JFP **18** (2): 141–178, 2008. © 2007 Cambridge University Press
doi:10.1017/S0956796807006399 First published online 5 July 2007 Printed in the United Kingdom

A type system with usage aspects

DAVID ASPINALL
LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
(*e-mail:* David.Aspinall@ed.ac.uk)
http://homepages.inf.ed.ac.uk/da/

MARTIN HOFMANN
Institut für Informatik, Oettingenstraße 67, 80538 München, Germany
(*e-mail:* mhofmann@informatik.uni-muenchen.de)
http://www.tcs.informatik.uni-muenchen.de/~mhofmann/

MICHAL KONEČNÝ
Aston University, Aston Triangle, Birmingham, B4 7ET, UK
(*e-mail:* m.konecny@aston.ac.uk)
http://www.aston.ac.uk/~konecnym/

Abstract

Linear typing schemes can be used to guarantee non-interference 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 limitation 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 direction by defining a new typing scheme that better approximates the semantic property of soundness of in-place update for a functional semantics. We begin from the observation that some data are used only in a "read-only" context, after which it may be safely re-used before being destroyed. Formalising the in-place update interpretation in a machine model semantics allows us to refine this observation, motivating three *usage aspects* apparent from the semantics that 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 with the result. The main novelty is aspect (2), which allows a linear value to be safely read and even aliased with a result of a function without being consumed. This novelty makes our type system more expressive than previous systems for functional languages in the literature. The system remains simple and intuitive, but it enjoys a strong soundness property whose proof is non-trivial. Moreover, our analysis features principal types and feasible type reconstruction, as shown in M. Konečný (In *TYPES 2002 workshop, Nijmegen, Proceedings*, Springer-Verlag, 2003).

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 available. These principles are sound and do not need to use stores or other auxiliary entities,

```
reverse_aux : L(A),L(A) → L(A)
def reverse_aux(l,acc) =
      match l with
              nil ⇒ acc
            | cons(h,t) ⇒ reverse_aux(t,cons(h,acc))

reverse : L(A) → L(A)
def reverse(l) = reverse_aux(l, nil)
```

Fig. 1. Functional list reverse.

as is invariably the case when reasoning about imperative programs, as, e.g. in Reynolds' separation logic (Reynolds 2002).

Consider the functional implementation of linked list reversal, as shown in Figure 1. 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 (see, e.g. Dor *et al.* 2000).

The advantage of the usual imperative implementation, of course, is that it modifies its argument in-place, whereas with the usual functional implementation the result must be created from scratch and garbage collection is necessary to salvage heap space. However, if by static analysis a clever compiler could determine that the original list `l` is not used after the call `reverse(l)`, it would be safe to instead use an optimised in-place update implementation, which avoids creating garbage. To be safe, one must trust both the compiler optimisation and the static analysis.

If we want to simplify the static analysis and ensure that an in-place update implementation is always possible, we can restrict programs, using a linear typing scheme. With linear typing, every variable is used exactly once (or, in an affine linear system, at most once). However, it is well known that pure linear schemes are overly restrictive for real programming and need to be relaxed so that variables can be used more than once as far as possible. For example, consider the function `sumlist` that operates on lists of integers:

```
sumlist : L(N) → N
def sumlist(l) =
      match l with
              nil ⇒ 0
            | cons(h,t) ⇒ h + sumlist(t)
```

In a purely linear type system, we would not be able to use the list `l` after a call to `sumlist(l)`, but this function merely examines its argument; the list would actually remain intact under any reasonable implementation so the linearity restriction that supposes it may be destroyed is overly restrictive.

Not only should the `sumlist` function inspect its argument list without modifying it, but the result it returns no longer refers to the argument list. This means that we should be able to use `l` after the call to `sumlist`, so an expression like

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

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

could be safely compiled even with the in-place implementation of `reverse`, if we assume that evaluation occurs from left to right and the list `l` is not used again afterwards.

There is less freedom with functions that return a result that contains some part of the argument. An example is the function `nth_tail` that returns the *n*th tail of a list:

```
nth_tail : N,L(A) → N
def nth_tail(n,l) =
  if n<=0 then l else match l with
                           nil ⇒ nil
                         | cons(h,t) ⇒ nth_tail(n-1, t)
```

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

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

will be sound, but

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

will *not* be soundly implemented using the in-place update version of `reverse`. 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]]` because of the aliasing. So, if we want to guarantee to use the in-place implementation of `reverse`, the second expression should not be allowed in the language. Simpler example functions in the same category as `nth_tail`, whose results share with the argument, include projection functions and the identity function.

As another example, consider the `append` function:

```
append : L(A),L(A) → L(A)
def append(l,m) =
       match l with
               nil ⇒ m
             | cons(h,t) ⇒ cons(h,append(t,m))
```

The desirable imperative implementation of `append` joins the second list to the first one in-place, by modifying the final pointer; it returns its first argument that now points to a new list. Thus, in `append(l,m)` 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`.

1.1 Usage aspects

Together with consideration of a machine model semantics for in-place update, these observations lead us to introduce three *usage aspects* for variables, which are the central innovation in our type system. The usage aspects are as follows:

- Aspect 1: modifying use, e.g. `l` in `reverse(l)` and `append(l,m)`
- 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)`.

As a first intuition, the numbers are in increasing order of “permissiveness to reuse” in the terms. Variables may be accessed only once with aspect 1. Variables can be used many times with aspect 2, but it 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. It will become apparent later (e.g. Lemma 5.6) how the ordering can be specified precisely and that it demonstrates itself also in other forms than the one suggested by the intuition of “permissiveness to reuse”. The fact that the usage aspects are numbered, rather than named mnemonically, will allow us to make use of the ordering in the presentation of our type system.

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

$$\begin{array}{rcl}
\mathtt{reverse} & : & \mathsf{L}(A)^1 \to \mathsf{L}(A) \\
\mathtt{sumlist} & : & \mathsf{L}(A)^3 \to \mathsf{N} \\
\mathtt{nth\_tail} & : & \mathsf{L}(A)^2, \mathsf{N}^3 \to \mathsf{L}(A) \\
\mathtt{append} & : & \mathsf{L}(A)^1, \mathsf{L}(A)^2 \to \mathsf{L}(A)
\end{array}$$

Types, such as N, that do not involve any heap storage in our machine model can always have the read-only aspect 3. We call these types *heap-free*. Functions that have a heap-free result (like `sumlist`) may have aspect 3 for their non-heap-free arguments, provided that they are not modified when computing the result.

As will be seen later, these usage aspects have an intuitive interpretation in our semantics but, perhaps surprisingly, the exact distinctions appear to be novel. Although ours is not the first type system that relaxes strict linearity, we believe that it is more expressive than other systems for functional languages, while remaining simple and intuitive. Although even more expressive typing systems and specialised program logics for aliasing exist, especially for low-level imperative languages, we believe that our type structure has the advantage of being considerably less complex than others, so that types may be more easily understood by the programmer. The apparent simplicity does not mean that our system is easy to prove sound; our soundness proof (that goes beyond traditional type safety) is quite involved and reveals subtleties in the precise interpretation of the usage aspects. A further advantage of our system is that it enjoys principal types and a feasible type reconstruction algorithm; this property is demonstrated elsewhere (Konečný 2003b). Specific comparison with other work appears at the end of the paper.

1.2 Outline

In Section 2, we introduce our setting for studying usage aspects, which is within a first-order linear functional programming language called LFPL, first introduced in Hofmann (2000). The particular appeal of LFPL for us here is that it has a canonical in-place update interpretation by using a special *resource type* that mediates construction and destruction of inductive data types. As well as adding usage aspects, we augment LFPL with a richer-type structure, including two kinds of products ($\otimes$ and $\times$), that allows data structures with or without sharing to be defined. We then give some larger examples that illustrate our usage aspects.

In Section 3, we present the formal syntax and typing rules. Although the types themselves are simple, some of our rules have slightly intricate side conditions; but they can be explained by an intuitive reading of the meaning of the aspects. In Section 4, we give an imperative operational semantics for LFPL that formalises how its in-place interpretation works; practical implementations are described elsewhere.

In Section 5, we prove our central soundness result, which makes precise the meaning of usage aspects and establishes that the in-place operational semantics agrees with a safe, classical call-by-value operational semantics that is known to correspond to the functional set-theoretic interpretation. This result justifies reasoning about the functional semantics and also proves that the various possibilities for aspect annotations in our typing rules are sound. Significantly, this result goes beyond most other published type safety results for type systems that do not connect operational semantics with denotations. Finally, Section 6 concludes with a comparison to some of the related work.

2 LFPL with usage aspects

Whereas functional languages enjoy powerful proof principles, imperative implementations that directly manipulate pointers enjoy efficiency. We are interested in having the best of both worlds by using a semantics-preserving translation of functional programs into imperative ones that use in-place update and need no garbage collection, as far as possible. In previous work by the second author (Hofmann 2000), a first-order functional language called LFPL was defined, together with such a translation into C.

LFPL relies on some programmer assistance to manage memory but without compromising the functional semantics in any way. This assistance is enabled by augmenting (non-nullary) constructors of inductive data types such as `cons` with an additional argument of an abstract “diamond” resource type $\diamond$. The elements of $\diamond$ can be thought of as heap space areas; the diamond type corresponds exactly to the heap space for storing a single cell of some inductive data type.

To construct an element of an inductive type, we must supply a value of this type. Values of type $\diamond$ can be obtained from formal parameters of functions or by deconstructing elements of recursive types in a pattern match. Consider the functions `reverse` and `append` discussed earlier, and implemented in LFPL in Figure 2. The functions have almost identical definitions as before, except that there is now an extra argument to `cons` both in pattern matches and in constructor positions. The pattern match releases a $\diamond$ cell used for a list node, whereas the constructor consumes one. The first argument to each use of `cons` in Figure 2 is a 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 appear in data structures.

Although it may be realised as a pointer, there is no way to directly examine or manipulate an element of $\diamond$, and in fact values of $\diamond$ are interchangeable in a

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

```
reverse_aux : L(A),L(A) → L(A)
def reverse_aux(l,acc) =
       match l with
                nil ⇒ acc
              | cons(d,h,t) ⇒ reverse_aux(t,cons(d,h,acc))

reverse : L(A) → L(A)
def reverse(l) = reverse_aux(l, nil)

append : L(A),L(A) → L(A)
def append(l,m) =
       match l with
                nil ⇒ m
              | cons(d,h,t) ⇒ cons(d,h,append(t,m))
```

Fig. 2. LFPL examples.

program. In the functional semantics `cons(d,h,(cons(d',h',t)))` is equivalent to `cons(d',h,(cons(d,h',t)))`.

Our LFPL implementations to date use a single $\diamond$ type that is large enough to store a cell of any data type in the program, so a `d` for a list `cons` constructor is also interchangeable with a `d` for a tree `node` constructor. This largest-sized $\diamond$ approach is potentially wasteful but gives flexibility and helps to avoid garbage collection. It could be improved by using several differently sized $\diamond$ types. The sizes in particular positions could be inferred using a data-flow analysis of the program.

2.1 Linear typing in LFPL

In Hofmann (2000) it was shown that the semantics of a translation of LFPL into C preserves the functional semantics of the source program *provided* the latter admits an affine linear typing for inductive types and $\diamond$ types, i.e. bound variables of inductive type are used at most once. The heap-free types were *not* subject to any linearity restriction. Linearity guarantees that the memory space pointed to by a $\diamond$ value is not needed anywhere else, which controls the space usage of programs. It prevents function definitions such as:

```
twice : L(A) → L(N)
def twice(l) =
  match l with
    nil ⇒ nil
  | cons(d,h,t) ⇒ cons(d,0,cons(d,0,twice(t)))
```

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![1]

[1] Incidentally, we can implement `twice` in LFPL if we use another typing:

$$\mathsf{L}(\mathsf{N}\otimes\diamond)\rightarrow\mathsf{L}(\mathsf{N})$$

where the argument list provides the right amount of extra space. In recent work with Steffen Jost (Hofmann & Jost 2003) and Dilsun Kırlı, we have shown that such additions of $\diamond$ types can be automatically inferred, using integer linear programming.

As one would hope, the translation of `append` in Figure 2 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.

Linear typing together with the resource type $\diamond$ seems restrictive at first sight. In particular, without dynamic creation of memory in the translation, we are heap-bounded: 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 Hofmann (2000) it was shown that every non-size-increasing function on lists over booleans in the complexity class ETIME can be represented. The language is therefore useful where careful control over space usage is needed. If the typing is made pure linear instead of affine, we could also prevent space leaks.

Nevertheless, if heap-boundedness is too restrictive for a particular domain, one can easily add constants to the language that generate and destroy elements of type $\diamond$, implemented using an external memory allocator; typings are shown later. If we moreover assumed existence of a garbage collector, we could alleviate the programmer from needing to always free memory. In this case we are getting back to the ordinary functional programming setting of the Introduction, except that we have the new forms of pattern matching and data construction available, which can be used to give the compiler directions about ways to reuse memory.

