   append : list[t]^1 * list[t]^2 -> list[t]
```

Heap-free types such as *int* will always have the read-only aspect 3. Functions which have a heap-free result (like `sumlist`) may have aspect 3 for their non heap-free arguments, provided they are not modified when computing the result.

1.2 Sharing in data structures

The strict linear type system in LFPL prevents sharing in data structures, which can lead to bad space behaviour in some programs. The append function shows how we might be able to allow some limited sharing within data structures but still use an in-place update implementation, provided we take care over when modification is allowed. For example, we would like to allow the expression

```
let x=append(u,w) and y=append(v,w) in e
```

provided that we don't modify both `x` and `y` in `e`; after either has been modified we should not refer to the other. Similarly, we would like to allow a tree which has sharing amongst subtrees, in the simplest case a node constructed like this:

```
let u=node(d,a,t,t) in e
```

(where `d:<>` and `a` is a label). This data structure should be safe so long as we do not modify both branches of u. The kinds of data structure we are considering here have a DAG-like layout in memory.

The "not modifying both parts" flavour of these examples leads us to include two kinds of products in our language. Consider binary trees. In a linear setting we have two kinds of trees, one corresponding to trees laid out in full in memory ($\otimes$-trees), the other corresponding more to an object-oriented representation ($\times$-trees) under which a tree can be sent messages asking it to return the outermost constructor or to evolve into one of its subtrees. In ordinary functional programming these two are extensionally equivalent; in the presence of linearity constraints they differ considerably. The $\otimes$-trees allow simultaneous access to all their components thus encompassing e.g., computing the list of leaf labellings, whereas access to the $\times$-trees is restricted to essentially search operations. Conversely, $\otimes$-trees are more difficult to construct; we must ensure that their overall size is polynomially bounded which precludes in particular the definition of a function which constructs the full binary tree of *depth* n. On the other hand, the typing rules would allow construction of a full binary $\times$-tree, which is represented as a rather small DAG. The novelty here is that we can reflect in the type system the kind of choices that a programmer would normally make in selecting the best data representation for a purpose.

The product already in LFPL as studied to date is the tensor product (denoted by $\otimes$, resp. `*` in code), accessed using a pattern matching construct:

```
match p with x*y -> e
```

This allows both `x` and `y` to be accessed simultaneously in e. (Typing rules are shown in the next section). Given a $\otimes$-product of two lists, we can access (maybe modify) both components; to be sound, the two lists must have no sharing.

The cartesian product (denoted $\times$, resp. `X` in code) which corresponds to the & connective of linear logic has a different behaviour. We may access one component or the other, but not both; this means that the two components may have sharing. With our usage aspects, we can be more permissive than allowing just access to one component of the product. We can safely allow access to both components, so long as at most one component is modified, and if it is, the other one is not referenced thereafter. The pairing rule for cartesian products has a special side condition which allows this. Cartesian products are accessed via projection functions:

```
fst : (t X u)^2 -> t
snd : (t X u)^2 -> u
```

The usage aspect 2 here indicates that the result shares with the argument, which is the other part of enforcing the desired behaviour.

To allow data structures with sharing, we can give constructors arguments of cartesian product types. Ideally, we would allow the user to choose exactly where cartesian products are used and where $\otimes$-products are used, to allow the user to define datatypes appropriate for their application. For the purpose of exposition in this paper, however, we will treat both lists and tree types as primitives, and consider just the $\otimes$-product style data structures as used in LFPL.

O'Hearn and Pym's "bunched implications" [OP99] are also based on $\otimes$ and $\times$ coexisting. In our case, $\times$ is not a true categorical product since the $\times$-pairing operation $\langle -, - \rangle$ is partial; it was shown in [Hof99] that implementing $\langle e_1, e_2 \rangle$ as a closure ($\lambda t.\mathsf{if}\ t\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$) recovers the categorical product, but it requires heap space, violating heap size boundedness.

2 Syntax and typing

Syntax. The grammar for the types and terms of our improved LFPL is given in Fig. 2. For brevity, we use N also for the type of booleans (as in C). Types not containing diamonds $\diamond$, lists $\mathsf{L}(-)$ or trees $\mathsf{T}(-)$ are called *heap-free*, e.g. N and $\mathsf{N} \otimes \mathsf{N}$ are heap-free. We use x and variants to range over (a set of) variables and f to range over function symbols.

