A type system for bounded space and functional in-place update

Martin Hofmann
Division of Informatics
University of Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK
`mxh@dcs.ed.ac.uk`

November 22, 2000

Abstract

We show how linear typing can be used to obtain functional programs which modify heap-allocated data structures in place.

We present this both as a "design pattern" for writing `C`-code in a functional style and as a compilation process from linearly typed first-order functional programs into `malloc()`-free `C` code.

The main technical result is the correctness of this compilation.

The crucial innovation over previous linear typing schemes consists of the introduction of a resource type $\diamond$ which controls the number of constructor symbols such as `cons` in recursive definitions and ensures linear space while restricting expressive power surprisingly little.

While the space efficiency brought about by the new typing scheme and the compilation into C can also be realised by with state-of-the-art optimising compilers for functional languages such as OCAML [16], the present method provides guaranteed bounds on heap space which will be of use for applications such as languages for embedded systems or automatic certification of resource bounds.

We show that the functions expressible in the system are precisely those computable on a linearly space-bounded Turing machine with an unbounded stack. By a result of Cook this equals the complexity class 'exponential time'. A tail recursive fragment of the language captures the complexity class 'linear space'.

We discuss various extensions, in particular an extension with FIFO queues admitting constant time catenation and enqueuing, and an extension of the type system to fully-fledged intuitionistic linear logic.

1 Introduction

In-place modification of heap-allocated data structures such as lists, trees, queues in an imperative language such as `C` is notoriously cumbersome, error prone, and difficult to teach.

Suppose that a type of lists has been defined[1] in `C` by

```
typedef enum {NIL, CONS} kind_t;

typedef struct lnode {
  kind_t kind;
  int hd;
  struct lnode * tl;
} list_t;
```

and that a function

```
list_t reverse(list_t l)
```

should be written which reverses its argument "in place" and returns it. Everyone who has taught `C` will agree that even when recursion is used this is not an entirely trivial task. Similarly, consider a function

```
list_t insert(int a, list_t l)
```

which inserts `a` in the correct position in `l` (assuming that the latter is sorted) allocating one `struct lnode`.

Next, suppose, you want to write a function

```
list_t sort(list_t l)
```

which sorts its argument in place according to the insertion sort algorithm. Note that you cannot use the previously defined function `insert()` here as it allocates new space.

As a final example, assume that we have defined a type of trees

```
typedef struct tnode {
  kind_t kind;
  int label;
  struct tnode * left;
  struct tnode * right;
} tree_t;
```

(with `kind_t` extended with `LEAF, NODE`) and that we want to define a function

```
list_t breadth(tree_t t)
```

which constructs the list of labels of tree `t` in breadth-first order by consuming the space occupied by the tree and allocating at most one extra `struct lnode`. While again, there is no doubt that this can be done, my experience is that all of the above functions are cumbersome to write, difficult to verify, and likely to contain bugs.

Now compare this with the ease with which such functions are written in a functional language such as OCAML [16]. For instance,

[1] Usually, one encodes the empty list as a `NULL`-pointer, whereas here it is encoded as a `list_t` with `kind` component equal to `NIL`. This is more in line with the encoding of trees we present below. If desired, we could go for the slightly more economical encoding, the only price being a loss of genericity.

```
let reverse l = let rec rev_aux l acc =
  match l with
      [] -> acc
  | a::l -> rev_aux l (a::acc)
in rev_aux l []

type tree = Leaf of int
          | Node of int*tree*tree

let rec breadth t = let rec breadth_aux l =
 match l with
  [] -> []
| Leaf(a)::t -> a::breadth_aux(t)
| Node(a,l,r)::t -> a::breadth_aux(t @ [l] @ [r])
in breadth_aux [t]
```

These definitions are written in a couple of minutes and are readily verified using induction and equational reasoning. The difference, of course, is that the functional programs do not modify their argument in place but rather construct the result anew by allocating fresh heap space. If the argument is not needed anymore it will eventually be reclaimed by garbage collection, but we have no guarantee whether and when this will happen. Accordingly, the space usage of a functional program will in general be bigger and less predictable than that of the corresponding `C` program.

The aim of this paper is to show that by imposing mild extra annotations one can have the best of both worlds: easy to write code which is amenable to equational reasoning, yet modifies its arguments in place and does not allocate heap space unless explicitly told to do so. We will describe a linearly[2] typed functional programming language with lists, trees, and other heap-allocated data structure which admits a compilation into `malloc()`-free `C`. This may seem paradoxical at first sight because one should think that at least a few heap allocations would be necessary to generate initial data. However, our type system is such that while it does allow for the definition of *functions* such as the above examples, it does not allow one to define constant terms of heap-allocated type other than trivial ones like nil. If we want to apply these functions to concrete data we either move outside the type system or we introduce an extension which allows for controlled introduction of heap space. However, to develop and verify functions, as opposed to concrete computations, this is largely unnecessary.

This is all made possible in a natural way through the presence of a special resource type $\diamond$ which in fact is the main innovation of the present system over earlier linear type systems, see Section 9.

While experiments with the prototype implementation show that the generated `C`-code can compete with the highly optimised OCAMLOPT native code compiler and outperforms the OCAML run time system by far we believe that the efficient space usage can also be realised by state-of-the-art garbage collection and caching.

[2]We always use "linear" in the sense of "affine linear", i.e. arguments may be used at most once.

The main difference is that we can *prove* that the code generated by our compilation comes with an explicit bound on the heap space used (none at all in the pure system, a controllable amount in an extension with an explicit allocation operator). This will make our system useful in situations where space economy and guaranteed resource bounds are of the essence. Examples are programming languages for embedded systems (see [13] for a survey), smart cards, or "proof-carrying code".

In a nutshell the approach works as follows. The type $\diamond$ (`dia_t` in the `C` examples) gets translated into a pointer type, say `void *` whose values point to heap space of appropriate size to store one list or tree node. It is the task of the type system to maintain the invariant that these pointers point to large enough space which can be overwritten without affecting the result.

When invoking a non-leaf constructor function such as `cons()` or `node()` (but node `nil()` or `leaf()` one must supply an appropriate number of arguments of type $\diamond$ to provide the required heap space. Conversely, if in a function definition an argument of list or tree type is decomposed these $\diamond$-values become available again.