2.2 Adding usage aspects

Whether or not we are heap-bounded, the linear restrictions remain for the in-place memory reuse interpretation. As with any other typing scheme, the linear restriction rejects many semantically valid programs. In our context, a program is semantically valid if its translation to imperative code computes its functional semantics.[2] We cannot catch all semantically valid programs by a decidable 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 that pass the typing discipline.

Here 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 reachable from this variable. This assumption is overly conservative: multiple uses of a variable need not compromise semantic validity, as long as only the last one is potentially destructive, and provided the results of the earlier accesses do not interfere with an ultimate destructive access.

We model this observation by relaxing the linearity by adding the usage aspects motivated in the Introduction and using them to control the typing. The starting

[2] We could show a slightly stronger soundness condition: a well-typed program in fact evaluates with *benign sharing* as defined in Hofmann & Jost (2003). It means that the program will pass certain conservative run-time checks of non-interference and thus the agreement with functional semantics is not merely accidental.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

```
sort  :  L(L(N))^1 → L(L(N))
def sort(l) =
  match l with
          nil ⇒ nil
        | cons(d,h,t) ⇒ insert(d,h,sort(t))

insert  :  ◇^1,L(N)^2,L(L(N))^1 → L(L(N))
def insert(d,l,ll) =
  match ll with
          nil ⇒ cons(d,l,nil)
        | cons(d',h,t) ⇒
            if sumlist(l) > sumlist(h)
            then cons(d',h,insert(d,l,t))
            else cons(d,l,cons(d',h,t))
```

Fig. 3. Example: Insertion sort.

point is with the typing for cons:

$$\mathsf{cons}_A \qquad : \qquad \diamond^1, A^2, \mathsf{L}(A)^2 \to \mathsf{L}(A)$$

The aspect 1 annotation indicates that the diamond argument `d` is destroyed in `cons(d,h,t)` and must not be used again. Any function that consumes a diamond element will give it aspect 1. The aspect 2 annotations on `cons` indicate that the head and tail arguments of the constructor are not destroyed, and moreover, they share with the result of the construction.

The typing rule for pattern matches allows some flexibility in usage aspect labels:

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

This rule says that the usage aspect for a list x in an expression matching against this list will be the most destructive aspect among the aspects exhibited for the pieces of a cons cell in the e_{cons} expression. This rule now allows both destructive matches (e.g. where the diamond x_d is used to construct a new cell in e_{cons}, as in `reverse`) and non-destructive matches (e.g. where the diamond x_d is not used and the list is only examined, as in `sumlist`).

The typing of `cons` and the rule for list elimination allow us to derive the usage aspect typings given in the Introduction for the corresponding LFPL programs.

A longer example is the LFPL implementation of insertion sort in Figure 3. The `sort` function here sorts a list of integer lists in ascending order of their sums. In the comparison test `sumlist(l) > sumlist(h)` both variables are used with aspect 3 and thus these variables can be safely re-used in the branches of the **`if`** statement. The type of the `insert` function indicates that `insert(d,a,l)` consumes the diamond `d`, inserts the data item `a` without modifying it, but sharing with the result; moreover, it destroys the input list `l`.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

2.3 Sharing data

The strict linear-type system of LFPL as presented in Hofmann (2000) 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 do not refer to x or y in e after the other has been modified. Similarly, we would like to allow a tree that has sharing amongst sub-trees, in the simplest case a node constructed as follows:

```
let u=sharednode(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 are DAGs.

The “not modifying both parts” flavour of these examples leads us to include two kinds of products in our augmented version of LFPL. Consider binary trees. In a linear setting we have two kinds of trees: one corresponding to trees laid out in full in memory (⊗-trees) and the other corresponding to trees with sharing of sub-trees (×-trees). In ordinary functional programming these two are extensionally equivalent; in the presence of linearity constraints they differ considerably. The ⊗-trees allow simultaneous access to all their components, thus encompassing, e.g. computing the list of leaf labellings, whereas access to the ×-trees is restricted to essentially search operations. Conversely, ⊗-trees are more difficult to construct because of their storage requirements, whereas the typing rules allow easy construction of a full binary ×-tree that is represented as a rather small DAG. Thus the type system can reflect the kind of choices that a programmer would normally make in selecting the best data representation for a purpose.

The tensor product (denoted by ⊗) is accessed using the pattern matching construct

```
match p with (x⊗y) ⇒ e
```

that allows both x and y to be accessed simultaneously in *e*. Given a ⊗-product of two lists in a pure linear-type system like that of LFPL, we can access (maybe modify) both components; to be sound, the two lists must have no sharing.

With usage aspects, this constraint is relaxed somewhat: we can construct ⊗-products *within* a program whose components share in certain cases when it is safe to do so, so long as the sharing does not escape the scope of an expression. A contrived but short example is the expression:

```
let l2 = l
    p = l⊗l2
 in match p with (x⊗y) ⇒ sumlist(x) + sumlist(y)
```

Here, the ⊗-product *p* is allowed to exhibit sharing because both of its components are used read-only. (The typing derivation for this term can be understood in

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

conjunction with the typing rules that will be explained in Section 3. In particular, the rules do not permit to form a tensor product `1⊗1` directly.)

The cartesian product (denoted ×), 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. Again, with 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 typing rules for let and cartesian product have special side conditions that allow this behaviour.) Cartesian products are accessed via projection functions:

$$\begin{array}{lll} \texttt{fst} & : & (A \times B)^2 \to A \\ \texttt{snd} & : & (A \times B)^2 \to B \end{array}$$

The usage aspect 2 here indicates that the result shares with the argument and allows us to use *both* components later in the program. Such usage would not be possible in a pure linear-type system with cartesian products.

To allow sharing to persist within data structures used in a program, we may give constructors arguments of cartesian product types. Ideally, we would allow the user to choose exactly where cartesian products are used and where ⊗-products are used, to allow the user to define data types appropriate for their application. For the purpose of exposition in this paper, however, we will include only tensor lists and tensor tree types as primitives (as in LFPL); however, the language and results are easily extended to their variants using cartesian products, or indeed, a general data type mechanism that would allow the user to choose.

2.4 Further examples

We will now describe two prototypical examples of aliasing that can be proved safe by our usage aspects and are not too far away from real-life programming scenarios. In Figure 4, a number of functions are defined that process lists of numbers. There are several supporting functions as well as the two example functions, called S1 and S2. The flow of data in the terms defining S1 and S2 is illustrated in Figure 5, including aspect annotations. The detail of the labelling for the annotations will be better understood by the reader after considering the typing rules given in the next section; however, the motivation of the examples can be explained first.

Function S1 takes a list of lists of numbers and returns one element of this list. The returned element is the first element of the input list that is similar to another element that is, in some sense, best. For brevity, we use simple notions of measure and similarity: we compare lists of numbers by their sums and consider two lists similar if they have the same length and all their elements are similar, i.e. differ by at most one. The important notion in this example is the following data flow: the input is first processed read-only, creating an aliased piece of data, namely the "best" element. Then both the original list and its "best" element are used as arguments to `first_similar`. This sharing of arguments is safe because neither of them is destroyed in the final step and only one of them is aliased with the final result.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

```
highest_sum : $L(L(N))^2 \to L(N)$
def highest_sum(l) =
   match l with
     nil ⇒ nil
   | cons(d,h,t) ⇒
      hs_aux(h,t)
```

```
hs_aux : $L(N)^2, L(L(N))^2 \to L(N)$
def hs_aux(prev,l) =
   match l with
     nil ⇒ prev
   | cons(d,h,t) ⇒
       if sumlist(h) > sumlist(prev)
       then hs(h,t)
       else hs(prev,t)
```

```
first_similar : $L(N)^3, L(L(N))^2 \to L(N)$
def first_similar(m,l) =
   match l with
     nil ⇒ nil
   | cons(d,h,t) ⇒
       if is_similar(m,h)
        then h
        else first_similar(t)
```

```
filter_similar : $L(N)^3, L(L(N))^1 \to L(L(N))$
def filter_similar(m,l) =
   match l with
     nil ⇒ nil
   | cons(d,h,t) ⇒
       let r = filter_similar(m,t)
       in if is_similar(m,h)
          then cons(d,h,r)
          else r
```

```
is_similar : $L(N)^3, L(N)^3 \to N$
def is_similar(l1,l2) =
  match l1 with
    nil ⇒
      match l2 with
        nil ⇒ 1
      | cons(d,h,t) ⇒ 0
  | cons(d1,h1,t1) ⇒
     match l2 with
       nil ⇒ 0
     | cons(d2,h2,t2) ⇒
         similar_number(h1,h2)
         * is_similar(t1,t2)
```

```
similar_number : $N^3, N^3 \to N$
def similar_number(n1,n2) =
  (n1 - n2 < 2)
  * (n2 - n1 < 2)
```

```
S1 : $L(L(N))^2 \to L(N)$
def S1(l) =
   let b = highest_sum(l)
   in first_similar(b,l)
```