$$
\begin{array}{rl}
A ::= & \mathsf{N} \mid \diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A_1 \otimes A_2 \mid A_1 \times A_2 \\
e ::= & c \mid f(x_1, \ldots, x_n) \mid x \mid \mathsf{let}\ x = e_x\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2 \\
\mid & e_1 \otimes e_2 \mid \mathsf{match}\ x\ \mathsf{with}\ (x_1 \otimes x_2) \Rightarrow e \mid (e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
\mid & \mathsf{nil} \mid \mathsf{cons}(x_d, x_h, x_t) \mid \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_n | \mathsf{cons}(x_d, x_h, x_t) \Rightarrow e_c \\
\mid & \mathsf{leaf}(x_d, x_a) \mid \mathsf{node}(x_d, x_a, x_l, x_r) \\
\mid & \mathsf{match}\ x\ \mathsf{with}\ \mathsf{leaf}(x_d, x_a) \Rightarrow e_l | \mathsf{node}(x_d, x_a, x_l, x_r) \Rightarrow e_n
\end{array}
$$

Fig. 2. LFPL grammar

To simplify the presentation, we restrict the syntax so that most term formers can only be applied to variables. In practice, we can define the more general forms such as $f(e_1, \ldots, e_n)$ easily as syntactic sugar for nested let-expressions. Also, compared with [Hof00] we will use a different translation scheme, where every non-nullary constructor of inductive type takes exactly one $\diamond$-argument, rather than a $\diamond$-argument for every ordinary argument of inductive type. This is closer to the Java compilation described in [AH01].

A program consists of a series of (possibly mutually recursive) function definitions of the form $f(x_1, \ldots, x_n) = e_f$. These definitions must be well-typed. To help ensure this, a program is given together with a signature Σ, which is a finite function from *function symbols* to first-order function types with usage aspects, i.e. of the form $A_1^{i_1}, \ldots, A_n^{i_n} \to A$. In the typing rules we will assume a fixed program with signature Σ.

We keep track of *usage aspects* for variables as introduced above. We write $x \overset{i}{:} A$ to mean that $x{:}A$ will be used with aspect $i \in \{1, 2, 3\}$ in the subject of the typing judgement. A *typing context* Γ is a finite function from identifiers to types A with usage aspects. If $x \overset{i}{:} A \in \Gamma$ we write $\Gamma(x) = A$ and $\Gamma[x] = i$.

We use familiar notation for extending contexts. If $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x \overset{i}{:} A$ for the extension of Γ with $x \overset{i}{:} A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$

then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise or conclusion of a rule below it is implicitly understood that these disjointness conditions are met. We write $e[x/y]$ for the term obtained from e by replacing all occurrences of the free variable y in e by x. We consider terms modulo renaming of bound variables.

In a couple of the typing rules we need some additional notation for manipulating usage aspects on variables. The "committed to i" context Δ^i is the same as Δ, but each declaration $x \overset{2}{:} A$ of an aspect 2 (aliased) variable is replaced with $x \overset{i}{:} A$. If we have two contexts Δ_1, Δ_2 which only differ on usage aspects, so $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$ and $\Delta_1(x) = \Delta_2(x)$ for all x, then we define the merged context $\Gamma = \Delta_1 \wedge \Delta_2$ by $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta_1)$, $\Gamma(x) = \Delta_1(x)$, $\Gamma[x] = \min(\Delta_1(x), \Delta_2(x))$. The merged context takes the "worst" usage aspect of each variable.

Signatures. We treat constructors as function symbols declared in the signature. We also include primitive arithmetic and comparison operations in the signature. Specifically, we can assume Σ contains a number of declarations:

$$\begin{array}{ll} +, -, <, > & : \mathsf{N}^3, \mathsf{N}^3 \to \mathsf{N} \\ \mathsf{nil}_A & : \mathsf{L}(A) \\ \mathsf{cons}_A & : \Diamond^1, A^2, \mathsf{L}(A)^2 \to \mathsf{L}(A) \\ \mathsf{leaf}_A & : \Diamond^1, A^2 \to \mathsf{T}(A) \\ \mathsf{node}_A & : \Diamond^1, A^2, \mathsf{T}(A)^2, \mathsf{T}(A)^2 \to \mathsf{T}(A) \\ \mathsf{fst}_{A\times B} & : (A \times B)^2 \to A \\ \mathsf{snd}_{A\times B} & : (A \times B)^2 \to B \end{array}$$

for suitable types A as used in the program. The comma between argument types is treated as a $\otimes$-product, which means that these typings, and the corresponding elimination rules below, describe lists and trees with simultaneous access to subcomponents. Hence they must be implemented without sharing unless the access is guaranteed to be read-only. (For trees $\mathsf{ST}(A)$ with unrestricted sharing between components we could use the typing:

$$\mathsf{sharednode}_A : \Diamond^1, (A \times \mathsf{ST}(A) \times \mathsf{ST}(A))^2 \to \mathsf{ST}(A).$$

In this typing, there can be sharing amongst the label and subtrees, but still no sharing with the $\Diamond$ argument, of course, since the region pointed to by the $\Diamond$-argument is overwritten to store the constructed cell.)

Typing rules. Now we explain the typing rules, shown in Fig. 2, which define a judgement of the form $\Gamma \vdash e : A$. Most rules are straightforward. We use an affine linear system, so include WEAK. In VAR, variables are given the default aspect 2, to indicate sharing. If the result is a value of heap-free type, then with RAISE we can promote variables of aspect 2 to aspect 3 to reflect that they do not share with the result. The rule DROP goes the other way and allows us to assume that a variable is used in a more destructive fashion than it actually is.