Linear typing then ensures that overwriting the heap space pointed to by these $\diamond$-values is safe.

It is important to realise that the `C` programs obtained as the target of the translation do not involve `malloc()` and therefore must necessarily update their heap allocated arguments in place. Traditional functional programs may achieve the same global space usage by clever garbage collection, but there will be no guarantee that under all circumstances this efficiency will be realised.

We also point out that while the language we present is experimental the examples we can treat are far from trivial: insertion sort, quick sort, breadth first traversal using queues, Huffman's algorithm, and many more. We therefore are lead to believe that with essentially just engineering effort our system could be turned into a usable programming language for the abovementioned applications where resources are restricted.

2 Functional programming with C

Before presenting the language we show what the translated code will look like by way of some direct examples. We point out that in these examples we assume that *every* constructor function uses heap space recuperated from decomposing function arguments. Thus, the functions we define require no additional heap space at all. In practice, allocating fresh heap space will sometimes be unavoidable. We explain later in Section 2.3 on how this can be achieved in a controlled way.

We also clarify that elements of recursive data types will be returned on the stack as a C-`struct` containing a tag field identifying the outermost constructor and fields allowing to access its components. We thus refrain from using the familiar optimisation of representing an empty list as a `NULL` pointer. This is done for the sake of uniformity but could be changed in a practical implementation.

2.1 Lists

For the above-defined list type we would make the following definitions:

```
typedef void * dia_t;
```

and

```
list_t nil(){
  list_t res;
  res.kind=NIL;
  return res;
}
```

and

```
list_t cons(dia_t d, int hd, list_t tl){
  list_t res;
  res.kind = CONS;
  res.hd = hd;
  *(list_t *)d = tl;
  res.tl = (list_t *)d;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  dia_t d;
  int hd;
  list_t tl;
} list_destr_t;
```

and

```
list_destr_t list_destr(list_t l) {
  list_destr_t res;
  res.kind = l.kind;
  if (res.kind == CONS) {
    res.hd = l.hd;
    res.d = (void *) l.tl;
    res.tl = *l.tl;
  }
  return res;
}
```

The function `nil()` simply returns an empty list on the stack. The function `cons()` takes a pointer to free heap space (`d`), an entry (`hd`) and a list (`tl`) and returns on the stack a list with `hd`-field equal to `hd` and `tl`-field pointing to a heap location containing `tl`. This latter heap location is of course the one explicitly provided through the argument `d`.

The destructor function `list_destr()`, finally, takes a list (`l`) and returns a structure containing a field `kind` with value `CONS` iff `l.kind` equals `CONS` and in this case containing in the remaining fields head and tail of `l`, as well as a pointer to a heap location capable of storing a list node (`d`). We will later precisely identify the circumstances under which it is safe to reuse, i.e., overwrite this heap location. Basically, we can do anything we like with the components of the returned structure, but must not anymore refer to the argument of `list_destr`; it has been "destroyed".

Once we have made these definitions we can implement `reverse()` in a functional style as follows:

```
list_t rev_aux(list_t ll, list_t acc) {
  list_destr_t l = list_destr(ll);
  return  l.kind==NIL ? acc
   : rev_aux(l.tl, cons(l.d, l.hd, acc));
}

list_t reverse(list_t l) {
  return rev_aux(l,nil());
}
```

Notice that `reverse()` updates its argument in place, as no call to `malloc()` is being made.

To implement `insert()` we need an extra argument of type `dia_t` since this function, just like `cons()`, increases the length. So we write:

```
list_t insert(dia_t d, int a, list_t ll) {
  list_destr_t l = list_destr(ll);
  return l.kind==NIL ? cons(d,a,nil())
   : a <= l.hd ? cons(d,a,cons(l.d,l.hd,l.tl))
     : cons(d,l.hd,insert(l.d,a,l.tl));
}
```

Using `insert()` we can implement insertion sort with in-place modification as follows:

```
list_t sort(list_t ll) {
  list_destr_t l = list_destr(ll);
  return l.kind==NIL ? nil()
   : insert(l.d,l.hd,sort(l.tl));
}
```

Notice, how the value `l.d` which becomes available in decomposing `l` is used to feed the `insert()` function.

One might object against the implementation of the resource type `dia_t` as a `void` pointer and require it to be a pointer to a list node instead to achieve extra type safety. This would, however, preclude the use of recovered heap space from decomposing a list in order to construct elements of another datatype, e.g., a binary tree, as we will now show. Type safety is achieved by manipulating elements of type `dia_t` only through the well-defined interface provided by the constructor and destructor functions.

2.2 Trees

Let us now look at binary `int`-labelled trees. We define

```
tree_t leaf(int label) {
  tree_t res;
  res.kind = LEAF;
  res.label = label;
  return res;
}
```

and

```
tree_t node(dia_t d1, dia_t d2,
    int label, tree_t l, tree_t r) {
  tree_t res;
  res.kind = NODE;
  res.label = label;
  *(tree_t *)d1 = l;
  *(tree_t *)d2 = r;
  res.left = (tree_t *)d1;
  res.right = (tree_t *)d2;
  return res;
}
```

followed by

```
typedef struct {              and    tree_destr_t tree_destr(tree_t t) {
  kind_t kind;                         tree_destr_t res;
  int label;                           res.label = t.label;
  dia_t d1, d2;                        if((res.kind = t.kind) == NODE) {
  tree_t left, right;                    res.d1 = (dia_t)t.left;
} tree_destr_t;                          res.d2 = (dia_t)t.right;
                                         res.left = *(tree_t *)t.left;
                                         res.right = *(tree_t *)t.right;
                                       }
                                       return res;
                                     }
```

Notice that we must "pay" *two* ◇s in order to build a tree node. In exchange, two ◇s become available when we decompose a tree.

To implement the above-mentioned function `breadth` we have to define a type `listtree_t` of lists of trees analogous to `list_t` with `int` replaced by `tree_t`. Of course, the associated helper functions need to get distinct names such as `niltree()`, etc.