```
S2 : $L(N)^1, L(L(N))^1, \diamond^1 \to L(L(N))$
def S2(m,l,d) =
   let l' = filter_similar(m,l)
   in map_reverse(cons(d,m,l'))
```

Fig. 4. Functions demonstrating the usefulness of usage aspects 2 and 3.

Function S2 is more involved but has less intricate effects. It, again, takes a list of lists of numbers l as its primary argument, but it takes also a list of numbers m and a single diamond d. The function S2 filters from l all elements that are similar to m and then adds m back into this list. Finally it processes the resulting list in some destructive way, e.g. reverses all its elements in place. The important notion for S2 is the guarantee that after filtering, m is not only kept intact but is also separate from the result of the filtering, shown by its aspect 3 on its input to filter_similar.

These examples justify the need for all three aspects. If one would have only two aspects, leaving out, say, aspect 2, then S1 would not type-check. If one would leave out aspect 3, then S2 would not type-check. The examples give some hint about why the soundness proof for our system becomes involved: one has to reason about the annotations in different ways when they appear in different positions, in particular, considering both aliasing and separation of portions of heap data as

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

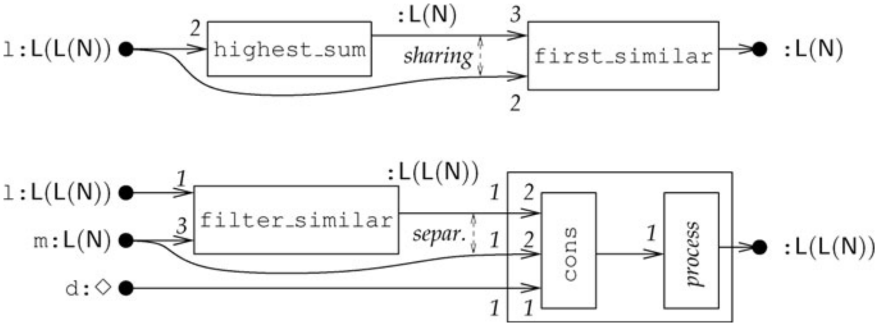


Fig. 5. The data flow and usage aspects within the terms S1 and S2.

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

Fig. 6. LFPL abstract syntax grammar.

shown in Figure 5. We will return to these examples again when we compare our system to the related work in Section 6.

3 Syntax and typing

The grammar for the types and terms of our language is given in Figure 6.

3.1 Types

Types consist of integers N, the resource type $\diamond$, lists $\mathsf{L}(-)$ and trees $\mathsf{T}(-)$, and the binary $\otimes$-product and $\times$-product. For brevity, we will use N also for the type of booleans. Types not containing diamonds $\diamond$ or $\mathsf{L}(-)$ or $\mathsf{T}(-)$ are called *heap-free*, e.g. N and $\mathsf{N}\otimes\mathsf{N}$ are heap-free. (A pair is stored in a single logical memory cell as explained in Section 4.)

3.2 Terms and programs

A program consists of a series of (possibly mutually recursive) function definitions of the form $f(x_1,\ldots,x_n) = e_f$. We use x and variants to range over variables and f to range over function symbols. To simplify the presentation, we restrict the syntax so that most term formers can be applied only to variables, like in the standard A-normal form (Wikipedia 2007; Sabry & Felleisen 1993). In practice, we can define the more general forms such as $f(e_1,\ldots,e_n)$ as syntactic sugar for nested let expressions. Using the same trick, we also impose a convention that in each application $f(x_1,\ldots,x_n)$ the variables $x_1,\ldots,x_n$ are equal to the formal parameters used in e_f. This convention greatly simplifies definitions and various rules related to function applications.

The function definitions e_f must be well-typed. To make this typing possible in the presence of recursion, 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 $A_1^{i_1}, \ldots, A_n^{i_n} \rightarrow A$. In the typing rules we will assume a fixed program with signature Σ.

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

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

for suitable types A as used in the program.[3] In examples where the types can be easily deduced from the context, we omit the type subscripts in nil, cons, etc.

The comma between argument types is treated as a $\otimes$-product, which means that these typings, and the corresponding elimination rules in Figure 7, describe lists and trees with simultaneous access to sub-components. Hence they must be implemented without sharing, unless the access is guaranteed to be read-only.

3.3 Typing contexts

We keep track of *usage aspects* for variables as introduced above. We write $x{:}^i A$ 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; $|\Gamma|$ denotes the domain of Γ. If $x{:}^i A \in \Gamma$ we write $\Gamma(x) = A$ and $\Gamma[x] = i$.

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

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 $x{:}^2 A$ of an aspect 2 (aliased) variable is replaced with $x{:}^i A$. If we have two contexts Δ_1, Δ_2 that differ only in usage aspects, so $|\Delta_1| = |\Delta_2|$ and $\Delta_1(x) = \Delta_2(x)$ for all x, then we define the merged context $\Gamma = \Delta_1 \wedge \Delta_2$ by $|\Gamma| = |\Delta_1|$,

[3] Thus the term constructor formers in Figure 6 are actually superfluous but make the grammar more comprehensible.

$$\frac{}{\vdash c : \mathsf{N}}\ (\textsc{Const}) \qquad \frac{}{x \overset{2}{:} A \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : A}{\Gamma, \Delta \vdash e : A}\ (\textsc{Weak})$$

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

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

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

$$\frac{\Gamma_1, \Delta_1 \vdash e_1 : A_x \quad \Gamma_2, \Delta_2, x \overset{i}{:} A_x \vdash e_2 : B \quad \text{condition} \star \text{ (see Table 8)}}{\Gamma_1^{i}, \Gamma_2, \Delta_1^{i} \wedge \Delta_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : B}\ (\textsc{Let})$$

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

$$\frac{\Gamma_1, \Delta_1 \vdash e_1 : A_1 \qquad \Gamma_2, \Delta_2 \vdash e_2 : A_2 \qquad \text{condition} \star\star \text{ (see page 17)}}{\Gamma_1, \Gamma_2, \Delta_1 \wedge \Delta_2 \vdash (e_1, e_2) : A_1 \times A_2}\ (\times\textsc{-Pair})$$

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

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

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

Note: $\Delta^{i}[x] = \begin{cases} i & \text{if } \Delta[x] = 2 \\ \Delta[x] & \text{otherwise} \end{cases}$ and $(\Delta_1 \wedge \Delta_2)[x] = \min(\Delta_1[x], \Delta_2[x])$.

Fig. 7. Typing rules.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

$\Gamma(x) = \Delta_1(x)$, $\Gamma[x] = \min(\Delta_1(x), \Delta_2(x))$. The merged context takes the "worst" usage aspect of each variable.

3.4 Richer types and signatures

Although we fix a particular list and tree types and their interpretation for this paper, it should be clear that the language and our results can easily be extended to a general inductive data-type mechanism. Moreover, we can introduce data types that admit sharing by giving suitable typings for constructors in the signature. For example, to add a type of "sharing trees" $\mathsf{ST}(A)$ with unrestricted sharing between components, we could use the constructor 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 sub-trees, 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.

LFPL and our examples do not use it, but it is also straightforward to add explicit allocation and deallocation to the language. One simply needs to include in the program signature two further constants:

$$\begin{array}{lcl} \mathsf{new} & : & \to \diamond \\ \mathsf{dispose} & : & \diamond^1 \to \mathsf{N} \end{array}$$

These built-ins can be implemented by interfacing to an external memory manager; of course, without further restriction, this extension breaks the heap-bounded nature of the system.[4]

3.5 Typing rules

Now we explain the typing rules, shown in Figure 7, which define a judgement of the form $\Gamma \vdash e : A$. Most rules are straightforward. Our system is affine linear, so it includes a rule for weakening, WEAK. In the rules VAR and $\otimes$-PAIR, variables are given the default aspect 2, to indicate their obvious sharing with the result. If the result is a value of a heap-free type, then with RAISE we can promote variables of aspect 2 to aspect 3 to reflect that they manifestly 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.

The two non-standard structural rules give insight into the semantics and allow us to give simplified versions of the other rules that would otherwise require greater flexibility. For example, the rule IF assumes the same context Γ when typing both branches but in fact the usage aspects of variables may differ between e_1 and e_2 – a variable may be used destructively in one branch but not the other; the rule DROP

[4] In other work (Hofmann & Jost 2003; MacKenzie & Wolverson 2004), we have implemented a similar language that does have mechanisms for predicting heap usage in the presence of allocation and deallocation.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

$\Delta_1[z] \rightarrow$	1			2			3		
$\downarrow i/\Delta_2[z] \rightarrow$	1	2	3	1	2	3	1	2	3
1	✗	✗	✗	✗	✗	✗	✓	✓	✓
2	✗	✗	✗	✗	✗	✓	✓	✓	✓
3	✗	✗	✗	✗	✓	✓	✓	✓	✓

Fig. 8. The side condition $\star$ of LET in Figure 7: $\forall z \in \Delta_1$ the table has to show a tick.

is used in this case to unite the contexts. The rule RAISE allows us to simplify VAR and $\otimes$-PAIR that would otherwise need to be generalised to also allow the aspect 3 in the case that the associated type is heap-free.

The rule LET is somewhat intricate. To type let $x = e_1$ in e_2, the context is split into three pieces: variables specific to the definition e_1, in Γ_1; variables specific to the body e_2, in Γ_2; 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_x. Then we type-check the body, using some usage aspect i for the bound variable x. The usage aspect the bound variable x has in the body is used for any aliased variables belonging to e_1. For example, if x is used destructively in e_2, then all aliased variables in Γ_1 and Δ_1 are used destructively in the overall expression; this observation accounts for the use of Γ_1^i and Δ_1^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 "$\star$" is defined by Figure 8.[5] It 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 x, provided it is not partly or completely destroyed in e_2: the modification may happen before the reference). Also, it is not safe to have $\Delta_1[z] = \Delta_2[z] = 2$ when $i = 2$ as such combination would allow one to create an illegally sharing tensor pair by let $x = z$ in $x \otimes z$.

As an instance of LET, we get a derived rule of contraction for aspect 3 variables:

$$\frac{\Gamma, x \overset{i}{:} A, y \overset{3}{:} A \vdash e : B \qquad i \geqslant 2}{\Gamma, x \overset{i}{:} A \vdash e[x/y] : B} \quad (\text{CONTR})$$

where $e[x/y]$ stands for let $y = x$ in e. This rule is comparable to the rule of contraction given in Hofmann (2000) that allows contraction only for heap-free types A.

The only constructor rules we need are for the two kinds of pairs because the other constructors are function symbols in the signature. The rule for constructing

[5] The table can be derived from the proof of soundness in Subsection 5.2, see Figure 14 on page 168.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

a $\times$-pair ensures that all variables that are shared between the components have aspect at least 2. The "condition $\star\star$" in rule $\times$-PAIR is

- $\Delta_1[z] \geqslant 2$ and $\Delta_2[z] \geqslant 2$ for all $z \in |\Delta_1| = |\Delta_2|$, $(\star\star)$

which ensures that no part of memory shared between the components is destroyed when the pair is constructed. If we made the evaluation or compilation dependent on the typing derivation, choosing which component to evaluate first, we could generalise condition $\star\star$ slightly, to say that either

- $\Delta_1[z] \geqslant 2$ and $\Delta_1[z] = 3 \Longrightarrow \Delta_2[z] \geqslant 2$ for all $z \in |\Delta_1| = |\Delta_2|$ or
- $\Delta_2[z] \geqslant 2$ and $\Delta_2[z] = 3 \Longrightarrow \Delta_1[z] \geqslant 2$ for all $z \in |\Delta_1| = |\Delta_2|$.

Intuitively, the reason for this condition is that we must be able to evaluate e_1 after e_2 or *vice versa* without corrupting the previous result.

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. These rules could be simplified by unifying all usage aspects and relying on the RAISE rule. Nevertheless, we feel that the explicit calculation of the result aspect better expresses the intuition.

4 Imperative operational semantics