$$\frac{}{\vdash c : \mathsf{N}}\ (\textsc{const}) \qquad \frac{}{x \overset{2}{:} A \vdash x : A}\ (\textsc{var}) \qquad \frac{\Gamma \vdash e : A}{\Gamma, \Delta \vdash e : A}\ (\textsc{weak})$$

$$\frac{\Gamma \vdash e : A \qquad A \text{ heap-free}}{\Gamma^3 \vdash e : A}\ (\textsc{raise}) \qquad \frac{\Gamma, x \overset{i}{:} A \vdash e : B \qquad j \le i}{\Gamma, x \overset{j}{:} A \vdash e : B}\ (\textsc{drop})$$

$$\frac{f : A^{i_1}, \ldots, A^{i_n} \to B \ \text{ in } \Sigma}{x_1 \overset{i_1}{:} A_1, \ldots, x_n \overset{i_n}{:} A_n \vdash f(x_1, \ldots, x_n) : B}\ (\textsc{fun})$$

$$\frac{\Gamma \vdash e_1 : C \qquad \Gamma \vdash e_2 : C}{\Gamma, x \overset{3}{:} \mathsf{N} \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : C}\ (\textsc{if})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \quad \Delta_2, \Theta, x \overset{i}{:} A \vdash e_2 : B \quad \begin{array}{l}\text{Either } \forall z.\Delta_1[z] = 3, \\ \text{or } i = 3, \forall z.\Delta_1[z] \ge 2, \Delta_2[z] \ge 2\end{array}}{\Gamma^i, \Theta, \Delta_1^i \wedge \Delta_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B}\ (\textsc{let})$$

$$\frac{}{x_1 \overset{2}{:} A_1, x_2 \overset{2}{:} A_2 \vdash x_1 \otimes x_2 : A_1 \otimes A_2}\ (\otimes\textsc{-pair})$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \qquad \Theta, \Delta_2 \vdash e_2 : A_2 \qquad \text{condition } \star}{\Gamma, \Theta, \Delta_1 \wedge \Delta_2 \vdash (e_1, e_2) : A_1 \times A_2}\ (\times\textsc{-pair})$$

$$\frac{\Gamma, x_1 \overset{i_1}{:} A_1, x_2 \overset{i_2}{:} A_2 \vdash e : B \qquad i = \min(i_1, i_2)}{\Gamma, x \overset{i}{:} A_1 \otimes A_2 \vdash \mathsf{match}\ x\ \mathsf{with}\ (x_1 \otimes x_2) \Rightarrow e : B}\ (\textsc{pair-elim})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_{\mathsf{nil}} : B \\ \Gamma, x_d \overset{i_d}{:} \diamond, x_h \overset{i_h}{:} A, x_t \overset{i_t}{:} \mathsf{L}(A) \vdash e_{\mathsf{cons}} : B \qquad i = \min(i_d, i_h, i_t)\end{array}}{\Gamma, x \overset{i}{:} \mathsf{L}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(x_d, x_h, x_t) \Rightarrow e_{\mathsf{cons}} : B}\ (\textsc{list-elim})$$

$$\frac{\begin{array}{l}\Gamma, x_d \overset{i_d}{:} \diamond, x_a \overset{i_a}{:} A \vdash e_{\mathsf{leaf}} : B \\ \Gamma, x_d \overset{i_d}{:} \diamond, x_a \overset{i_a}{:} A, x_l \overset{i_l}{:} \mathsf{T}(A), x_r \overset{i_r}{:} \mathsf{T}(A) \vdash e_{\mathsf{node}} : B \qquad i = \min(i_a, i_d, i_l, i_r)\end{array}}{\Gamma, x \overset{i}{:} \mathsf{T}(A) \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{leaf}(x_d, x_a) \Rightarrow e_{\mathsf{leaf}} | \mathsf{node}(x_d, x_a, x_l, x_r) \Rightarrow e_{\mathsf{node}} : B}\ (\textsc{tree-elim})$$

Fig. 3. Typing rules

The LET rule is somewhat intricate. The context is split into three pieces: variables specific to the definition e_1, in Γ; variables specific to the body e_2, in Θ; and common variables, in Δ_1 and Δ_2, which may be used with different aspects in e_1 and e_2. First, we type-check the definition to find its type A. Then we type-check the body using some usage aspect i for the bound variable x. The way the bound variable x is used in the body is used to commit any aliased variables belonging to e_1. For example, if x is used destructively in e_2, then all aliased variables in Γ and Δ_1 are used destructively in the overall expression; this accounts for the use of Γ^i and Δ^i in the conclusion. The aspects in Δ_1 and Δ_2 are merged in the overall expression, taking into account the way that x is used in e_2. The side condition prevents any common variable z being modified in e_1 or e_2 before it is referenced in e_2. More exactly, $\Delta_1[z] = 1$ is not allowed (the value of the variable would be destroyed in the binding); $\Delta_1[z] = 3$ is always allowed (the value of the variable has no heap overlap with the binding value), and $\Delta_1[z] = 2$ is allowed provided neither $i = 1$ nor $\Delta_2[z] = 1$ (the value of the common variable may have aliasing with e_2, provided it is not partly or completely destroyed in e_2: the modification may happen before the reference). As an instance of LET, we get a derived rule of contraction for aspect 3 variables.