We can then define a function `br_aux`

```
list_t br_aux(listtree_t l){
  listtree_destr_t l = listtree_destr(l0);
  tree_destr_t t;

  if(l.kind==NIL)
    return nil();
  else /*l.kind==CONS*/
    {
      t = tree_destr(l.hd);
      if (t.kind == LEAF)
        return cons(l.d,t.label,br_aux(l.tl));
      else /*t. kind == NODE*/
        return cons(l.d,t.label,br_aux(snoc(t.d1,snoc(t.d2,l.tl,t.left),t.right)));
    }
}
```

where we have used the following helper function:

```
listtree_t snoc(dia_t d,listtree_t l0,tree_t t){
  listtree_destr_t l = listtree_destr(l0);
  if (l.kind==NIL)
    return constree(d,t,niltree());
  else
    return constree(l.d,l.hd,snoc(d,l.tl,t));
```

which appends an element at the end of a list. This function uses linear time (and stack size!) which is unfortunate; we show below in Section 6 how to incorporate a queue type admitting constant time appending.

We now obtain the desired function `breadth` as

```
list_t breadth(dia_t d, tree_t t) {
  return br_aux(cons(d,t,nil()));
}
```

Notice that the type of `breadth` shows that the result requires one memory region more than the input provides.

We also remark that for the functioning of this example it was crucial that we could rededicate "diamonds" stemming from a tree to be used for the construction of a list. Conversely, in algorithms such as tree sort (which sorts a list by turning it into a binary search tree which is subsequently flattened in depth first-order) we must use "diamonds" obtained by deconstructing a list in order to construct a tree. For this to function correctly, we must assume that the layout of the argument list was sufficiently "spreaded out", i.e., that each pointer in the list points to an otherwise unused region large enough to hold a tree node (as opposed to merely a list node). For this, and other reasons it is convenient to have a function as described in the next subsection which provides fresh heap space in such portions.

One may object against the inherent space wastage of this policy. In situations where it becomes unacceptable one might be lead to introduce several kinds of "diamonds" which are mutually incompatible or can be "traded" against each other at fixed "exchange rates", e.g. in our example four "list diamonds" for three "tree diamonds". We notice though, that at any time this wastage is within a constant multiple of the size of the data so that the advantages of having only one resource type may often outweigh the wastage.

2.3 Allocating fresh space

All these functions do not use dynamic memory allocation because the heap space needed to store the result can be taken from the argument. To construct concrete lists in the first place we need of course dynamic memory allocation. This can be done in a controlled way using a function `dia_t new()` which allocates heap space in portions of one "diamond":

```
dia_t new() {
  return (dia_t) malloc(sizeof(max_t));
}
```

where `max_t` is a `union` of all the datatypes used in the program. This allows us to write an input function

```
list_t getlist() {
  int x;
  scanf("%d", &x);
  return x==-1 ? nil()
   : cons(new(),x,getlist());
}
```

Of course, for these programs to be correct it is crucial that we do not overwrite heap space which is still in use. The main message of this paper is that this can be guaranteed systematically by adhering to a linear typing discipline. In other words, a function can use its argument at most once.

For instance, the following code which attempts to double the size of its argument will be ruled out:

```
list_t twice(list_t ll) {
  list_destr_t l = list_destr(ll);

  return l.kind==NIL ? nil()
   : cons(l.d,0,(cons(l.d,0,twice(l.tl))));
}
```

It is rightly ruled out because rather than returning a list of 0's twice the size of its input it returns a circular list! A similar effect happens, if we replace the last line of the code for `insert()` by

```
      cons(d,l.hd,insert(d,a,l.tl));
```

In each case the reason is the double usage of the $\diamond$-values `d` and `l.d`.

3 A linear functional programming language

We will now introduce a linearly typed functional language (to be called LFPL for reference) and translate it systematically into `C`. This will be done with the following aims. First, it allows us to formally prove the correctness of the methodology sketched above, second it will relieve us from having to rewrite similar code many times. Suppose, for instance, you wanted to use lists of trees (as needed to implement breadth first search). Then all the basic list code (`list_t, nil(), cons()`, etc.) will have to be rewritten (this problem could presumably also be overcome through the use of `C`++ templates [14]). Thirdly, a formalised language with linear type system will allow us to enforce the usage restrictions on which the correctness of the above code relies. Finally, this will open up the possibility to extend the language to a fully-fledged functional language which would be partly compiled into `C` whenever this is possible and executed in the traditional functional way when this is not the case.

3.1 Syntax and typing rules

The terms of LFPL are given by the following grammar:

$e ::=$	x	(variable)
$\mid$	$f(e_1, \ldots, e_n)$	function application
$\mid$	c	integer constant
$\mid$	$e_1 \star e_2$	infix op., $\star \in \{+, -, \times, =, \leq\ \ldots\}$
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$	conditional
$\mid$	$\mathsf{inl}(e)$	left injection
$\mid$	$\mathsf{inr}(e)$	right injection
$\mid$	$e_1 \otimes e_2$	pairing
$\mid$	nil	empty list
$\mid$	$\mathsf{cons}(e_1, e_2, e_3)$	cons with res. arg.
$\mid$	$\mathsf{leaf}(e)$	leaf constructor
$\mid$	$\mathsf{node}(e_1, e_2, e_3, e_4, e_5)$	node constr. w. two res. args.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ nil{\Rightarrow}e_2 \mid \mathsf{cons}(d, h, t){\Rightarrow}e_3$	list elimination
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{leaf}(a){\Rightarrow}e_2 \mid$ $\mathsf{node}(d_1, d_2, a, l, r){\Rightarrow}e_3$	tree elim.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ x \otimes y{\Rightarrow}e_2$	pair elim.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{inl}(x){\Rightarrow}e_2 barr \mathsf{inr}(x){\Rightarrow}e_3$	sum elim.

Intuitively, a program consists of a series of function definitions of the form $f(x_1, \ldots, x_n) = e$. The meaning of a program depends, however, on its being well-typed which is why we need to consider types now.