To establish the soundness of our typing rules, we need to formalise the intended in-place update interpretation of the language. In Hofmann (2000), a translation to C and a semantics for the target sub-language of C was used. Here we use an abstract machine model instead; this approach allows us to more easily consider alternative translations to other languages, such as the typed assembly language interpretation, as given in Aspinall and Compagnoni (2003).

Let Loc be a set of *locations* that 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:

$$\begin{array}{lcl} v & ::= & c \mid l \mid \mathsf{NULL} \mid (v,v) \\ h & ::= & \{\mathsf{f}_1 = v_1 \ldots \mathsf{f}_n = v_n\} \end{array}$$

A stack value is a constant c (in our case an integer), a location l, a null value NULL, or a pair of stack values (v,v). A heap value is an n-ary record consisting of named fields with stack values. The operational semantics is based around an abstract notion of stack and heap. A stack $S\colon \mathsf{Var} \rightharpoonup \mathsf{SVal}$ is a partial mapping from variables to stack values, and a heap $\sigma\colon \mathsf{Loc} \rightharpoonup \mathsf{HVal}$ 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'.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

The only way the heap is modified is as a side effect of evaluating constructors that take $\diamond$ arguments, following our in-place update interpretation of the language.

The stack is extended with additional variable bindings whenever we enter a new scope, inside sub-terms in the premises of the evaluation rules. When we evaluate a function body, we use a stack that mentions only the actual parameters, intuitively preventing access beyond the stack frame. Notice that the stack may contain pointers into the heap (i.e. locations) but there are no pointers going from the heap into the stack.

Most of the rules defining the evaluation relation are in Figure 9. The rule APP does not apply to the built-in functions, such as nil, cons, because they have no bodies. Thus we also need the obvious rules for evaluating arithmetic and comparison operators, conditional expressions, and rules for trees, which are similar to those for lists. To represent trees on the heap, we store leaves as records $\{\mathsf{label} = a\}$ and nodes as records $\{\mathsf{label} = a, \mathsf{left} = t_l, \mathsf{right} = t_r\}$. The only interesting cases in the operational semantics are the ones for the heap data types, which make use of $\diamond$ values as heap locations. In the CONS case, the first argument x_d of cons is a variable of $\diamond$ type. The result is the location $S(x_d)$ where we make the cons cell by updating the heap, using a record with hd and tl fields. The match rule LIST-ELIM-CONS performs the opposite operation, exposing the contents and location of a cons cell.

5 Soundness

In this section we will prove that, for a well-typed program, the imperative operational semantics is sound with respect to a "safe" operational semantics that does not update in-place and is therefore equivalent to the usual functional denotational semantics. The safe operational semantics is expressed by the relation:

$$\eta \vdash e \leadsto_{\mathsf{SF}} a$$

It is defined by very similar rules as the semantics $\leadsto$ described in the previous section except that all references to a heap are dropped, stack values v are replaced with semantic values a and stacks S are replaced with valuations η, which map variables to semantic values.

We do not explicitly define semantic values because they are simply symbolic expressions built from the operators $\mathrm{null}, \mathrm{cons}(h,t)$ for lists, $\mathrm{leaf}(a)$, $\mathrm{node}(a,l,r)$ for trees and constants $c, \diamond$. A formal definition of semantic values is implicit in Definition 5.2, where it is shown how these values can be represented on the heap.

The only rules that differ significantly are the ones that overwrite a heap location, i.e. CONS and similar rules for trees:

$$\eta \vdash \mathsf{cons}(x_d, x_h, x_t) \leadsto_{\mathsf{SF}} \mathrm{cons}(\eta(x_h), \eta(x_t)) \qquad \text{(CONS)}$$

Notice that x_d is completely ignored by this evaluation because no in-place update could possibly take place at this level of abstraction.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

$$S, \sigma \vdash c \rightsquigarrow c, \sigma \qquad (\textsc{const})$$

$$S, \sigma \vdash x \rightsquigarrow S(x), \sigma \qquad (\textsc{var})$$

$$\frac{S(x_1) = v_1, \ldots, S(x_n) = v_n \qquad [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n], \sigma \vdash e_f \rightsquigarrow v, \sigma'}{S, \sigma \vdash f(x_1, \ldots, x_n) \rightsquigarrow v, \sigma'} \qquad (\textsc{app})$$

$$\frac{S, \sigma \vdash e_1 \rightsquigarrow v_x, \sigma' \qquad S[x \mapsto v_x], \sigma' \vdash e_2 \rightsquigarrow v, \sigma''}{S, \sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v, \sigma''} \qquad (\textsc{let})$$

$$\frac{}{S, \sigma \vdash x_1 \otimes x_2 \rightsquigarrow (S(x_1), S(x_2)), \sigma} \qquad (\otimes\textsc{-pair})$$

$$\frac{S(x) = (v_1, v_2) \qquad S[x_1 \mapsto v_1][x_2 \mapsto v_2], \sigma \vdash e \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{match}\ x_p\ \mathsf{with}\ (x_1 \otimes x_2) \Rightarrow e\ \rightsquigarrow v, \sigma'} \qquad (\textsc{pair-elim})$$

$$\frac{S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma' \qquad S, \sigma' \vdash e_2 \rightsquigarrow v_2, \sigma''}{S, \sigma \vdash (e_1, e_2) \rightsquigarrow (v_1, v_2), \sigma''} \qquad (\times\textsc{-pair})$$

$$\frac{S(x) = (v_1, v_2)}{S, \sigma \vdash \mathsf{fst}(x) \rightsquigarrow v_1, \sigma} \quad (\textsc{fst}) \qquad\qquad \frac{S(x) = (v_1, v_2)}{S, \sigma \vdash \mathsf{snd}(x) \rightsquigarrow v_2, \sigma} \quad (\textsc{snd})$$

$$S, \sigma \vdash \mathsf{nil} \rightsquigarrow \mathsf{NULL}, \sigma \qquad (\textsc{nil})$$

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

$$\frac{S(x) = \mathsf{NULL} \qquad S, \sigma \vdash e_{\mathsf{nil}} \rightsquigarrow v, \sigma'}{S, \sigma \vdash \mathsf{match}\ x\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(x_d, x_h, x_t) \Rightarrow e_{\mathsf{cons}} \rightsquigarrow v, \sigma'} \qquad (\textsc{list-elim-nil})$$

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

Fig. 9. Evaluation relation with in-place update.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

We now attempt to formulate the soundness theorem:

Preliminary Theorem 5.1 (*Soundness*)
Assume the following data and conditions:

H1. a well-typed program P over some signature Σ;
H2. a well-typed term $\Gamma \vdash e : A$ over Σ for some Γ, e, A;
H3. a heap σ, a stack S and a valuation η such that
each value $\eta(x)$ is *appropriately represented* by $S(x)$ and σ.

Then we have also

C1. $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ implies that there is a such that $\eta \vdash e \rightsquigarrow_{\mathsf{SF}} a$ and
v, σ' *appropriately represent* value a
(i.e. in-place update evaluation is correct);
C2. $\eta \vdash e \rightsquigarrow_{\mathsf{SF}} a$ implies that there are v and σ' such that $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ and
v, σ' *appropriately represent* value a
(i.e. in-place update evaluation is complete).

The statement of the theorem is not yet in a form ready to be proved because assumption H3 and the conclusions are rather vague. We need to clarify what kinds of heap representations of values are appropriate for this theorem. Contrary to our initial expectation, the soundness proof is highly non-trivial. In particular, the just-mentioned "appropriateness" assumption needs to be elaborated in a non-obvious way with added separation conditions, as expressed by our separation theorem (Theorem 5.4).

5.1 Meaningful stack values in a heap

A stack value is *meaningful* for a particular type and heap if it has a sensible interpretation in the heap for that type, i.e. it *appropriately represents* a semantic value of this type. For instance, if $\sigma(v) = \{\mathsf{hd} = 1, \mathsf{tl} = \mathsf{NULL}\}$ then v would be a meaningful stack value of type $\mathsf{L}(\mathsf{N})$ with respect to σ and it would represent the singleton semantic list [1] (using the usual notation for list expressions). In that same heap (v, v) would be a meaningful stack value of type $\mathsf{L}(A) \times \mathsf{L}(A)$ representing the semantic pair $([1], [1])$. Perhaps surprisingly, the value (v, v) will also be a meaningful stack value of type $\mathsf{L}(A) \otimes \mathsf{L}(A)$ in case it is used in a read-only fashion. This occurs, for example, in the term $f(x \otimes x)$ when $f : (A \otimes A)^3 \rightarrow B$. This means that "meaningfulness" of a stack value depends on the aspect with which the value is going to be used. We will parametrise our interpretation on a product separation flag that reflects this intuition. The flag value $\circ\circ$ indicates that heap separation of the components of all tensor products is required while flag value ⚭ indicates that sharing is allowed even inside tensor products.

To express heap separation, we need first to define the *region* $R_A(v, \sigma)$ of a stack value v of type A in heap σ. It is defined as the least set of locations satisfying the rules in Figure 10 (if such a set exists). This is a type-sensitive notion of reachability. It should be clear that $R_A(v, \sigma)$ is the part of the domain of σ that is relevant for v. According to this intuition, we have $R_A(v, \sigma) = R_A(v, \sigma')$ whenever $\sigma(l) = \sigma'(l)$ for all $l \in R_A(v, \sigma)$. Also, if A is a heap-free type, then $R_A(v, \sigma) = \emptyset$.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

$$R_{\mathsf{N}}(c,\sigma)=\emptyset \qquad\qquad R_{\mathsf{L}(A)}(\mathsf{NULL},\sigma)=\emptyset \qquad\qquad R_{\diamond}(l,\sigma)=\{l\}$$

$$R_{A\times B}((v_1,v_2),\sigma)=R_{A\otimes B}((v_1,v_2),\sigma)=R_A(v_1,\sigma)\cup R_B(v_2,\sigma)$$

$$\frac{\sigma(l)=\{\mathsf{hd}=h,\mathsf{tl}=t\}}{R_{\mathsf{L}(A)}(l,\sigma)=\{l\}\cup R_A(h,\sigma)\cup R_{\mathsf{L}(A)}(t,\sigma)}$$

$$\frac{\sigma(l)=\{\mathsf{label}=v\}}{R_{\mathsf{T}(A)}(l,\sigma)=\{l\}\cup R_A(v,\sigma)}$$

$$\frac{\sigma(l)=\{\mathsf{label}=v,\mathsf{left}=t_l,\mathsf{right}=t_r\}}{R_{\mathsf{T}(A)}(l,\sigma)=\{l\}\cup R_A(v,\sigma)\cup R_{\mathsf{T}(A)}(t_l,\sigma)\cup R_{\mathsf{T}(A)}(t_r,\sigma)}$$

Fig. 10. Heap region of a stack value.

$$c,\sigma\Vdash^{p}_{\mathsf{N}} c \qquad\qquad l,\sigma\Vdash^{p}_{\diamond}\diamond \qquad\qquad \mathsf{NULL},\sigma\Vdash^{p}_{\mathsf{L}(A)}\mathrm{null}$$

$$\frac{v_k,\sigma\Vdash^{p}_{A_k} a_k \text{ for } k=1,2}{(v_1,v_2),\sigma\Vdash^{p}_{A_1\times A_2}(a_1,a_2)} \qquad \frac{\begin{array}{c}v_k,\sigma\Vdash^{p}_{A_k} a_k \text{ for } k=1,2,\\ (p=\circ\circ\Rightarrow R_{A_1}(v_1,\sigma),R_{A_2}(v_2,\sigma)\text{ are disjoint})\end{array}}{(v_1,v_2),\sigma\Vdash^{p}_{A_1\otimes A_2}(a_1,a_2)}$$

$$\frac{\begin{array}{c}\sigma(l)=\{\mathsf{hd}{=}v_h,\mathsf{tl}=v_t\},\quad v_h,\sigma\Vdash^{p}_{A} h,\quad v_t,\sigma\Vdash^{p}_{\mathsf{L}(A)} t,\\ (p=\circ\circ\Rightarrow R_{\diamond}(l,\sigma),R_A(v_h,\sigma),R_{\mathsf{L}(A)}(v_t,\sigma)\text{ are pairwise disjoint})\end{array}}{l,\sigma\Vdash^{p}_{\mathsf{L}(A)}\mathrm{cons}(h,t)}$$

$$\frac{\begin{array}{c}\sigma(l)=\{\mathsf{label}{=}v_a\},\quad v_a,\sigma\Vdash^{p}_{A} a,\\ (p=\circ\circ\Rightarrow R_{\diamond}(l,\sigma),R_A(v_a)\text{ are disjoint})\end{array}}{l,\sigma\Vdash^{p}_{\mathsf{T}(A)}\mathrm{leaf}(a)}$$

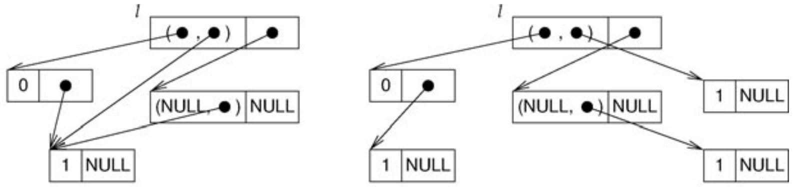
$$\frac{\begin{array}{c}\sigma(l)=\{\mathsf{label}{=}v_a,\mathsf{left}{=}v_l,\mathsf{right}{=}v_r\},\quad v_a,\sigma\Vdash^{p}_{A} a,\quad v_l,\sigma\Vdash^{p}_{\mathsf{T}(A)} l,\quad v_r,\sigma\Vdash^{p}_{\mathsf{T}(A)} r\\ (p=\circ\circ\Rightarrow R_{\diamond}(l,\sigma),R_A(v_a,\sigma),R_{\mathsf{T}(A)}(v_l,\sigma),R_{\mathsf{T}(A)}(v_r,\sigma)\text{ are pairwise disjoint})\end{array}}{l,\sigma\Vdash^{p}_{\mathsf{T}(A)}\mathrm{node}(a,l,r)}$$

Fig. 11. Heap representation relation, with separation flag $p=\circ\circ$ or ⚭.

Definition 5.2
Given a stack value v, a heap σ, a semantic value a of type type A and a product separation flag $p\in\{\circ\circ,\text{⚭}\}$, we define a five-place relation $v,\sigma\Vdash^{p}_{A} a$, which expresses that the semantic value a is *appropriately represented* by a *meaningful* stack value v and heap σ with or without the condition of separation of tensor products. It is defined inductively by the rules shown in Figure 11. □