The only constructor rules we need are for the two kinds of pairs. The rule for constructing a $\times$-pair ensures that all variables which are shared between the components have aspect at least 2. The "condition $\star$" in rule $\times$-PAIR is:

- $\Delta_1[z] \geq 2$ and $\Delta_2[z] \geq 2$ for all $z \in \mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$.

which ensures that no part of memory shared between the components is destroyed when the pair is constructed. (A more liberal condition which considers evaluation order is possible, but we omit it here.)

In the destructor rules we type-check the branches in possibly extended contexts, and then pass the worst-case usage aspect as the usage for the term being destructed. For example, if we destroy one half of a pair in PAIR-ELIM, so x_1 has usage aspect 1, then the whole pair is considered destroyed in the conclusion.

3 Imperative operational semantics

To establish the correctness of our typing rules, we need to formalize the intended in-place update interpretation of the language. In [Hof00], a translation to 'C' and a semantics for the target sublanguage for 'C' was used. Here we instead use an abstract machine model; this allows us to more easily consider alternative translations to other languages, such as the Java translation given in [AH01], or a typed assembly language interpretation, as given in [AC02]. The interpretation we consider here is closest to the Java translation from [AH01].

Let Loc be a set of *locations* which model memory addresses on a heap. We use l to range over elements of Loc. Next we define two sets of values, *stack values* SVal, ranged over by v, and *heap values* HVal, ranged over by h, thus:

$$
\begin{aligned}
v &::= n \quad | \quad l \quad | \quad \mathsf{NULL} \quad | \quad (v, v) \\
h &::= v \quad | \quad \{\mathsf{f_1} = v_1 \ldots \mathsf{f_n} = v_n\}
\end{aligned}
$$

A stack value is either an integer n, a location l, a null value NULL, or a pair of stack values (v, v). A heap value is either a stack value or an n-ary record consisting of named fields with stack values. A stack $S{:}\mathsf{Var} \rightharpoonup \mathsf{SVal}$ is a partial mapping from variables to stack values, and a heap $\sigma{:}\mathsf{Loc} \rightharpoonup \mathsf{HVal}$ is a partial mapping from locations to heap values. Evaluation of an expression e takes place with a given stack and heap, and yields a stack value and a possibly updated heap. Thus we have a relation of the form $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ expressing that the evaluation of e under stack S and heap σ terminates and results in stack value v. As a side effect the heap is modified to σ'.

The only interesting cases in the operational semantics are the ones for the heap datatypes, which make use of $\diamond$-values as heap locations. For example, the following rules for cons:

$$\frac{S, \sigma \vdash e_d \rightsquigarrow l_d, \sigma' \qquad S, \sigma' \vdash e_h \rightsquigarrow v_h, \sigma'' \qquad S, \sigma'' \vdash e_t \rightsquigarrow v_t, \sigma'''}{S, \sigma \vdash \mathsf{cons}(e_d, e_h, e_t) \rightsquigarrow l_d, \sigma'''[l_d \mapsto \{\mathsf{hd}{=}v_h, \mathsf{tl} = v_t\}]}$$

$$\frac{\begin{array}{c} S, \sigma \vdash e \rightsquigarrow l, \sigma' \qquad \sigma'(l) = \{\mathsf{hd}{=}v_h, \mathsf{tl}{=}v_t\} \\ S[x_d \mapsto l, x_h \mapsto v_h, x_t \mapsto v_t], \sigma' \vdash e_{\mathsf{cons}} \rightsquigarrow v, \sigma'' \end{array}}{S, \sigma \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} {\Rightarrow} e_{\mathsf{nil}} | \mathsf{cons}(x_d, x_h, x_t) {\Rightarrow} e_{\mathsf{cons}} \rightsquigarrow v, \sigma''}$$

In the constructor rule, the first argument e_d of cons is a term of $\diamond$ type, which we evaluate to a heap location l_d. We then evaluate the head and the tail of the list in turn, propagating any changes to the heap. Finally, the result is the location l_d where we make the cons cell by updating the heap, using a record with hd and tl fields. The match rule performs the opposite operation, breaking apart a cons-cell.

This operational semantics describes the essence of our in-place update interpretation of the functional language, without considering more complex translations or optimizations that might be present in a real compiler.

4 Correctness

In this section we will prove that for a typable program, the imperative operational semantics is sound with respect to a functional (set-theoretic) semantics.