The *zero-order types* are given by the following grammar.

$$A ::= \mathsf{N} \mid \Diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A_1 \otimes A_2 \mid A_1 + A_2$$

Here $\Diamond$ denotes the resource type, $\mathsf{L}(A)$ and $\mathsf{T}(A)$ stand for lists and trees with entries of type A, respectively; $A_1 \otimes A_2$ is the type of pairs with first component of type A_1 and second component of type A_2. The type $A_1 + A_2$, finally, is the disjoint union of A_1 and A_2.

A first-order type is an expression of the form $T = (A_1, \ldots, A_n) \to B$ where $A_1 \ldots A_n$ and B are zero-order types.

A signature Σ is a finite function from identifiers (thought of as *function symbols*) to first-order types.

A *typing context* Γ is a finite function from identifiers (thought of as parameters) to zero order types; if $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x{:}A$ for the extension of Γ with $x \mapsto A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise or conclusion of a rule below it is implicitly understood that these disjointness conditions are met. We write $e[x/y]$ for the term obtained from e by replacing all occurrences of the free variable y in e by x. We consider terms modulo renaming of bound variables.

Types not including $\mathsf{L}(-), \mathsf{T}(-), \diamond$ are called *heap-free*, e.g. N and $\mathsf{N} \otimes \mathsf{N}$ are heap-free.

Let Σ be a signature. The *typing judgement* $\Gamma \vdash_\Sigma e : A$ read "expression e has type A in typing context Γ and signature Σ" is defined by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \tag{Var}$$

$$\frac{\Sigma(f) = (A_1, \ldots, A_n) \rightarrow B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1 \ldots n}{\Gamma_1, \ldots, \Gamma_n \vdash_\Sigma f(e_1, \ldots, e_n) : B} \tag{Sig}$$

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e : B \qquad A \text{ heap-free}}{\Gamma, x{:}A \vdash_\Sigma e[x/y] : B} \tag{Contr}$$

$$\frac{c \text{ a } \mathtt{C} \text{ integer constant}}{\Gamma \vdash_\Sigma c : \mathsf{N}} \tag{Const}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{N} \qquad \Delta \vdash_\Sigma e_2 : \mathsf{N} \qquad \star \text{ a } \mathtt{C} \text{ infix opn.}}{\Gamma, \Delta \vdash e_1 \star e_2 : \mathsf{N}} \tag{Infix}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{N} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \tag{If}$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \tag{Pair}$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}A, y{:}B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e' : C} \tag{Split}$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{inl}(e) : A + B} \tag{Inl}$$

$$\frac{\Gamma \vdash_\Sigma e : B}{\Gamma \vdash_\Sigma \mathsf{inr}(e) : A + B} \tag{Inr}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e : A + B \\ \Delta, x{:}A \vdash_\Sigma e' : C \qquad \Delta, x{:}B \vdash_\Sigma e'' : C\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow e' | \mathsf{inr}(x) \Rightarrow e'' : C} \tag{Sum-Elim}$$

$$\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A) \qquad \text{(NIL)}$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \qquad \text{(CONS)}$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_{\mathsf{nil}} | \mathsf{cons}(d,h,t){\Rightarrow}e_{\mathsf{cons}} : B} \qquad \text{(LIST-ELIM)}$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{leaf}(e) : \mathsf{T}(A)} \qquad \text{(LEAF)}$$

$$\frac{\begin{array}{c}\Gamma_{d1} \vdash_\Sigma e_{d1} : \diamond \qquad \Gamma_{d2} \vdash_\Sigma e_{d2} : \diamond \qquad \Gamma_a \vdash_\Sigma e_a : A \\ \Gamma_l \vdash_\Sigma e_l : \mathsf{T}(A) \qquad \Gamma_r \vdash_\Sigma e_r : \mathsf{T}(A)\end{array}}{\Gamma_{d1}, \Gamma_{d2}, \Gamma_a, \Gamma_l, \Gamma_r \vdash_\Sigma \mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r) : \mathsf{T}(A)} \qquad \text{(NODE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e : \mathsf{T}(A) \qquad \Delta, a{:}A \vdash_\Sigma e_{\mathsf{leaf}} : B \\ \Delta, d_1{:}\diamond, d_2{:}\diamond, a{:}A, l{:}\mathsf{T}(A), r{:}\mathsf{T}(A) \vdash_\Sigma e_{\mathsf{node}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a){\Rightarrow}e_{\mathsf{leaf}} | \mathsf{node}(d_1, d_2, a, l, r){\Rightarrow}e_{\mathsf{node}} : B} \qquad \text{(TREE-ELIM)}$$

Remarks The symbol $\star$ in rule INFIX ranges over a set of binary infix operations such as `+, - ,/ , *, <=, ==,` ... We may include more such operations and also other base types such as floating point numbers or characters.

The constructs involving match bind variables.

Application of function symbols or operations to their operands is linear in the sense that several operands must in general not share common free variables. This is because of the implicit side condition on juxtaposition of contexts mentioned above. In view of rule CONTR, however, variables of a heap-free type may be shared and moreover the same free variable may appear in different branches of a case distinction as follows e.g. from the form of rule IF. Here is how we typecheck $x + x$ when $x{:}\mathsf{N}$. First, we have $x{:}\mathsf{N} \vdash x : \mathsf{N}$ and $y{:}\mathsf{N} \vdash y : \mathsf{N}$ by VAR. Then $x{:}\mathsf{N}, y{:}\mathsf{N} \vdash x{+}y : \mathsf{N}$ by INFIX and finally $x{:}\mathsf{N} \vdash x{+}x : \mathsf{N}$ by rule CONTR. It follows by standard type-theoretic techniques that typechecking for this system is decidable in linear time. More precisely, we have a linear time computable function which given a context Γ, a term e, and a type A either returns a minimal subcontext Δ of Γ such that $\Delta \vdash e : A$ or returns "failure" in the case where no such $\Gamma \vdash e : A$ does not hold. This function can be defined by primitive recursion over e.

The restriction that only variables of heap-free type may be shared may seem overly restrictive in view of the fact that not all accesses to heap-allocated data actually modifies it. In future work we will propose a refinement of the above type system which distinguishes between destructive and “read-only” uses of a variable. The issues involved are, however, orthogonal to the ones studied in this paper and are sufficiently complicated to warrant a separate publication.