When we want to ignore the semantic value, we can leave it out and write $v,\sigma\Vdash^{p}_{A}$.

The difference between $\Vdash^{\text{⚭}}$ and $\Vdash^{\circ\circ}$ is that the latter prevents any "internal sharing" within $\otimes$-product types in the heap representation (see Figure 12 for an illustration). We extend this relation to stacks, valuations and typing contexts.



$l, \sigma \Vdash^{\infty}_{\mathsf{L(L(N)\otimes L(N))}} [([0,1],[1]),([],[1])]$ $l, \sigma \Vdash^{\circ\circ}_{\mathsf{L(L(N)\otimes L(N))}} [([0,1],[1]),([],[1])]$

Fig. 12. Heap representation of $[([0,1],[1]),([],[1])]$ with and without sharing.

Definition 5.3
For a stack S, heap σ, typing context Γ, a separation flag p and a valuation η, it holds that $S, \sigma \Vdash^{p}_{\Gamma} \eta$ if

- $S(x), \sigma \Vdash^{p_{\Gamma[x]}}_{\Gamma(x)} \eta(x)$ for each $x \in |\Gamma|$ where $p_1 = \circ\circ$, $p_2 = p$, $p_3 = \infty$ and
- if for any two variables $x \neq y$ the regions $R_{\Gamma,x}(S,\sigma)$, $R_{\Gamma,y}(S,\sigma)$ are *not* disjoint, then $\Gamma[x] \geqslant 2, \Gamma[y] \geqslant 2$ and if $p = \circ\circ$, then moreover either $\Gamma[x] = 3$ or $\Gamma[y] = 3$. □

This definition amounts to saying that $S, \sigma \Vdash^{\infty}_{\Gamma} \eta$ holds if stack S and heap σ are meaningful 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.) The case that $S, \sigma \Vdash^{\circ\circ}_{\Gamma} \eta$ is stronger: variables with aspect 2 have internally separated tensor products and cannot share with each other. These extra separation conditions will guarantee that the result of a computation typed using this judgement is represented with its tensor products separated on the heap.

5.2 *Separation theorem*

With Definitions 5.2 and 5.3 in place, we can prove that the evaluation of a well-typed term under a meaningful stack and heap gives a meaningful result. This is the hardest part of the soundness theorem. Instead of finishing the formulation and proof of the soundness theorem, we will, therefore, first prove a theorem that states the separation properties of the in-place update evaluation and ignores the values themselves. It is easier to focus on separation before considering other aspects of soundness. Our soundness proof in Subsection 5.3 is an extension of the separation proof shown here.

Theorem 5.4 (*Separation*)
Assume the following data and conditions:

H1. a well-typed program P over some signature Σ;
H2. a well-typed term $\Gamma \vdash e : A$ over Σ for some Γ, e, A;
H3. a heap σ and a stack S with $S, \sigma \Vdash^{\infty}_{\Gamma}$
(i.e. arguments are meaningful, separated according to aspects);
H4. $S, \sigma \vdash e \rightsquigarrow v, \sigma'$.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

Then we have also:

C1. $R_A(v, \sigma') \subseteq \bigcup_{x \in \Gamma, \Gamma[x]<3} R_{\Gamma,x}(S, \sigma)$
(i.e. the result is contained entirely within the heap space of arguments with aspect 1 or 2);

C2. if $l \in R_{\Gamma,x}(S, \sigma)$ and $\Gamma[x] \geqslant 2$ then $\sigma(l) = \sigma'(l)$
(i.e. heap occupied by arguments with aspect 2 or 3 is not modified);

C3. $v, \sigma' \Vdash^{\infty}_{A}$ (i.e. the result is meaningful);

C4. $S, \sigma \Vdash^{\circ\circ}_{\Gamma}$ implies $v, \sigma' \Vdash^{\circ\circ}_{A}$
(i.e. if tensors are respected in the arguments, so they are in the result).

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

Corollary 5.5
Let P be a well-typed program having a function symbol $f : \mathsf{L(N)}^i \to \mathsf{L(N)}$.

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 and $[x \mapsto l], \sigma \vdash f(x) \rightsquigarrow v, \sigma'$ for some v, σ', then v points in σ' to a linked list with integer entries.

Moreover, if $i \neq 1$, then the argument list w remains intact in σ'; if $i = 3$, then the heap region $R_{\mathsf{L(N)}}(v, \sigma')$ of the result list is disjoint from the heap region $R_{\mathsf{L(N)}}(l, \sigma)$ of the argument.

Since our system has no memory allocation, we can conclude that if $i = 2$, the result must be some tail of the argument list; and if $i = 3$, the result must be the empty list.

To prove the theorem, we need the following two elementary observations.

Lemma 5.6
Whenever we have two contexts Γ, Γ' that differ only in usage aspects and $\Gamma[x] \leqslant \Gamma'[x]$ for all x in Γ then

1. $S, \sigma \Vdash^{\infty}_{\Gamma}$ implies $S, \sigma \Vdash^{\infty}_{\Gamma'}$
2. $S, \sigma \Vdash^{\circ\circ}_{\Gamma}$ implies $S, \sigma \Vdash^{\circ\circ}_{\Gamma'}$.

In plain words, raising usage aspects weakens the separation conditions.

Lemma 5.7
For any evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, such that $S, \sigma \Vdash^{\infty}_{\Gamma}$ and $v, \sigma' \Vdash^{\infty}_{A}$, it holds

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

In plain words, operational semantics neither overwrites memory that is not referenced via the context nor uses it in the result.

Furthermore, we define a shortcut notation for several assertions about which we reason in the situation of the above lemma. The name *rg* stands for *heap region*,

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

sep stands for *heap region separation*, *tens* for *tensor product separation* and *pres* for *heap preservation.*

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

Moreover, the symbol *res* is treated as a special variable representing the result on the heap σ', i.e. $S(res) = v$, $\Gamma(res) = A$ and in relation to *res* the heap σ' is used instead of σ. When we want to relate these assertions to a particular evaluation $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, we can write $v, \sigma', S, \sigma \Vdash$ in front of it. If there is no occurrence of *res* in it, we can leave v and σ' out. The relevant typing context Γ and result type A will always be clear from the context.

Using this notation, we can express more succinctly the defining properties of heap representation for contexts:

$$
\begin{aligned}
S, \sigma \Vdash^{\circ\!\!\circ}_{\Gamma} &\implies sep(|\Gamma|_1, |\Gamma|) \wedge tens(|\Gamma|_1) \\
S, \sigma \Vdash^{\circ\circ}_{\Gamma} &\implies sep(|\Gamma|_1, |\Gamma|) \wedge sep(|\Gamma|_2, |\Gamma|_2) \wedge tens(|\Gamma|_{1,2})
\end{aligned}
$$

Proof of the separation theorem (Theorem 5.4)

We proceed by induction on the lexicographic product of the depth of the derivations of $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, i.e. computation time (first priority) and of $\Gamma \vdash e : A$ (second priority). In most cases, we consider a derivation step of the operational semantics together with one corresponding typing rule. To be able to derive valid typing judgements for the premises of operational semantics rules, sometimes it is necessary to additionally use the WEAK, DROP and RAISE typing rules on their own.

CONST: Conclusions hold trivially because $\sigma = \sigma'$ and v is heap-free.

VAR: C1 holds because both sides are clearly equal – the region of the result is equal to the region of the only argument and this argument has aspect 2. C2 holds since $\sigma = \sigma'$. C3 follows directly from H3. C4 is similarly trivial.

APP: We assume $e = f(x_1, \ldots, x_n)$ and $\Sigma(f) = A_1^{i_1}, \ldots, A_n^{i_n} \to A$. Thus we have $x_1 :^{i_1} A_1, \ldots, x_n :^{i_n} A_n \vdash e_f : A$ and $S, \sigma \vdash e_f \rightsquigarrow v, \sigma'$, using the operational and typing rules. Now the induction hypothesis applies and gives all the desired conclusions verbatim.

IF-TRUE: We assume $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, $\Gamma = \Gamma_1, x :^3 \mathsf{N}$, $\Gamma \vdash e_i : A$ for $i = 1, 2$, and $S(x) \neq 0$. Thus we must have $S, \sigma \vdash e_1 \rightsquigarrow v, \sigma'$ and the induction hypothesis can be applied to this judgement, and yields the desired conclusions verbatim.

IF-FALSE: Symmetric to IF-TRUE.

WEAK: Assume $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma \vdash e : A$ can be derived from a valid judgement $\Gamma_1 \vdash e : A$. Then conditions H2–H4 still hold after replacing Γ with Γ_1 and S with $S_1 = S|_{\Gamma_1}$. Thus we can apply the induction hypothesis and it yields the desired conclusions except that instead of Γ, S, they refer to Γ_1, S_1 respectively. The desired conclusions follow: C1 because the right-hand side set of locations increases with Γ instead of Γ_1; C2 because heap space occupied by Γ_2 cannot share with Γ_1 except on the region of $|\Gamma_1|_{2,3}$, which is not modified, and the rest

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

is not modified, thanks to Lemma 5.7; C3 is identical to C3 in the induction hypothesis; C4 is weakened by a strengthened premise with S instead of S_1.

DROP: The induction hypothesis can be applied, thanks to Lemma 5.6. The resulting conclusions imply, the desired conclusions because C1,C2,C4 become weaker and C3 remains the same when the aspect is dropped back to its original value.

RAISE: Assume $\Gamma_* \vdash e : A$ for some context Γ_* with $\Gamma = \Gamma_*^3$ and let A be heap-free. Thus H2 holds when replacing Γ by Γ_*. Also H3 holds after the change because the weaker form of heap representation does not make any distinction between aspects 2 and 3. We can, therefore, apply the induction hypothesis and obtain the desired conclusions with Γ replaced by Γ_*. The desired conclusion C3 is identical in this transformation and is thus proved. C4 follows trivially from C3 because A is heap-free. C2 does not make distinction between aspects 2 and 3 and is thus proved directly from the induction hypothesis. Finally, C1 is trivial for a heap-free type A.

$\otimes$-PAIR, NIL, CONS, LEAF **and** NODE: Conclusions follow directly from the assumptions, like in the case of VAR.

PAIR-ELIM: We have $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, 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 $S'' = S[x_1 \mapsto v_1][x_2 \mapsto v_2]$. Furthermore, we have $S, \sigma \Vdash^{\infty}_{\Gamma}$, where $\Gamma = \Gamma', x{:}^{i} A_1 \otimes A_2$. Put $\Gamma'' = \Gamma', x_1{:}^{i_1} A_1, x_2{:}^{i_2} A_2$. We aim at proving $S'', \sigma \Vdash^{\infty}_{\Gamma''}$, in order to be able to apply the induction hypothesis. Basic separation conditions confined within Γ' translate directly from Γ to Γ''. Since $i \leqslant i_1, i_2$, any separation required in Γ'' between x_1 or x_2 and some variable $y \in \Gamma'$ follows from the separation between x and y in Γ. Also, if $tens(x_1)$ or $tens(x_2)$ is required in Γ'', then it can be obtained from $tens(x)$, which must be required in Γ in this case. Finally, if $sep(x_1, x_2)$ is required in Γ'' then $i = 1$, which implies that $tens(x)$ is required in Γ, which is sufficient for $sep(x_1, x_2)$ to hold. This concludes the proof of $S'', \sigma \Vdash^{\infty}_{\Gamma''}$.

From the induction hypothesis we get $v, \sigma' \Vdash^{\infty}_{A}$. It remains to prove the preservation and separation conclusions. Again, conditions confined to Γ' and the result translate from Γ'' trivially. If $i > 1$, we need to show $pres(x)$ and we can derive it from $pres(x_1)$ and $pres(x_2)$, which hold thanks to $i_1, i_2 \geqslant i > 1$. Similarly, we get $rg(res) \subseteq rg(|\Gamma''|_{1,2}) \subseteq rg(|\Gamma|_{1,2})$.

From the separation precondition for $tens(res)$ present in $S, \sigma \Vdash^{\circ\circ}_{\Gamma}$ we can deduce $S'', \sigma \Vdash^{\circ\circ}_{\Gamma''}$ analogously to deducing the ∞ version above. Thus the desired guarantee $tens(res)$ follows from $tens(res)$ for the induction hypothesis, proving C4.

$\times$-PAIR: From $S, \sigma \vdash (e_1, e_2) \rightsquigarrow v, \sigma''$ we get $S, \sigma \vdash e_1 \rightsquigarrow v_1, \sigma'$, $S, \sigma' \vdash e_2 \rightsquigarrow v_2, \sigma''$, and $v = (v_1, v_2)$. It is straightforward to derive the assumptions of the first induction hypothesis, in particular, $S_1, \sigma \Vdash^{\infty}_{\Gamma_1, \Delta_1}$, where $S_1 = S|_{\Gamma_1, \Delta_1}$, from the present assumption $S, \sigma \Vdash^{\infty}_{\Gamma}$, where $\Gamma = \Gamma_1, \Gamma_2, \Delta_1 \wedge \Delta_2$. Thus the conclusions of the first induction hypothesis hold, in particular, $v_1, \sigma' \Vdash^{\infty}_{A_1}$, where $A = A_1 \times A_2$ and $pres(|\Gamma_1, \Delta_1|_{2,3})$.