Set-theoretic interpretation. We define the set-theoretic interpretation of types $[\![A]\!]$, by setting $[\![\mathsf{N}]\!] = \mathbf{Z}$, $[\![\diamond]\!] = \{0\}$, $[\![\mathsf{L}(A)]\!] =$ finite lists over $[\![A]\!]$, $[\![\mathsf{T}(A)]\!] =$ binary $[\![A]\!]$-labelled trees, and $[\![A \otimes B]\!] = [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$. To each program $(\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ we can now associate a mapping ρ such that $\rho(f)$ is a *partial* function from $[\![A_1]\!] \times \ldots [\![A_n]\!]$ to $[\![B]\!]$ for each $f : A_1^{i_1}, \ldots, A_n^{i_n} \to B$. This meaning is given in the standard fashion as the least fixpoint of an appropriate compositionally defined operator, as follows. A *valuation* of a context Γ is a function η such that $\eta(x) \in [\![\Gamma(x)]\!]$ for each $x \in \mathrm{dom}(\Gamma)$; a valuation of a signature Σ is a function ρ such that $\rho(f) \in [\![\Sigma(f)]\!]$ whenever $f \in \mathrm{dom}(\Sigma)$. To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ in the obvious way: function symbols and variables are interpreted according to the valuations;

basic functions and expression formers are interpreted by the eponymous set-theoretic operations, ignoring the $\diamond$-type arguments in the case of constructor functions. The formal definition of $[\![-]\!]_{\eta,\rho}$ is by induction on terms. A *program* $(\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$ is then the least valuation ρ such that

$$\rho(f)(v_1,\ldots,v_n) = [\![e_f]\!]_{\rho,\eta}$$

where $\eta(x_i) = v_i$, for any $f \in \mathrm{dom}(\Sigma)$.

Notice that this set-theoretic semantics does not say anything about space usage and treats $\diamond$ as a single-point type; its only purpose is to pin down the functional denotations of programs so that we can formally state a correctness result for the in-place update operational interpretation.

Heap regions. Given a stack value v, a type A and a heap σ we define the associated *region* $R_A(v,\sigma)$ as the least set of locations satisfying

- $R_{\mathsf{N}}(n,\sigma) = \emptyset$,
- $R_{\diamond}(l,\sigma) = \{l\}$,
- $R_{A\times B}((v_1,v_2),\sigma) = R_{A\otimes B}((v_1,v_2),\sigma) = R_A(v_1,\sigma) \cup R_B(v_2,\sigma)$,
- $R_{\mathsf{L}(A)}(\mathsf{NULL},\sigma) = \emptyset$,
- $R_{\mathsf{L}(A)}(l,\sigma) = \{l\} \cup R_A(h,\sigma) \cup R_{\mathsf{L}(A)}(t,\sigma)$ when $\sigma(l) = \{\mathsf{hd} = h, \mathsf{tl} = t\}$ (otherwise $\emptyset$),
- $R_{\mathsf{T}(A)}(l,\sigma) = \{l\} \cup R_A(v,\sigma)$, when $\sigma(l) = \{\mathsf{label} = v\}$,
- $R_{\mathsf{T}(A)}(l,\sigma) = \{l\} \cup R_A(v,\sigma) \cup R_{\mathsf{T}(A)}(t_l,\sigma) \cup R_{\mathsf{T}(A)}(t_r,\sigma)$, when $\sigma(l) = \{\mathsf{label} = v, \mathsf{left} = t_l, \mathsf{right} = t_r\}$ (otherwise $\emptyset$).

It should be clear that $R_A(v,\sigma)$ is the part of the (domain of) σ that is relevant for v. Accordingly, if $\sigma(l) = \sigma'(l)$ for all $l \in R_A(v,\sigma)$ then $R_A(v,\sigma) = R_A(v,\sigma')$. If A is a heap-free type, then $R_A(v,\sigma) = \emptyset$.

Meaningful stack values in a heap. Next, we need to single out the meaningful stack values and relate them to the corresponding semantic values. A stack value is *meaningful* for a particular type and heap if it has a sensible interpretation in the heap for that type. For instance, if $\sigma(a) = \{\mathsf{hd} = 1, \mathsf{tl} = \mathsf{NULL}\}$ then a would be a meaningful stack value of type $\mathsf{L}(\mathsf{N})$ with respect to σ and it would correspond to the semantic list $[1]$. Again, w.r.t. that same heap (a,a) would be a meaningful stack value of type $\mathsf{L}(A) \times \mathsf{L}(A)$ corresponding to the semantic pair $([1],[1]) \in [\![\mathsf{L}(A) \times \mathsf{L}(A)]\!]$. That same value (a,a) will also be a meaningful stack value of type $\mathsf{L}(A) \otimes \mathsf{L}(A)$ in case it will be used in a read only fashion. This occurs for example in the term $f(x \otimes x)$ when $f : (A \otimes A)^3 \to B$. This means that "meaningfulness" is parametrised by the aspect with which the value is going to be used. No distinction is made, however, between aspects 2 and 3 in this case.