For the moment we can comfort the worried reader that in many cases the effect of such a read-only typing discipline can be simulated by returning the argument. For example, rather than writing, e.g., a function $f : (\mathsf{L}(\mathsf{N}))\to\mathsf{N}$ which returns the first element of an integer list (and 0 when applied to the empty list) we would write a function $g : (\mathsf{L}(\mathsf{N}))\to\mathsf{N}\otimes\mathsf{L}(\mathsf{N})$ which return both the first element and the list itself for subsequent use.

Programs A *program* consists of a signature Σ and for each symbol

$$f : (A_1, \dots, A_n)\to B$$

contained in Σ a term e_f such that

$$x_1{:}A_1, \dots, x_n{:}A_n \vdash_\Sigma e_f : B$$

3.2 Set-theoretic interpretation

In order to specify the purely functional meaning of programs we introduce a set-theoretic interpretation as follows: types are interpreted as sets by

$$\begin{aligned}
&[\![\mathsf{N}]\!] = \mathbf{Z}\\
&[\![\Diamond]\!] = \{0\}\\
&[\![\mathsf{L}(A)]\!] = \text{finite lists over } [\![A]\!]\\
&[\![\mathsf{T}(A)]\!] = \text{binary } [\![A]\!]\text{-labelled trees}\\
&[\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!]\\
&[\![A + B]\!] = \{\mathsf{inl}(a) \mid a \in [\![A]\!]\} \cup \{\mathsf{inr}(b) \mid b \in [\![B]\!]\}
\end{aligned}$$

To each program $(\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$ we can now associate a mapping ρ such that $\rho(f)$ is a *partial* function from $[\![A_1]\!] \times \dots [\![A_n]\!]$ to $[\![B]\!]$ for each $f : (A_1, \dots, A_n)\to B$.

This meaning is given in the standard fashion as the least fixpoint of an appropriate compositionally defined operator, as follows.

A *valuation* of a context Γ is a function η such that $\eta(x) \in [\![\Gamma(x)]\!]$ for each $x \in \mathrm{dom}(\Gamma)$; a valuation of a signature Σ is a function ρ such that $\rho(f) \in [\![\Sigma(f)]\!]$ whenever $f \in \mathrm{dom}(\Sigma)$.

To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ in the obvious way, i.e. function symbols and variables are interpreted according to the valuations; basic functions and expression formers are interpreted by the eponymous set-theoretic operations, ignoring the arguments of type $\Diamond$ in the case of constructor functions. The formal definition of $[\![-]\!]_{\eta,\rho}$ is by induction on terms.

Here are a few representative clauses.

$$
\begin{aligned}
&[\![x]\!]_{\eta,\rho} = \eta(x)\\
&[\![f(e_1,\dots,e_n)]\!]_{\eta,\rho} = \rho(f)([\![e_1]\!]_{\eta,\rho},\dots,[\![e_n]\!]_{\eta,\rho})\\
&[\![\mathsf{cons}(e_1,e_2,e_3)]\!]_{\eta,\rho} = [\![e_2]\!]_{\eta,\rho} :: [\![e_3]\!]_{\eta,\rho}\\
&[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(x)\Rightarrow e_1|\mathsf{node}(d_1,d_2,x,l,r)\Rightarrow e_2]\!]_{\eta,\rho}\\
&\quad = [\![e_1]\!]_{\eta[x\mapsto a],\rho}\\
&\quad \text{when } [\![e]\!]_{\eta,\rho} = \mathrm{leaf}(a) \text{ and}\\
&\quad = [\![e_2]\!]_{\eta[d_1\mapsto 0,d_2\mapsto 0,x\mapsto a,l\mapsto u,r\mapsto v],\rho}\\
&\quad \text{when } [\![e]\!]_{\eta,\rho} = \mathrm{node}(a,u,v)
\end{aligned}
$$

A *program* $(\Sigma,(e_f)_{f\in\mathrm{dom}(\Sigma)})$ is interpreted as the least valuation ρ such that

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

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

We stress that this set-theoretic semantics does not say anything about space usage. Its *only* purpose is to pin down the functional denotations of programs so that we can formally state what it means to implement a function. Accordingly, the resource type is interpreted as a singleton set and $\otimes$ product is interpreted as cartesian product.

It will be our task to show that the `malloc()`-free interpretation of LFPL is faithful with respect to the set-theoretic semantics. Once this is done, the user of the language can think entirely in terms of the semantics as far as extensional verification and development of programs is concerned. In addition, he or she can benefit from the resource bounds obtained from the interpretation but need not worry about how these are guaranteed.

3.3 Examples

Reverse:

$$
\begin{aligned}
&\texttt{rev\_aux} : (\mathsf{L}(\mathsf{N}),\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})\\
&\texttt{reverse} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})\\
&e_{\texttt{rev\_aux}}(l,acc) = \mathsf{match}\ l\ \mathsf{with}\\
&\qquad \mathsf{nil}\Rightarrow acc\\
&\qquad |\mathsf{cons}(d,h,t)\Rightarrow\texttt{rev\_aux}(t,\mathsf{cons}(d,h,acc))\\
&e_{\texttt{reverse}}(l) = \texttt{rev\_aux}(l,\mathsf{nil})
\end{aligned}
$$

Insertion sort

$$
\begin{array}{l}
\texttt{insert} : (\diamond, \mathsf{N}, \mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})\\
\texttt{sort} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})\\
e_{\texttt{insert}}(d,a,l) = \mathsf{match}\ l\ \mathsf{with}\\
\quad\ \ \mathsf{nil}\Rightarrow\mathsf{nil}\\
\quad\ \ |\mathsf{cons}(d',b,t)\Rightarrow\mathsf{if}\ a\le b\\
\quad\quad\ \ \mathsf{then}\ \mathsf{cons}(d,a,\mathsf{cons}(d',b,t))\\
\quad\quad\ \ \mathsf{else}\ \mathsf{cons}(d,b,\texttt{insert}(d',a,t))\\
e_{\texttt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with}\\
\quad\ \ \mathsf{nil}\Rightarrow\mathsf{nil}\\
\quad\ \ |\mathsf{cons}(d,a,t)\Rightarrow\texttt{insert}(d,a,\texttt{sort}(t))
\end{array}
$$