Analogously, for the second induction hypothesis we can define S_2 and obtain $S_2, \sigma \Vdash^{\infty}_{\Gamma_2, \Delta_2}$. We need to show that this representation still holds after the first evaluation, i.e. $S_2, \sigma' \Vdash^{\infty}_{\Gamma_2, \Delta_2}$. To make this conclusion, we realise that the region of

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

Γ_2, Δ_2 on σ is preserved in the first evaluation, i.e. this region is separated from $|\Gamma_1, \Delta_1|_1$, which follows from $sep(|\Gamma|_1, (|\Gamma_2, \Delta_2|))$ and the side condition that Δ_2 does not contain any variable that has aspect 1 in Δ_1. The other assumptions of the second induction hypothesis are straightforward. Thus we can use the conclusions of the second induction hypothesis, including $v_2, \sigma'' \Vdash^{\boldsymbol{\infty}}_{A_2}$.

Since $R_{A_1}(v_1, \sigma') \subseteq R_{|\Gamma_1, \Delta_1|_{1,2}}(S|_{|\Gamma_1, \Delta_1|_{1,2}}, \sigma)$, the region $R_{A_1}(v_1, \sigma')$ has not been modified by the evaluation of e_2 thanks to $sep(|\Gamma_2, \Delta_2|_1, \Gamma)$ and the side condition. Thus we also have $v_1, \sigma'' \Vdash^{\boldsymbol{\infty}}_{A_1}$. Now we can deduce $(v_1, v_2), \sigma'' \Vdash^{\boldsymbol{\infty}}_{A_1 \times A_2}$.

From the conclusions of both hypotheses, it is easy to deduce also $pres(|\Gamma|_{2,3})$ and $rg(res) \subseteq rg(|\Gamma|_{1,2})$ using the straightforward properties:

$$|\Gamma|_{2,3} \subseteq |\Gamma_1, \Delta_1|_{2,3} \cup |\Gamma_2, \Delta_2|_{2,3} \text{ and } |\Gamma_1, \Delta_1|_{1,2} \cup |\Gamma_2, \Delta_2|_{1,2} \subseteq |\Gamma|_{1,2}\,.$$

When $sep(|\Gamma|_2, |\Gamma|_2)$ and $tens(|\Gamma|_2)$ hold on σ, then the same holds for Γ_1, Δ_1 and Γ_2, Δ_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 separation needed between v_1 and v_2, we get $(v_1, v_2) = v, \sigma'' \Vdash^{\infty}_{A_1 \times A_2}$, i.e. the final conclusion C4.

LIST-ELIM and TREE-ELIM Proofs for these cases can be obtained by a straightforward combination and adaptation of the proofs for IF and PAIR-ELIM.

LET: The skeleton of this part of the proof is in Figure 13. A typical row of the table contains a statement preceded by its code and followed by a list of references to statements and definitions from which it has been derived. (We will elaborate on these, often non-trivial, deductions shortly.) The statements whose names end with *i1* are valid under the condition that $i = 1$, etc. A proof of conclusion C4 includes the dotted parts while the proof of C1–C3 is obtained by ignoring the dotted parts. To improve clarity, the table also includes the definitions of certain important symbols, each definition immediately preceding the first use of the symbol.

The horizontal lines divide the proof into three parts. The first and second parts prove that the induction hypothesis can be applied to the first and second sub-terms (i.e. e_1 and e_2), respectively. The final part shows the validity of conclusions C1–C4 for the whole let expression.

The proof is much more involved than originally anticipated. One unexpected aspect of this proof is that the stronger conclusion C4 of the induction hypothesis for e_1 is needed even when proving conclusions C1–C3 of the theorem when $i = 1$ (see condition *Hr1*). Moreover, even when proving C4, we need C3 and not C4 for the induction hypothesis for e_1 when $i = 3$. Intuitively, we need the stronger ($p = \infty$) interpretation of the theorem even when proving the weaker version ($p = \boldsymbol{\infty}$) and *vice versa*. This means that we cannot decouple the two interpretations from each other and prove them separately; we have no choice but to prove them both in parallel.

The core of the proof lies in the statements *Hs2g*, *S2xi1*, *S2xi2* and *S2xi3*, which claim that the evaluation of e_1 has not interfered with the context for e_2 and prepared the interim result in such a way that it fits the representation criteria of the typing for e_2. These statements directly rely on the side condition. Figure 14

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

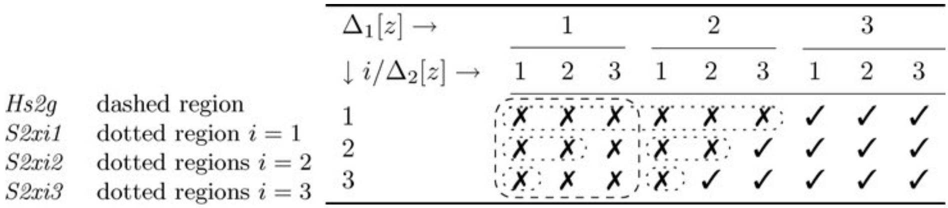
Implicit conditions: $(\ldots i1) \sim (i = 1)$, $(\ldots i12) \sim (i \in \{1,2\})$, etc.
Proof of conclusion C4 additionally includes all the $\boxed{\text{dotted}}$ material.

(Op)	$S,\sigma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \leadsto v,\sigma''$	assumption H4
$(Op1,2)$	$S,\sigma \vdash e_1 \leadsto v_x,\sigma' \quad S[x \mapsto v_x],\sigma' \vdash e_2 \leadsto v,\sigma''$	$Op, \leadsto$
(Hs)	$S,\sigma \Vdash^{\infty}_{\Gamma} \quad \boxed{S,\sigma \Vdash^{\circ\circ}_{\Gamma}}$	assumption H3 $\boxed{C4a}$
(Sep)	$S,\sigma \Vdash sep(\lvert\Gamma\rvert_1,\Gamma)\boxed{\wedge\, sep(\lvert\Gamma\rvert_2,\lvert\Gamma\rvert_2)} \wedge tens(\lvert\Gamma\rvert_{1\boxed{,2}})$	Hs
	$(\Gamma = \Gamma_1^i,\Gamma_2,\Delta_1^i \wedge \Delta_2$ as used in the derivation of $\Gamma \vdash e : A)$	
$(1\text{-}23)$	$\lvert\Gamma_1,\Delta_1\rvert_1 \cup \lvert\Gamma_2,\Delta_2\rvert_1 \subseteq \lvert\Gamma\rvert_1 \quad \lvert\Gamma\rvert_{2,3} \subseteq \lvert\Gamma_1\rvert_{2,3} \cup \lvert\Gamma_2\rvert_{2,3} \cup (\lvert\Delta_1\rvert_{2,3} \cap \lvert\Delta_2\rvert_{2,3})$	
	$(S_1 = S\rvert_{\Gamma_1,\Delta_1})$	
$(Hs1)$	$S_1,\sigma \Vdash^{\infty}_{\Gamma_1,\Delta_1}$	$Hs,\ Sep,\ \Gamma_1^i,\Delta_1^i$
$(Hs1i1\boxed{2})$	$S_1,\sigma \Vdash^{\circ\circ}_{\Gamma_1,\Delta_1}$	$Hs,\ Sep,\ \Gamma_1^i,\Delta_1^i$
$(Hr1i1\boxed{2})$	$v_x,\sigma' \Vdash^{\circ\circ}_{A_x}$	induction 1, C4
$(Hr1i23)$	$v_x,\sigma' \Vdash^{\infty}_{A_x}$	induction 1, C3
$(G1)$	$v_x,\sigma',S_1,\sigma \Vdash pres(\lvert\Gamma_1,\Delta_1\rvert_{2,3}) \wedge rg(res) \subseteq rg(\lvert\Gamma_1,\Delta_1\rvert_{1,2})$	ind. 1, C1,C2
	$(S_2 = S\rvert_{\Gamma_2,\Delta_2})$	
$(Hs2p)$	$S_2,\sigma \Vdash^{\infty}_{\Gamma_2,\Delta_2} \quad \boxed{S_2,\sigma \Vdash^{\circ\circ}_{\Gamma_2,\Delta_2}}$	Hs
$(Hs2g)$	$S_2,\sigma' \Vdash^{\infty}_{\Gamma_2,\Delta_2} \quad \boxed{S_2,\sigma' \Vdash^{\circ\circ}_{\Gamma_2,\Delta_2}}$	$Hs2p,\ Sep,\ 1\text{-}23,\ G1$, side cond.
	$(S_x = S_2[x \mapsto v_x])$	
$(S2xi1)$	$S_x,\sigma' \Vdash sep(x,(\Gamma_2,\Delta_2)) \wedge tens(x)$	$Sep,\ G1,\ Hr1i1\boxed{2}$, side cond.
$(S2xi2)$	$S_x,\sigma' \Vdash sep(x,\lvert\Gamma_2,\Delta_2\rvert_{1\boxed{,2}})\boxed{\wedge\, tens(x)}$	$Sep,\ Hr1i1\boxed{2},\ \Delta_1^i \wedge \Delta_2$, side cond.
$(S2xi3)$	$S_x,\sigma' \Vdash sep(x,\lvert\Gamma_2,\Delta_2\rvert_1)$	Sep, side cond.
	$(\Gamma_x = \Gamma_2,\Delta_2,x \overset{i}{:} A_x)$	
$(Sep2)$	$S_x,\sigma' \Vdash sep(\lvert\Gamma_x\rvert_1,\Gamma_x)\boxed{\wedge\, sep(\lvert\Gamma_x\rvert_2,\lvert\Gamma_x\rvert_2)} \wedge tens(\lvert\Gamma_x\rvert_{1\boxed{,2}})$	$Hs2g,\ S2x$
$(Hs2)$	$S_x,\sigma' \Vdash^{\infty}_{\Gamma_x} \quad \boxed{S_x,\sigma' \Vdash^{\circ\circ}_{\Gamma_x}}$	$Hs2g,\ Hr1,\ Sep2$
$(Hr2)$	$v,\sigma'' \Vdash^{\infty}_{A'} \quad \boxed{v,\sigma'' \Vdash^{\circ\circ}_{A'}}$	induction 2, C3 $\boxed{C4}$
$(G2)$	$v,\sigma'',S_x,\sigma' \Vdash pres(\lvert\Gamma_x\rvert_{2,3}) \wedge rg(res) \subseteq rg(\lvert\Gamma_x\rvert_{1,2})$	induction 2, C1,C2
$(Ri12)$	$rg(res) \subseteq rg(\lvert\Gamma_2,\Delta_2\rvert_{1,2}) \cup rg(\lvert\Gamma_1,\Delta_1\rvert_{1,2}) = rg(\lvert\Gamma\rvert_{1,2})$	$G2,\ G1$
$(Ri3)$	$rg(res) \subseteq rg(\lvert\Gamma_2,\Delta_2\rvert_{1,2}) \subseteq rg(\lvert\Gamma\rvert_{1,2})$	$G2$
$(a23i1)$	$\lvert\Gamma\rvert_{2,3} = \lvert\Gamma_1\rvert_3 \cup \lvert\Gamma_2\rvert_{2,3} \cup (\lvert\Delta_1\rvert_3 \cap \lvert\Delta_2\rvert_{2,3})$	$\Delta_1^i \wedge \Delta_2$
$(S1\text{-}23)$	$(\lvert\Delta_1\rvert_{2,3} \cap \lvert\Delta_2\rvert_{2,3})$ disjoint from $\lvert\Delta_1\rvert_1 \cup \lvert\Delta_2\rvert_1$	$\Delta_1^i \wedge \Delta_2$
(P)	$v,\sigma'',S,\sigma \Vdash pres(\lvert\Gamma\rvert_{2,3})$	$G1,\ G2,\ 1\text{-}23,\ a23i1,\ S1\text{-}23,\ Sep$
(G)	$v,\sigma'',S,\sigma \Vdash pres(\lvert\Gamma\rvert_{2,3}) \wedge rg(res) \subseteq rg(\lvert\Gamma\rvert_{1,2})$	$P,\ R$

Fig. 13. Evaluation of let expressions preserves separation properties.

is a copy of Figure 8, in which it is shown on which forbidden cases each of these four statements relies.

Individual statements up to induction 1. Op is an elaborated version of assumption H4. Statements *Op1,2* follow from the operational LET rule that we assume was used to derive *Op*. Statement *Hs* (i.e. heap representation of the stack) combines H3 with the assumption of C4. *Hs* is further elaborated as *Sep*. Observing the particular form Γ takes thanks to the typing rule LET, we make the elementary observation that any variable with aspect 1 in either of the two sub-expressions has aspect 1 in Γ (*1-23*). The second inclusion in *1-23* is a rephrasing of the first inclusion, focusing on aspects 2,3 instead of aspect 1.



Hs2g	dashed region
S2xi1	dotted region $i = 1$
S2xi2	dotted regions $i = 2$
S2xi3	dotted regions $i = 3$

$\Delta_1[z] \rightarrow$	1			2			3		
$\downarrow i/\Delta_2[z] \rightarrow$	1	2	3	1	2	3	1	2	3
1	✗	✗	✗	✗	✗	✗	✓	✓	✓
2	✗	✗	✗	✗	✗	✓	✓	✓	✓
3	✗	✗	✗	✗	✓	✓	✓	✓	✓

Fig. 14. Condition ⋆ and proof of the LET case.

At this point, all assumptions of the induction hypothesis for e_1 are established by the typing and operational rules except for H3 (i.e. statement *Hs1*). When $i = 1$, we will also need conclusion C4 and will therefore need to strengthen *Hs1* as shown in the table. The stronger stack representation clause is also needed when $i = 2$ while proving C4 for the let expression. This is because separation of tensors will be required within x in these cases.

Why does *Hs1* follow from *Hs*? All separation requirements of *Hs* (i.e. *Sep*) restricted to the variables in Γ_1, Δ_1 are stronger than the separation requirements in *Hs1* by Lemma 5.6 because the aspects of these variables are lower in *Hs* than in *Hs1*. If $i = 1$, then all variables with aspect 2 in Γ_1, Δ_1 gain aspect 1 in Γ and thus, by *Hs*, have their tensors separated and are separated from each other. Thus we have the stronger version of *Hs1* when $i = 1$. When observing the dotted reading, the stronger assumption holds in *Hs* and thus also *Hs1* has separated tensors and variables with aspect 2, unless $i = 3$ because then variables with aspect 2 get aspect 3 in Γ and we do not know anything about them from *Hs*. This concludes the proof of *Hs1* and the induction hypothesis can be used as required, yielding, in particular, statements *Hr1* and *G1*.

Individual statements up to induction 2. The next major goal is to derive the assumptions of the induction hypothesis for e_2 and its corresponding context $\Gamma_x = \Gamma_2, \Delta_2, x{:}^i A_x$ and stack S_x. Again, the only hard part of this part of the proof is to derive that the stack values are represented properly with all the separation conditions as dictated by the aspects in the context (i.e. assumptions H3 and C4a, formulated in statement *H2s*, elaborated as *Sep2*).

It is quite easy to see that, apart from x, all the values used in e_2 are properly separated on the original heap σ (statement *Hs2p*) because their aspects are not raised in Γ, although they may be lowered. To lift this condition to heap σ' (statement *Hs2g*), we need to establish that the entire region of these values was left unmodified by the evaluation of e_1, i.e. there is no aliasing between aspect 1 variables in Γ_1, Δ_1 and the region of S_2 on heap σ. This claim is trivial for Γ_2 because of its inclusion in Γ (statement *Sep*) and the observation *1-23*. For Δ_2, however, we have to additionally use the fact that the side condition forbids a shared variable to appear in e_1 with aspect 1. This concludes the proof of *Hs2g*. Now we need to prove that x on σ' is separated from the remainder of the context Γ_x as required by *Sep2*. To do so, we will consider the three possible values of i separately.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

- $i = 1$ (*S2xi1*): The required internal separation of x follows straight from *Hr1*. The separation of x from the rest of the context follows from the fact that the value of x is contained within the region of variables with aspect 1 or 2 in Γ_1, Δ_1. All these variables are separated from everything else, thanks to *Sep* and the definition of Γ_1^1, Δ_1^1, and are forbidden to appear in Δ_2 by the side condition.