Given a stack value v, a type A, a heap σ, a denotation $a \in [\![A]\!]$ and an aspect $i \in \{1,2,3\}$, we define a five-place relation $v \Vdash^{\sigma}_{A,i} a$ which expresses that v is a meaningful stack value of type A with respect to heap σ corresponding to semantic value a in aspect i. It is defined inductively as follows:

- $n \Vdash^{\sigma}_{\mathsf{N},i} n'$, if $n = n'$.

- $l \Vdash^{\sigma}_{\diamond,i} 0$.
- $(v_1, v_2) \Vdash^{\sigma}_{A_1 \times A_2,i} (a_1, a_2)$ if $v_k \Vdash^{\sigma}_{A_k,i} a_k$ for $k = 1, 2$.
- $(v_1, v_2) \Vdash^{\sigma}_{A_1 \otimes A_2,i} (a_1, a_2)$ if $v_k \Vdash^{\sigma}_{A_k,i} a_k$ for $k = 1, 2$. Additionally, $R_{A_1}(v_1, \sigma) \cap R_{A_2}(v_2, \sigma) = \emptyset$ in case $i = 1$.
- $\mathsf{NULL} \Vdash^{\sigma}_{\mathsf{L}(A),i} \mathsf{nil}$.
- $l \Vdash^{\sigma}_{\mathsf{L}(A),i} \mathsf{cons}(h, t)$, if $\sigma(l) = \{\mathsf{hd}{=}v_h, \mathsf{tl} = v_t\}$, $l \Vdash^{\sigma}_{\diamond,i} 0$ and $v_h \Vdash^{\sigma}_{A,i} h$ and $v_t \Vdash^{\sigma}_{\mathsf{L}(A),i} t$. Additionally, $R_\diamond(l, \sigma)$, $R_A(v_h, \sigma)$, $R_{\mathsf{L}(A)}(v_t, \sigma)$ are pairwise disjoint in case $i = 1$.
- $l \Vdash^{\sigma}_{\mathsf{T}(A),i} \mathsf{leaf}(a)$ if $\sigma(l) = \{\mathsf{label}{=}v_a\}$ and $l \Vdash^{\sigma}_{\diamond,i} 0$ and $v_a \Vdash^{\sigma}_{A} a$. Additionally, $R_\diamond(l, \sigma) \cap R_A(v_a) = \emptyset$ in case $i = 1$.
- $l \Vdash^{\sigma}_{\mathsf{T}(A),i} \mathsf{node}(a, l, r)$ if $\sigma(l) = \{\mathsf{label}{=}v_a, \mathsf{left}{=}v_l, \mathsf{right}{=}v_r\}$ and $l \Vdash^{\sigma}_{\diamond,i} 0$ and $v_a \Vdash^{\sigma}_{A,i} a$ and $v_l \Vdash^{\sigma}_{A,i} l$ and $v_r \Vdash^{\sigma}_{A,i} r$. Additionally $R_\diamond(l, \sigma)$, $R_A(v_a, \sigma)$, $R_{\mathsf{T}(A)}(v_l, \sigma)$, $R_{\mathsf{T}(A)}(v_r, \sigma)$ are pairwise disjoint in case $i = 1$.

Notice that $\Vdash^{\sigma}_{A,2}$ and $\Vdash^{\sigma}_{A,3}$ are identical, whereas $\Vdash^{\sigma}_{A,1}$ prevents any "internal sharing" within $\otimes$-product types in the heap representation. We extend this relation to stacks and valuations for a context, by defining $S \Vdash^{\sigma}_{\Gamma} \eta$ thus:

- $S(x) \Vdash^{\sigma}_{\Gamma(x),\Gamma[x]} \eta(x)$ for each $x \in \mathrm{dom}(\Gamma)$
- $x \neq y$ and $R_{\Gamma,x}(S, \sigma) \cap R_{\Gamma,y}(S, \sigma) \neq \emptyset$ implies $\Gamma[x] \geq 2, \Gamma[y] \geq 2$.

where as a shorthand, $R_{\Gamma,x}(S, \sigma) =_{\mathrm{def}} R_{\Gamma(x)}(S(x), \sigma)$. So $S \Vdash^{\sigma}_{\Gamma} \eta$ holds if stack S and heap σ are meaningful for the valuation η at appropriate types and aspects, and moreover, the region for each aspect 1 variable does not overlap with the region for any other variable. (Informally: the aspect 1 variables are safe to update.) Below we use the shorthand $R_\Gamma(S, \sigma) =_{\mathrm{def}} \bigcup_{x \in \mathrm{dom}(\Gamma)} R_{\Gamma,x}(S, \sigma)$.

Correctness theorem. With this definition of meaningfulness in place, we can prove that the evaluation of a term under a meaningful stack and heap gives a meaningful result corresponding to the set-theoretic semantics. As usual, we prove a stronger statement to get an inductive argument through.