Breadth-first search

$$
\begin{array}{l}
\texttt{snoc} : (\diamond, \mathsf{L}(\mathsf{T}(\mathsf{N})), \mathsf{T}(\mathsf{N}))\to\mathsf{L}(\mathsf{T}(\mathsf{N}))\\
\texttt{breadth} : (\mathsf{L}(\mathsf{T}(\mathsf{N})))\to\mathsf{L}(\mathsf{N})\\
e_{\texttt{snoc}}(d,l,t) = \mathsf{match}\ l\ \mathsf{with}\\
\quad\ \ \mathsf{nil}\Rightarrow\mathsf{cons}(d,t,\mathsf{nil})\\
\quad\ \ |\mathsf{cons}(d',t',q)\Rightarrow\mathsf{cons}(d',t',\texttt{snoc}(d,q,t))\\
e_{\texttt{breadth}}(q) = \mathsf{match}\ q\ \mathsf{with}\\
\quad\ \ \mathsf{nil}\Rightarrow\mathsf{nil}\\
\quad\ \ |\mathsf{cons}(d,t,q) = \mathsf{match}\ t\ \mathsf{with}\\
\quad\quad\ \mathsf{leaf}(a)\Rightarrow\mathsf{cons}(d,a,\texttt{breadth}(q))\\
\quad\quad\ \mathsf{node}(d_1,d_2,a,l,r)\Rightarrow\mathsf{cons}(d,a,\texttt{breadth}(\texttt{snoc}(d_2,\texttt{snoc}(d_1,q,l),r)))
\end{array}
$$

Other examples we have tried out include quicksort, treesort, and the Huffman algorithm.

Remark 3.1 *It can be shown that all definable functions are non size-increasing, e.g., if* $f : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$ *then, semantically,* $|f(l)| \le |l|$. *This would not be the case if we would omit the* $\diamond$ *argument in* cons*, even if we keep linearity. We would then, for example, have the function* $f(l) = \mathsf{cons}(0, l)$ *which increases the length. The presence of such a function in the body of a recursive definition gives rise to arbitrarily long lists.*

4 Compilation into C

Our aim is to compute the meanings of programs by `malloc()`-free `C`-programs.

Let a program $P = (\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$ be given. Let types(P) be the (finite) set of zero-order types mentioned in P. To each $A \in$ types(P) we assign a unique identifier $\nu(A)$. Next, for each $A \in$ types(P) we introduce a piece of code $[\![A]\!]^{\mathtt{C}}$ defining a `C` type $\nu(A)$ and some functions (e.g. constructors and destructors) associated with A.

The code $[\![\diamond]\!]^{\mathtt{C}}$ is: `typedef void *` $\nu(\diamond)$`;`

The code $[\![\mathsf{N}]\!]^{\mathtt{C}}$ is:

```
typedef int ν(N);
```

The code $[\![A \otimes B]\!]^{\mathsf{C}}$ is:

```
typedef struct {
  ν(A) fst;
  ν(B) snd;
} ν(A⊗B);
```

followed by

```
ν(A⊗B) ν(A⊗B)_pair
(ν(A) fst, ν(B) snd){
  ν(A⊗B) res;
  res.fst = fst;
  res.snd = snd;
  return res;
}
```

The code $[\![A + B]\!]^{\mathsf{C}}$ is:

```
[[A+B]]^C =
typedef struct {
  kind_t kind;
  union {
    ν(A) inl;
    ν(B) inr;
  } body;
} ν(A+B);
```

followed by

```
ν(A+B) ν(A+B)_inl(ν(A) x){
  ν(A+B) res;
  res.kind = INL;
  res.body.inl = x;
  return res; }

ν(A+B) ν(A+B)_inr(ν(B) x){
  ν(A+B) res;
  res.kind = INR;
  res.body.inr = x;
  return res; }
```

The code $[\![\mathsf{L}(A)]\!]^{\mathsf{C}}$ is:

```
typedef struct {
  kind_t kind;
  ν(A) hd;
  void * tl;
} ν(L(A));

ν(L(A)) ν(L(A))_nil(void){
  ν(L(A)) res;
  res.kind = NIL;
  return res;
}

ν(L(A)) ν(L(A))_cons
(ν(◇) d, ν(A) hd, ν(L(A)) tl){
  ν(L(A)) res;
  res.kind = CONS;
  res.hd = hd;
  *(ν(L(A)) *)d = tl;
  res.tl = d;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  ν(A) hd;
  ν(L(A)) tl;
  ν(◇) d;
} ν(L(A))_destr_t;

ν(L(A))_destr_t ν(L(A))_destr
(ν(L(A)) l){
  ν(L(A))_destr_t res;
  res.kind = l.kind;
  if(res.kind = CONS) {
    res.hd = l.hd;
    res.d = (void *)l.tl;
    res.tl = *l.tl;
  }
  return res;
}
```

The code $[\![\mathsf{T}(A)]\!]^{\mathsf{C}}$ is:

```
typedef struct {
  kind_t kind;
  ν(A) label;
  void * left;
  void * right;
} ν(T(A));
ν(T(A)) ν(T(A))_leaf
(ν(A) label){
  ν(T(A)) res;
  res.kind=LEAF;
  res.label=label;
  return res;
}
ν(T(A)) ν(T(A))_node
(ν(◇) d1, ν(◇) d2, ν(A) label,
    ν(T(A)) left, ν(T(A)) right){
  ν(T(A)) res;
  res.kind = NODE;
  res.label = label;
  *(ν(T(A)) *)d1 = left;
  *(ν(T(A)) *)d2 = right;
  res.left = d1;
  res.right = d2;
  return res;
}
ν(T(A))_destr_t ν(T(A))_destr
(ν(T(A)) t){
  ν(T(A))_destr_t res;
  res.label = t.label;
  res.kind = t.kind;
  if(res.kind == NODE) {
    res.d1 = (ν(◇))t.left;
    res.d2 = (ν(◇))t.right;
    res.left = *(ν(T(A)) *)t.left;
    res.right = *(ν(T(A)) *)t.right;
  }  return res; }
```

```
typedef struct {
  kind_t kind;
  ν(A) label;
  ν(◇) d1, d2;
  ν(T(A)) left, right; } ν(T(A))_destr_t;
```

We define $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$ as the declaration

```
typedef enum {INL, INR, NIL,
 CONS, LEAF, NODE} kind_t;
```

followed by the declarations $[\![A]\!]^{\mathtt{C}}$ for $A \in \mathrm{types}(P)$ ordered in such a way that $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$ is valid C code (this means that if type A occurs as a subexpression in type B then $[\![A]\!]^{\mathtt{C}}$ must occur before $[\![B]\!]^{\mathtt{C}}$.)