- $i = 2$ (*S2xi2*): The minimum requirement here is that x is separated from all variables with aspect 1 in Γ_2, Δ_2. This claim can be argued in the same way as in the previous case from *Sep* and the side condition.

  In the dotted reading, we also need separation from variables with aspect 2 and the separation of tensors in x, which follows straight from *Hr1*. The separation of x (i.e. variables with aspect 1 or 2 in Γ_1, Δ_1) from variables with aspect 2 in Γ_2, Δ_2 can be derived from *Sep* and the side condition as follows. No variable can appear in both sets under consideration because the side condition does not allow any variable to have aspect 1 or 2 in Δ_1 and aspect 2 in Δ_2. The rest follows directly from the dotted version of *Sep*, considering the fact that the aspects of these variables cannot change to 3 in Γ.
- $i = 3$ (*S2xi3*): The side condition implies that variables with aspect 1 in Δ_2 must have aspect 3 in Δ_1. The variables with aspect 1 in Γ_2, Δ_2 still have aspect 1 in Γ. Thus *Sep* guarantees that variables with aspect 1 or 2 in Γ_1, Δ_1 (which contain the region of x) are separated from the variables with aspect 1 in Γ_2, Δ_2 because the two sets of variables are disjoint.

Putting together all statements from *S2x* and *Hs2g*, we get *Sep2*, which is a rephrasing of *Hs2*. This statement allows us to use the induction hypothesis for expression e_2, yielding the conclusions *Hr2* and *G2*.

Individual statements after induction 2. *Hr2* is equal to the desired conclusions C3 and C4. The remainder of the proof is concerned with proving C1 and C2, which have been rephrased as statement G.

To prove that the result is contained in $|\Gamma|_{1,2}$, consider that we know from *G2* that it is contained in $rg(|\Gamma_x|_{1,2})$, which itself is contained in $rg(|\Gamma_2, \Delta_2|_{1,2}) \cup rg(|\Gamma_1, \Delta_1|_{1,2})$ if $i < 3$ (thanks to *G1*) and is equal to $rg(|\Gamma_2, \Delta_2|_{1,2})$ if $i = 3$. The proof of *Ri3* is concluded when we take into account $|\Gamma_2, \Delta_2|_{1,2} \subseteq |\Gamma|_{1,2}$. The other case (i.e. *Ri12*) needs additionally $rg(|\Gamma_1, \Delta_1|_{1,2}) \subseteq |\Gamma|_{1,2}$, which holds whenever $i < 3$ because in this case the aspect of a variable cannot be higher in Γ than in Γ_1, Δ_1.

It remains to prove the preservation of the heap contents for variables $|\Gamma|_{2,3}$ (statement P). First consider which of these variables play which role in the two sub-evaluations, using elementary statements *1-23* and *a23i1*. Next we will show that these preserved variables are not aliasing with what may be modified by the sub-evaluations. This claim follows almost straightforwardly from *Sep* because the modified variables have aspect 1, but we also need to confirm that the two sets of variables do not overlap. The only danger of overlap is for variables within Δ_1 but the elementary statement *S1-23* shows that no overlap happens for these variables either. Now we can see that the variables $|\Gamma|_{2,3}$ are not modified in either of the two evaluations.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

The last danger comes from the fact that variables with aspect 2 in e_1 may share with x, which may be destroyed if $i = 1$. Nevertheless, in this case the variables that have aspect 2 in e_1 have the aspect raised to 1 in Γ and are therefore not obliged to be preserved (which is implicit in *a23i1*). This concludes the proof of P, which, as stated earlier, concludes the proof of the theorem for let expressions. □

5.3 Soundness theorem

Now we are ready to complete Preliminary Theorem 5.1 by interpreting its assumption H3 as $S, \sigma \Vdash^{\infty}_{\Gamma} \eta$ and its conclusions as $v, \sigma' \Vdash^{\infty}_{A} a$:

Theorem 5.8 (*Soundness*)
Assume the following data and conditions:

H1. a well-typed program P over some signature Σ;
H2. a well-typed term $\Gamma \vdash e : A$ over Σ for some Γ, e, A; and
H3. a heap σ, a stack S and a valuation η such that $S, \sigma \Vdash^{\infty}_{\Gamma} \eta$.

Then we have also:

C1. $S, \sigma \vdash e \rightsquigarrow v, \sigma'$ implies that there is a such that $\eta \vdash e \rightsquigarrow_{\mathsf{SF}} a$ and $v, \sigma' \Vdash^{\infty}_{A} a$ (i.e. in-place update evaluation is correct);
C2. $\eta \vdash e \rightsquigarrow_{\mathsf{SF}} a$ implies that there are v and σ' such that $S, \sigma \vdash e \rightsquigarrow v, \sigma'$, and $v, \sigma' \Vdash^{\infty}_{A} a$
(i.e. in-place update evaluation is complete).

Proof
Conclusion C1 is obtained straightforwardly by adapting the separation theorem (Theorem 5.4) and its proof to include semantic values alongside the separating heap representation statements, e.g. $v, \sigma' \Vdash^{\infty}_{A} a$ instead of $v, \sigma' \Vdash^{\infty}_{A}$. We thus consider conclusion C1 proved.

The assumptions of this theorem for proving conclusion C2 are almost identical to the assumptions of the separation theorem. The only difference is in switching from ordinary operational semantics to safe operational semantics. We will, therefore, be able to reuse the structure and certain parts of the proof of the separation theorem.

Like in the separation theorem, the proof of C2 is by induction simultaneously on the derivation of the operational semantics and on the derivation of the typing judgement. The rules for safe operational semantics are in an obvious one-to-one correspondence with the rules for in-place updating operational semantics and the corresponding rules are quite similar.

Whenever we need to use the induction hypothesis, one of the most laborious steps is to prove assumption H3 for the sub-expression. Fortunately, this step can always be reused from the proof of the separation theorem without any modifications. We will therefore omit this step in our proof description. The reason why we can reuse the proof of H3 for sub-expressions from the separation theorem is that the proofs made no use of the operational semantics. In cases where there are two sub-expressions, when proving H3 for the second one, the conclusions of the separation theorem are needed for the first sub-expression. This can be obtained here by the separation theorem applied on the first sub-expression with the missing operational semantics condition (H4 in Theorem 5.4) supplied by the first induction hypothesis.

CONST: In the ordinary operational semantics, we get $S, \sigma \vdash c \rightsquigarrow c, \sigma$, trivially proving the conclusion.

VAR: This case is very similar to CONST except that we need to realise that $S(x), \sigma \Vdash^{\infty}_{\Gamma} \eta(x)$ by H4 to prove that the value is equivalent.

APP, IF-TRUE, IF-FALSE, DROP, RAISE: We can apply the induction hypothesis with an updated typing judgement and operational rule but the same S, η and a to give us the desired conclusions verbatim.

WEAK: Assume $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma \vdash e : A$ can be derived from a valid judgement $\Gamma_1 \vdash e : A$. We can apply the induction hypothesis on the same assumptions except for replacing Γ with Γ_1, S with $S_1 = S|_{\Gamma_1}$ and η with $\eta_1 = \eta|_{\Gamma_1}$. In the conclusions that we get, we can now substitute back S for S_1 because extending the stack in operational semantics does not invalidate it.

$\otimes$-PAIR, NIL, LEAF **and** NODE: Conclusions follow directly from the assumptions, like in the case of VAR and CONST.

CONS: Aligning the CONS rules for safe and in-place updating operational semantics allows us to draw conclusion C1, provided we can show that $S(x_d)$ is not in $R_{\Gamma(x_h)}(S(x_h), \sigma)$ and $R_{\Gamma(x_t)}(S(x_t), \sigma)$. This is guaranteed by condition H3 and the fact that x_d has aspect 1.

PAIR-ELIM: We apply the induction hypothesis on the only operational premise, substituting $\Gamma, x{:}^{i} A_1 \otimes A_2$ with $\Gamma, x_1{:}^{i_1} A_1, x_2{:}^{i_2} A_2$ and η with $\eta[x_1 \mapsto a_1, x_2 \mapsto a_2]$. We get the desired conclusions except for having to apply the imperative operational rule to turn $S[x_1 \mapsto v_1, x_2 \mapsto v_2]$ back to S. The remainder of the induction hypothesis is identical to the desired statement.

LET: We can trivially apply the induction hypothesis on the first operational premise and also on the second premise, thanks to the ability to reuse the proof from Theorem 5.4 to get H3 for it. These hypotheses yield the two premises of the imperative operational rule $\times$-PAIR, which then derives the operational statement in conclusion C1. The representation of the final value follows directly from the second induction hypothesis.

$\times$-PAIR: This is very similar to LET, except that we need to work harder to prove that the final value is equivalent to the semantic value.

To obtain $(v_1, v_2), \sigma'' \Vdash^{\infty}_{A_1 \times A_2} (a_1, a_2)$, we need $v_1, \sigma'' \Vdash^{\infty}_{A_1} a_1$, which follows from $v_1, \sigma' \Vdash^{\infty}_{A_1} a_1$, and the fact that evaluation of e_2 does not modify the region of v_1, which follows from

- conclusion C2 of the separation theorem applied on the second premise;
- conclusion C1 of the separation theorem applied on the first premise; and
- the side condition of $\times$-PAIR. □

6 Conclusions and related work

In this paper we introduced a typing scheme that relaxes linear or affine typing schemes, allowing some cases of contraction. Variables can be used more than

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

once in certain points in program expressions. The points of re-use are determined by three *usage aspects* that characterise variable usages as *destructive* (aspect 1), *read-only shared* (aspect 2) and *read-only unshared* (aspect 3). Aspect 2 is the most interesting and novel: it captures when the machine representations of function arguments and result may share memory (i.e. exhibit aliasing). It allows the aliasing to be tracked through expressions.

Our typing scheme was demonstrated in a version of the resource-controlled linear functional programming language LFPL (Hofmann 2000), which has an affine typing scheme that additionally distinguishes between heap-allocated data types and register or stack-allocated ones. These “heap-free” types are always given aspect 3 here. LFPL has an intended implementation using an in-place update interpretation for data-type construction; the resource type $\diamond$ is used to represent a unit of heap space that can be destructively updated to construct a cell of a data structure. With usage aspects, a data-type constructor typing always attaches aspect 1 to its resource argument.

We gave an operational semantics to formalise a machine model of the in-place update interpretation of LFPL. We proved that usage aspect typing ensures that evaluation in the language is both type sound for a memory model and equivalent to the standard call-by-value semantics, which corresponds to the usual denotational functional semantics. The denotational correctness establishes that the program is always properly evaluated; it justifies reasoning about programs using their functional semantics, without caring about the underlying imperative behaviour. The proof is highly non-trivial because of the semantics of aspects in contexts, especially the need to establish separation and preservation properties between memory areas occupied by variables in the context and the results of expression evaluation. This suggests that a simple usage aspect typing can express a rather complex aliasing and separation relation. We believe that the result here goes beyond most existing proofs of type safety in the literature for related type systems.

6.1 Related work

The closest strand of work begins with Wadler’s introduction of the idea of a sequential let (Wadler 1990). If we assume that e_1 is evaluated before e_2 in the expression

$$\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2,$$

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 the type of e_1 is such that its value x cannot possibly be aliased with z. This condition is strictly stronger than z having aspect 3 in our system: e.g. our example term $S2$ in Subsection 2.4 would not type-check translated into Wadler’s language in a straightforward way:

```
let! (m)l'=filter_similar(m,l) in map_reverse(cons(m,l'))
```

because the type of `l'`, i.e. $\mathsf{L}(\mathsf{L}(\mathsf{N}))$ is potentially sharing with the type of `m`, i.e. $\mathsf{L}(\mathsf{N})$.

Odersky (1992) improved Wadler's system by introducing *observer* type annotations, which semantically correspond to our aspect 3: example $S2$ works in his system. In certain ways, Odersky's system is more powerful than ours: it supports polymorphism, higher order functions and observer status can be independently granted to individual components of an argument's type rather than the argument as a whole. Nevertheless, the observer annotations cannot model aspect 2, which can be demonstrated with our example $S1$. In Odersky's system, when typing the term

```
    let! b = find_best(l) in first_similar(b,l)
```

the inner list within the type of `l`, i.e. L(L(N)), will have to be non-linear because it cannot be made an observer in the first term where it shares with the result `b`.

Kobayashi (1999) introduces *quasi-linear* types. This typing scheme also allows sharing in let expressions. It has a δ *use*, which corresponds roughly to our aspect 3 usage. Kobayashi's motivation was to detect statically points where deallocation occurs; this requires stack-managed extra heap, augmenting region analysis (Tofte & Talpin 1997). To illustrate Kobayashi's system, we quote the typing of the append function from Kobayashi (1998, p. 30):

$$\forall\alpha,\beta,\gamma,i,j,k :: \{\alpha \geqslant \gamma, \beta \geqslant \gamma, i \geqslant \delta, j \geqslant k, j \geqslant \delta, k \geqslant \lceil k \rceil, l \geqslant \delta\}\,.$$
$$((\alpha\ \mathit{list}^{i} \times^{l} \beta\ \mathit{list}^{j}) \rightarrow^{\omega,\omega} \gamma\ \mathit{list}^{k})$$

The judgement entails that the cons cells of both input lists have been used but are not bound to the result ($i \geqslant \delta, j \geqslant \delta$) and thus can be safely deallocated and re-used. In particular, it is safe to implement the append in-place. Also, the judgement says that the elements of the lists are not touched ($\alpha \geqslant \gamma, \beta \geqslant \gamma$) and there are no restrictions on how many times the function may be safely used ($\rightarrow^{\omega,\omega}$).

The example typing shows that our usage aspect annotation is considerably more concise than Kobayashi's typings, although we assume a more restricted scenario where an in-place update interpretation is intended rather than inferred. 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.

To summarise so far, in the realm of functional programming, the work that is related to ours mostly encompasses our usage aspect 3, but does not have anything analogous to our aspect 2. There is also a long history of work on aliasing in the realm of imperative programming, most lately introducing expressive type systems and specialised logics, which we outline next. Although type systems in this setting may appear to be closely comparable to ours, there are important differences. For example, it seems hard to concisely represent aspect 2 in an imperative setting: an imperative procedure may have many non-local effects during its execution that could introduce aliasing at almost arbitrary positions in the store, leading to the

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

need to introduce names for locations (and then polymorphism over those names). Things are more controlled in a functional setting, since a function returns a single result, so we can express the aspect 2 concisely as aliasing with this single result and monitor the effect of the introduced aliasing more easily.

Some formalisms for reasoning about aliasing in imperative programs are flexible enough to express and prove safe an aliasing corresponding to our aspect 2. Probably the closest of these to our system are *fractional permissions* (Boyland 2003), which track all aliases of a linear value across procedures (and even parallel threads) and statically detect when the value becomes unique and can be safely overwritten. The tool for analysing C programs described in Evans (1996) has several annotations whose meaning is related to our aspects. Although this system has been popular and useful for finding errors, its theoretical underpinning is limited and its guarantees are therefore much weaker than those of the other systems mentioned. Also, separation logic (Reynolds 2002) and alias types (Smith *et al.* 2000) can be used to express and verify safety of aliasing across procedures.

In separation logic and the logic of bunched implications (Ishtiaq & O'Hearn 2001), it is natural to express aliasing properties but not a read-only behaviour. This means that without amendments neither aspect 2 or 3 can be expressed. One way to express read-only behaviour that has been suggested is to combine these logics with passive types from syntactic control of interference (Reynolds 1978; O'Hearn *et al.* 1995). Another way to express read-only properties in separation logic is by the use of auxiliary variables that provide a link between the heap before and after the execution of a command. But again, such systems provide much more complicated assertions than ours, and need sophisticated machinery to reason with them.

Separation logic and alias types are complex general mechanisms in which automatic checking of higher-level type safety (such as aliasing over values of recursive data types) is difficult to obtain. Some works close to alias types (e.g. Fahndrich & DeLine 2002; Aiken *et al.* 2003) and the work by Boyland already mentioned do make steps in this direction.

All the mentioned methods for analysing memory aliasing in imperative programmes make assertions about individual memory cells or unstructured heap blocks. Defining recursive data types and expressing aliasing properties for them as a whole is not straightforward. It is apparent that, e.g. for alias types (Walker & Morrisett 2001), the resulting assertions are complex and it will be more difficult to infer them than to infer our usage aspects.

We would like to stress again that imperative languages do not admit an intuitive denotational semantics comparable with that of our functional language. Consequently, it does not make as much sense to prove the correctness of the execution of well-typed programmes in the quoted systems as much as it does in ours. Indeed, the mentioned works prove only type safety of the execution, not relating the behaviour of the programmes to their specifications. Our approach gives the programmer a transparent denotational semantics that can be used to check that programmes meet their specifications. Reasoning at this level, usage aspects and diamond values can be completely ignored; our proof guarantees that the imperative interpretation with aliasing is sound. Diamond values or other means of explicit

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

memory deallocation or reuse allow the programmer to safely optimise the memory usage of their programs without changing its functional behaviour.

There is further less-closely related work on formal systems for reasoning or type-checking in the presence of aliasing, including, e.g. work on the imperative λ-calculus (Yang & Reddy 1997), uniqueness types (Barendsen & Smetsers 1996), usage types for optimised compilation of lazy functional programmes (Peyton Jones & Wansbrough 2000) and program analyses for destructive array updates (Draghicescu & Purushothaman 1993; Wand & Clinger 1998) as automated in PVS (Shankar 1999). 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 (Wilhelm *et al.* 2000), already mentioned alias types (Smith *et al.* 2000), and static capabilities (Crary *et al.* 1999), 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 optimisations and typed low-level languages in the hope that we can *guarantee* that certain optimisations will always be possible in LFPL by virtue of its type system. This is in contrast to the behaviour of many present optimising compilers where it is often difficult for the programmer to be sure if a certain desirable optimisation will be performed by the compiler or not. Work in this directions has begun in Aspinall & Compagnoni (2003) where a typed assembly language is developed that has high-level types designed to support compilation from LFPL to obviate the need for garbage collection.

6.2 Outlook

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. We do not expect significant problems integrating Milner-style polymorphism into our language and higher-order functions should be possible using the standard types and effects technique (Gifford & Lucassen 1986; Talpin & Jouvelot 1994).

Another step is to consider inference mechanisms for adding resource annotations, including the $\diamond$ arguments (we mentioned some progress on such inference in Section 1) and usage aspects, as well as the possibility of automatically choosing between $\otimes$-types and $\times$-types. Inference of our usage aspects has been addressed in Konečný (2003b) and a generalisation of the inference of the product types is included in Konečný (2003a) where usage aspects are assigned to certain types sub-terms instead of variables.

In Konečný (2003b), the third author showed that for every program typable in our system, there is a typing with best (i.e. largest) usage aspects. These usage aspects can be automatically reconstructed using an iterative search for a fixed point,

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

starting with the most optimistic typing of all functions in the program, i.e. assuming that every argument in every function is used with aspect 3.

We have supported our theoretical work with the development of experimental prototype compilers for LFPL. Type-checking and usage aspect inference for the present system have been implemented by the third author as a front-end to a compiler by Robert Atkey. More information about the compiler can be found via Aspinall & Konečný (2003).

Acknowledgements

This work was supported by the EPSRC grant GR/N28436/01. The authors are grateful to Robert Atkey for discussion and comments and indebted to the referees who provided useful suggestions for improvement. The authors owe a number of good ideas to them. Especially they would like to thank the referee who suggested the use of safe operational semantics to simplify the presentation and proof of soundness. An earlier version of this work was reported as Aspinall & Hofmann (2002). The details of the soundness proof reported there turned out to be considerably more intricate than first anticipated; they were provided by the third author in this paper.

References

Wikipedia article on the administrative normal form. (2007). `http://en.wikipedia.org/wiki/Administrative_Normal_Form`. Accessed March 27, 2007.

Aiken, A., Foster, J. S., Kodumal, J. & Terauchi, T. (2003). Checking and inferring local non-aliasing. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* New York: ACM Press, pp. 129–140.

Aspinall, D. & Compagnoni, A. (2003). Heap-bounded assembly language. *J. Automated Reason.* **31**(3/4), 261–302.

Aspinall, D. & Hofmann, M. (2002). Another type system for in-place update. In *Programming Languages and Systems, Proceedings of 11th European Symposium on Programming*, D. L. Métayer (ed), Springer-Verlag. Lecture Notes in Computer Science 2305.

Aspinall, D. & Konečný, M. (2003, February). *Type Systems for Resource Bounded Programming and Compilation Project Homepage.* `http://homepages.inf.ed.ac.uk/da/resbnd`. Accessed 1 June 2007.

Barendsen, E. & Smetsers, S. (1996). Uniqueness typing for functional languages with graph rewriting semantics. *Math. Struct. Comput. Sci.* **6**, 579–612.

Boyland, J. (2003). Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, R. Cousot (ed), Lecture Notes in Computer Science, vol. 2694. Berlin, Heidelberg, New York: Springer, pp. 55–72.

Crary, K., Walker, D. & Morrisett, G. (1999). Typed memory management in a calculus of capabilities. In *Proceedings ACM Principles ofProgramming Languages*, Kobayashi: Hofmann & Jost, Istiaq & O'Hearn, pp. 262–275.

Dor, N., Rodeh, M. & Sagiv, M. (2000). Checking cleanness in linked lists. In *Proceedings of the Seventh International Static Analysis Symposium.* Springer, Berlin/Heidelberg, pp. 115–134. Lecture Notes in Computer Science 1824.

Draghicescu, M. & Purushothaman, S. (1993). A uniform treatment of order of evaluation and aggregate update. *Theor. Comput. Sci.* **118**(2), 231–262.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

Evans, D. (1996). Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation.* New York: ACM Press, pp. 44–53.

Fahndrich, M. & DeLine, R. (2002). Adoption and focus: Practical linear types for imperative programming. In *Pldi '02: Proceedings of the ACM Sigplan 2002 Conference on Programming Language Design and Implementation.* New York: ACM Press, pp. 13–24.

Gifford, D. K., & Lucassen, J. M. (1986). Integrating functional and imperative programming. *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* New York: ACM Press, pp. 28–38.

Hofmann, M. (2000). A type system for bounded space and functional in-place update. *Nordic J. Comput.*, **7**(4), 258–289. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

Hofmann, M. & Jost, S. (2003). Static prediction of heap space usage for first-order functional programs. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '03)*, New York: ACM Press, pp. 185–197.

Ishtiaq, S. & O'Hearn, P. W. (2001). BI as an assertion language for mutable data structures. *The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, New York: ACM Press, pp. 14–26.

Kobayashi, N. (1998). *Quasi-linear Types.* Tech. rept. 98–02. Department of Information Science, University of Tokyo.

Kobayashi, N. (1999). Quasi-linear types. *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, New York: ACM Press, pp. 29–42.

Konečný, M. (2003a). Functional in-place update with layered datatype sharing. *TLCA 2003, Valencia, Spain, Proceedings.* Springer-Verlag, pp. 195–210. Lecture Notes in Computer Science 2701.

Konečný, M. (2003b). Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen, Proceedings.* Springer-Verlag, pp. 182–199. Lecture Notes in Computer Science 2646.

MacKenzie, K. & Wolverson, N. (2004). Camelot and grail: resource-aware functional programming on the JVM. *Trends in Functional Programing*, Vol. 4. Bristol: Intellect, pp. 29–46.

Odersky, M. (1992). Observers for linear types. *4th European Symposium on Programming (ESOP'92)*, B. Krieg-Brückner (ed), Rennes, France: Springer-Verlag, pp. 390–407. Lecture Notes in Computer Science 582.

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

Peyton, J. S. & Wansbrough, K. (2000, September). Simple usage polymorphism. In *Proc. 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC 2000).* Technical Report CMU–CS–00–161.

Reynolds, J. C. (1978). Syntactic control of interference. *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages (POPL).* Tucson, AZ: ACM Press, pp. 39–46.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. *Proceedings of 17th annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pp. 55–74.

Sabry, A. & Felleisen, M. (1993). Reasoning about programs in continuation-passing style. *LISP and Symbolic Comput.*, **6**(3/4), 289–360.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press

Shankar, N. (1999, November). *Efficiently Executing PVS*. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA.

Smith, F., Walker, D. & Morrisett, G. (2000). Alias types. In: *9th European Symposium on Programming (ESOP'00)*, G. Smolka (ed), Springer-Verlag, pp. 366–381. Lecture Notes in Computer Science 1782.

Talpin, J.-P. & Jouvelot, P. (1994). The type and effect discipline. *Inf. Comput.*, **111**(2), 245–296.

Tofte, M., & Talpin, J.-P. (1997). Region-based memory management. *Inf. Comput.*, **132**(2), 109–176.

Wadler, P. (1990). Linear types can change the world. *IFIP TC 2 Working Conference on Programming Concepts and Methods*, M. Broy & C. B., Jones (eds), Sea of Gallilee, Israel: North-Holland, pp. 561–581.

Walker, D. & Morrisett, J. G. (2001). Alias types for recursive data structures. In *TIC '00: Selected Papers from the Third International Workshop on Types in Compilation*. London, UK: Springer-Verlag, pp. 177–206.

Wand, M. & Clinger, W. D. (1998). Set constraints for destructive array update optimization. In *Proc. IEEE International Conference on Computer Languages (ICCL'98)*, IEEE, pp. 184–193.

Wilhelm, R., Sagiv, M., & Reps, T. (2000). Shape analysis. In *Proc. 9th International Conference on Compiler Construction (CC 2000)*. Springer-Verlag, pp. 1–70. Lecture Notes in Computer Science 1781.

Yang, H., & Reddy, U. (1997). Imperative lambda calculus revisited. Electronic manuscript.

https://doi.org/10.1017/S0956796807006399 Published online by Cambridge University Press