Theorem 1. *Assume the following data and conditions:*

1. *a program P over some signature Σ with meaning ρ,*
2. *a well-typed term $\Gamma \vdash e : C$ over Σ for some Γ, e, C,*
3. *a heap σ, a stack S and a valuation η, such that $S \Vdash^{\sigma}_{\Gamma} \eta$*

Then $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ for some (uniquely determined) v, σ' if and only if $[\![e]\!]_{\eta,\rho}$ is defined. Moreover, in this case the following hold:

1. *$R_C(v, \sigma') \subseteq R_\Gamma(S, \sigma)$,*
2. *if $l \notin R_\Gamma(S, \sigma)$ then $\sigma(l) = \sigma'(l)$,*
3. *if $l \in R_{\Gamma,x}(S, \sigma)$ and $\Gamma[x] \geq 2$ then $\sigma(l) = \sigma'(l)$,*
4. *$v \Vdash^{\sigma'}_{C,2} [\![e]\!]_{\eta,\rho}$,*
5. *$S \Vdash^{\sigma}_{\Gamma^1} \eta$ implies $v \Vdash^{\sigma'}_{C,1} [\![e]\!]_{\eta,\rho}$ and $R_C(v, \sigma') \cap R_{\Gamma,x}(S, \sigma) = \emptyset$ when $\Gamma[x] = 3$.*

This theorem expresses both the meaningfulness of the representation of semantic values on the machine, and the correctness of the operational semantics with respect to the set-theoretic semantics. The five consequences capture the expected behaviour of the imperative operational semantics and the variable aspects. In brief: (1) no new memory is consumed; (2) heap locations outside those reachable from input variables are unchanged in the result heap; (3) in-place updates are only allowed for locations in the regions of aspect 1 variables; (4) the operational semantics agrees with the set-theoretic result for aspects 2 and 3; (5) if the heap additionally has no variable-variable overlaps or "internal" sharing for aspect 2 variables, then the meaningfulness relation also holds in aspect 1 (in particular, there is no internal sharing within the result value v), and moreover, there is no overlap between the result region and the region of any aspect 3 variable. This means it is safe to use the result in an updating context.

Specialising this perhaps daunting theorem to the particular case of a unary function on lists yields the following representative corollary:

Corollary 1. *Let P be a program having a function symbol $f : \mathsf{L}(\mathsf{N})^i \to \mathsf{L}(\mathsf{N})$.*

If σ is a store and l is a location such that l points in σ to a linked list with integer entries $w = [x_1, \ldots, x_n]$ in σ then $\rho(f)(w)$ is defined iff $[x \mapsto l], \sigma \vdash f(x) \leadsto v, \sigma'$ for some v, σ' and in this case v points in σ' to a linked list with integer entries $\rho(f)(w)$.

Additionally, one can draw conclusions about the heap region of the result list depending on the value of i.

In further work (partly underway) we are examining the two kinds of product in more detail, and considering array types with and without sharing between entries. We are also looking at dynamically allocated store via built-in functions $\mathsf{new} :\to \Diamond$ and $\mathsf{dispose} : \Diamond^1 \to \mathsf{N}$. These built-ins can be implemented by interfacing to an external memory manager, for example, using `malloc` and `free` system calls augmented with a free list. This allows more functions to be defined but breaks the heap-bounded nature of the system, in general.

5 Conclusions and related work

We defined an improved version of the resource-aware linear programming language LFPL, which includes *usage aspect* annotation on types. We also added datatypes based on cartesian products, to allow sharing in data structures on the heap. Using an operational semantics to formalize the in-place update interpretation, we proved that evaluation in the language is both type-sound for a memory model and correct for a set-theoretic functional semantics.

The philosophy behind LFPL is that of providing a *static guarantee* that efficient in-place implementations are used, while allowing as many programs as possible. The guarantee is enforced by the type system. This is in contrast to various other proposed systems which perform static analysis during compilation, or mix linear and non-linear typing schemes, to achieve compilations which are often efficient in practice, but which provide no absolute guarantee. In its pure form, LFPL does not include any instructions for allocating heap space, so all computation is done in-place, using constant heap space. This guarantee is provided for any program which can be written in the language. Apart from differences in philosophy, our usage aspects and their semantic motivation from the memory model are somewhat novel compared with previous work. Related ideas and annotations do already appear in the literature, although not always with semantic soundness results. We believe that our system is simpler than much of the existing work, and in particular, the use of the resource type $\diamond$ is crucial: it is the appearance of $\diamond^1$ in the typing of constructors like cons that expresses that constructors are given an in-place update interpretation. Without the resource type $\diamond$ there would be no aspect 1 component.