To each declaration $f : (A_1, \ldots, A_n) \to B$ in Σ we associate a prototype $[\![f]\!]^{\mathtt{C}}$ given by
$\nu(B)$ f($\nu(A_1)$ x_1, ..., $\nu(A_n)$ x_n)

The interpretation of the signature $[\![\Sigma]\!]^{\mathtt{C}}$ is given as the concatenation of $[\![f]\!]^{\mathtt{C}}$; (note the semicolon) for $f \in \mathrm{dom}(\Sigma)$.

A C-valuation of a context Γ is a finite function mapping identifiers in $\mathrm{dom}(\Gamma)$ to syntactically correct C-expressions.

To each expression $\Gamma \vdash_\Sigma e : B$ and C-valuation σ satisfying Γ we associate a C expression $[\![e]\!]^{\mathtt{C}}_\sigma$ of type $\nu(B)$. The definition will rely on the presence of certain C variables needed to store intermediate results. When we write something like "where $\nu(A)\,p$ fresh" we mean that p should be a variable of type $\nu(A)$ which is not used anywhere else in $[\![e]\!]^{\mathtt{C}}_\sigma$. This could be formalised by defining $[\![e]\!]^{\mathtt{C}}_\sigma$ as a pair $\langle V, D\rangle$ where V maps C-valuations to C expressions and where D maps sequences of variable declarations to sequences of variable declarations in such a way that $D(s)$ always contains s. We refrain, however, from such a formalisation to reduce notational clutter.

The definition of $[\![e]\!]^{\mathtt{C}}_\sigma$ is by induction on typing derivations of well-typed expressions. Recall the following usage of the comma operator in C which allows one to simulate a functional "let": an expression of the form $(x{=}\mathit{exp}_1, \mathit{exp}_2)$ is evaluated by first assigning the value of exp_1 to the variable x and then evaluating exp_2 (which may involve x). Unlike in the case of "let" the variable x must have been declared beforehand.

$$[\![x : A]\!]^{\mathtt{C}}_\sigma = \sigma(x)$$

$$[\![f(e_1,\dots,e_n) : A]\!]^{\mathtt{C}}_\sigma = f\texttt{(}[\![e_1]\!]^{\mathtt{C}}_\sigma\texttt{,}\ \dots\texttt{,}\ [\![e_n]\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$[\![c : \mathsf{N}]\!]^{\mathtt{C}}_\sigma = c$$

$$[\![e_1 \star e_2 : \mathsf{N}]\!]^{\mathtt{C}}_\sigma = \texttt{(}[\![e_1]\!]^{\mathtt{C}}_\sigma \star [\![e_2]\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$[\![\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A]\!]^{\mathtt{C}}_\sigma = \texttt{(}[\![e]\!]^{\mathtt{C}}_\sigma\ \texttt{?}\ [\![e']\!]^{\mathtt{C}}_\sigma\ \texttt{:}\ [\![e'']\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$[\![e \otimes e' : A \otimes B]\!]^{\mathtt{C}}_\sigma = \nu(A \otimes B)\_\texttt{pair(}[\![e]\!]^{\mathtt{C}}_\sigma\texttt{,}\ [\![e']\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$[\![\mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e']\!]^{\mathtt{C}}_\sigma = \texttt{(}p\texttt{=}[\![e]\!]^{\mathtt{C}}_\sigma\texttt{,}[\![e']\!]^{\mathtt{C}}_{\sigma'}\texttt{)}$$

where $\sigma' = \sigma[x \mapsto p\texttt{.fst}][y \mapsto p\texttt{.snd}]$
where $e : P$ and $\nu(P)\,p$ is fresh.

$$[\![\mathsf{inl}_{A,B}(e) : A + B]\!]^{\mathtt{C}}_\sigma = \nu(A+B)\_\texttt{inl(}[\![e]\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$[\![\mathsf{inr}_{A,B}(e) : A + B]\!]^{\mathtt{C}}_\sigma = \nu(A+B)\_\texttt{inr(}[\![e]\!]^{\mathtt{C}}_\sigma\texttt{)}$$

$$\llbracket \mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow e' | \mathsf{inr}(x) \Rightarrow e'' : C \rrbracket^{\mathsf{C}}_{\sigma} =$$
$$(\llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{.kind==INL ?}$$
$$\llbracket e' \rrbracket^{\mathsf{C}}_{\sigma[x \mapsto \llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{.body.inl}]}\ \texttt{:}$$
$$\llbracket e'' \rrbracket^{\mathsf{C}}_{\sigma[x \mapsto \llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{.body.inr}]})$$

$$\llbracket \mathsf{nil}_A : \mathsf{L}(A) \rrbracket^{\mathsf{C}}_{\sigma} = \nu(\mathsf{L}(A))\texttt{\_nil()}$$

$$\llbracket \mathsf{cons}(e_d, e_h, e_t) \rrbracket^{\mathsf{C}}_{\sigma} = \nu(\mathsf{L}(A))\texttt{\_cons(}\llbracket e_d \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\llbracket e_h \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\llbracket e_t \rrbracket^{\mathsf{C}}_{\sigma}\texttt{)}$$