Here is a necessarily brief comparison with some of the previous work. The closest strand of work begins with Wadler's introduction of the idea of a sequential let [Wad90]. If we assume that e_1 is evaluated before e_2 in the expression

$$\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2[x]$$

then we can allow sharing of a variable z between e_1 and e_2, as long as z is not modified in e_1 and some other side-conditions which prevent examples like

$$\mathsf{let}\ x{=}y\ \mathsf{in}\ \mathsf{append}(x, y).$$

Our rule for let follows similar ideas. Odersky [Ode92] built on Wadler's idea of the sequential let. He has an *observer* annotation, which corresponds to our aspect 2 annotations: not modified, but still occurs in the result. He too has side conditions for the let rule which ensure soundness, but there is no proof of this (Odersky's main result is a type reconstruction algorithm). Kobayashi [Kob99] introduces *quasi-linear* types. This typing scheme also allows sharing in let expressions. It has a *δ-usage* which corresponds roughly to our aspect 3 usage. Kobayashi's motivation was to statically detect points where deallocation occurs; this requires stack-managed extra heap, augmenting region analysis [TT97]. Kobayashi also allows non-linear use of variables (we might similarly add an extra aspect to LFPL to allow non-linear variables, if we accepted the use of a garbage collector). Kobayashi proves a traditional type soundness (subject reduction) property, which shows an internal consistency of his system, whereas we have characterised and proved equivalence with an independently meaningful semantic property. It might well be possible to prove similar results to ours for Kobayashi's system, but we believe that by considering the semantical property at the outset, we have introduced a rather more natural syntactic system, with simpler types and typing rules.

There is much other related work on formal systems for reasoning or type-checking in the presence of aliasing, including for example work by Reynolds, O'Hearn and others [Rey78,OTPT95,Rey00,IO01]; work on the imperative λ-calculus [YR97]; uniqueness types [BS96], usage types for optimised compilation of lazy functional programs [PJW00] and program analyses for destructive array updates [DP93,WC98] as automated in PVS [Sha99]. There is also related work in the area of compiler construction and typed assembly languages, where researchers have investigated static analysis techniques for determining when optimisations such as in-place update or compile-time garbage collection are admissible; recent examples include shape analysis [WSR00], alias types [SWM00], and static capabilities [CWM99], which are an alternative and more permissive form of region-based memory management. One of our future goals is to relate our work back to research on compiler optimizations and typed low-level languages, in the hope that we can *guarantee* that certain optimizations will always be possible in LFPL, by virtue of its type system. This is in contrast to the behaviour of many present optimizing compilers, where it is often difficult for the programmer to be sure if a certain desirable optimization will performed by the compiler or not. Work in this directions has begun in [AC02], where a typed assembly language is developed which has high-level types designed to support compilation from LFPL, to obviate the need for garbage collection.

We see the work reported here as a step along the way towards a powerful high-level language equipped with notions of resource control. There are more steps to take. We want to consider richer type systems closer to those used in present functional programming languages, in particular, including polymorphic and higher-order types. For the latter, recent work by the second author [Hof02] shows that a large class of functions on lists definable in a system with higher-order functions can be computed in bounded space. Another step is to consider inference mechanisms for adding resource annotations, including the $\diamond$ arguments (we mentioned some progress on this in Section 1) and usage aspects, as well as the possibility of automatically choosing between $\otimes$-types and $\times$-types. Other work-in-progress was mentioned at the end of the previous section. We are supporting some of the theoretical work with the ongoing development of an experimental prototype compiler for LFPL; see the first author's web page for more details.

Acknowledgements. The authors are grateful to Michal Konečný and Robert Atkey for discussion and comments on this work.

References

[AC02] David Aspinall and Adriana Compagnoni. Heap bounded assembly language. Technical report, Division of Informatics, University of Edinburgh, 2002.

[AH01] David Aspinall and Martin Hofmann. Heap bounded functional programming in Java. Implementation experiments, 2001.

[BS96] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings ACM Principles of Programming Languages*, pages 262–275, 1999.

[DP93] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, September 1993.

[Hof99] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*, pages 464–476. IEEE, Computer Society Press, 1999.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[Hof02] Martin Hofmann. The strength of non size-increasing computation. In *Proceedings ACM Principles of Programming Languages*, 2002.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM.

[Kob99] Naoki Kobayashi. Quasi-linear types. In *Proceedings ACM Principles of Programming Languages*, pages 29–42, 1999.

[Ode92] Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407. Springer-Verlag, February 1992. Lecture Notes in Computer Science 582.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–243, 1999.

[OTPT95] P. W. O'Hearn, M. Takeyama, A. J. Power, and R. D. Tennent. Syntactic control of interference revisited. In *MFPS XI, Conference on Mathematical Foundations of Program Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[PJW00] Simon Peyton-Jones and Keith Wansbrough. Simple usage polymorphism. In *Proc. 3rd ACM SIGPLAN Workshop on Types in Compilation, Montreal*, September 2000.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *Proc. Fifth ACM Symp. on Princ. of Prog. Lang. (POPL)*, 1978.

[Rey00] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[Sha99] Natarajan Shankar. Efficiently executing PVS. Technical report, Computer Science Laboratory, SRI International, 1999.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782, pages 366–381. Springer LNCS, 2000.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Wad90] Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, 1990. North-Holland.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193, 1998.

[WSR00] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Proceedings Compiler Construction, CC 2000*, 2000.

[YR97] H. Yang and U. Reddy. Imperative lambda calculus revisited, 1997.