$$\llbracket \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(d,h,t) \Rightarrow e_{\mathsf{cons}} : B \rrbracket^{\mathsf{C}}_{\sigma} =$$
$$(p\texttt{=}\nu(\mathsf{L}(A))\texttt{\_destr(}\llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{),}p\texttt{.kind==NIL ?}\ \llbracket e_{\mathsf{nil}} \rrbracket^{\mathsf{C}}_{\sigma}\ \texttt{:}\quad \llbracket e_{\mathsf{cons}} \rrbracket^{\mathsf{C}}_{\sigma'}\texttt{)}$$
where $\sigma' = \sigma[d \mapsto p\texttt{.d}, h \mapsto p\texttt{.hd}, t \mapsto p\texttt{.tl}]$
and $\nu(\mathsf{L}(A))\texttt{\_destr\_t}\ \ p$ is fresh

$$\llbracket \mathsf{leaf}(e) : \mathsf{T}(A) \rrbracket^{\mathsf{C}}_{\sigma} = \nu(\mathsf{T}(A))\texttt{\_leaf(}\llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{)}$$

$$\llbracket \mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r) \rrbracket^{\mathsf{C}}_{\sigma} =$$
$$\nu(\mathsf{T}(A))\texttt{\_node(}\llbracket e_{d1} \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\ \llbracket e_{d2} \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\llbracket e_a \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\llbracket e_l \rrbracket^{\mathsf{C}}_{\sigma}\texttt{,}\llbracket e_r \rrbracket^{\mathsf{C}}_{\sigma}\texttt{)}$$

$$\llbracket \mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a) \Rightarrow e_{\mathsf{leaf}} | \mathsf{node}(d_1, d_2, a, l, r) \Rightarrow e_{\mathsf{node}} : B \rrbracket^{\mathsf{C}}_{\sigma} =$$
$$(p\texttt{=}\nu(\mathsf{L}(A))\texttt{\_destr(}\llbracket e \rrbracket^{\mathsf{C}}_{\sigma}\texttt{),}p\texttt{.kind==LEAF ?}\ \llbracket e_{\mathsf{leaf}} \rrbracket^{\mathsf{C}}_{\sigma[a \mapsto p\texttt{.label}]}\ \texttt{:}$$
$$\llbracket e_{\mathsf{node}} \rrbracket^{\mathsf{C}}_{\sigma[d_1 \mapsto p\texttt{.d1}, d_2 \mapsto p\texttt{.d2}, a \mapsto p\texttt{.label}, l \mapsto p\texttt{.left}, r \mapsto p\texttt{.right}]}\texttt{)}$$
where $\nu(\mathsf{T}(A))\texttt{\_destr\_t}\, p$ is fresh

The interpretation $\llbracket P \rrbracket^{\mathsf{C}}$ of the *program* $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$, finally, is obtained as the concatenation of the following blocks:

1. the interpretation of the types: $\llbracket \mathrm{types}(P) \rrbracket^{\mathsf{C}}$,
2. the function prototypes: $\llbracket \Sigma \rrbracket^{\mathsf{C}}$,
3. for each $f \in \mathrm{dom}(\Sigma)$ a function definition

   $\llbracket f \rrbracket^{\mathsf{C}}$ {
   $\quad DECL(e)$
   $\quad$ `return` $\llbracket e \rrbracket^{\mathsf{C}}_{[x_1 \mapsto x_1, \ldots x_1 \mapsto x_1]}$;
   }

   where $DECL(e)$ consists of all the variable declarations which have occurred in the definition of $\llbracket e \rrbracket^{\mathsf{C}}_{[x_1 \mapsto x_1, \ldots x_1 \mapsto x_1]}$.

We state the following type-correctness property which may be proved by induction on typing derivations.

Lemma 4.1 *If* $\Gamma \vdash_\Sigma e : A$ *and* $\sigma(x)$ *is a* C*-expression of type* $\nu(\Gamma(x))$ *for each* $x \in \mathrm{dom}(\Gamma)$ *then* $[\![e]\!]^{\mathtt{C}}_{\sigma}$ *is a* C*-expression of type* $\nu(A)$ *when it appears after the declarations* $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$ *and* $[\![\Sigma]\!]^{\mathtt{C}}$ *and DECL*(e)*.*

It follows that the interpretation of a program is a legal C program which implements functions of the same type as the ones mentioned in the signature.

4.1 Correctness of the translation

We will now show that the translation is correct in the sense that the translation of a program computes the program's set-theoretic semantics under some reasonable assumptions on the semantics of C.

Here we face the problem that C does not have a formally defined semantics and if it had it would be fairly complex. We could of course replace C with an artificial language or an abstract machine, but this would mean that we would have to re-define our translation and the question what this artificial language has to do with the translation into C must again be answered informally.

For these reasons, we have decided to opt for a semi-formal proof based on some reasonable assumptions on the semantics of C which we now outline.

4.2 Informal semantics of C

We view addresses as integers and write $\mathcal{L}$ for the set of integers when used as addresses.

We assume a set $\mathcal{S}$ of *stack values* comprising integers, heap addresses, as well as records consisting of stack values. Records may be understood as finite functions from identifiers to stack values. For simplicity, we view unions as records. We use dot notation to access components of records.

For each C type A we have a subset of the stack values corresponding to that type. E.g. a stack value of type `list_t` would be a 3-tuple consisting of an integer (the kind), another integer (the head), and an address (a pointer to the tail). Of course, in the case of an empty list (kind field equal to `NIL`) these latter entries are garbage.

We assume for each C type A an integer $\mathtt{sizeof}(A)$ and a bijection ϕ_A between the stack values of type A and $\mathtt{sizeof}(A)$-tuples of integers.

A *heap configuration* H is a function from $\mathcal{L}$ to integers. We write $\mathcal{H}$ for the set of heap configurations. If H is a heap configuration, h is an address, and A is a C type, then we define the *stack value of type* A *pointed to by* h *in* H, written $H_A(h)$, by

$$H_A(h) = \phi_A^{-1}(H(h), H(h+1), \dots, H(h + \mathtt{sizeof}(A) - 1))$$

If s is a stack value of type A and H is a heap configuration and h is an address then we define a new heap configuration $H[h \mapsto_A s]$ as the heap configuration which contains the components of $\phi_A(s)$ at addresses $h, h+1, \ldots, h+\texttt{sizeof}(A)-1$ and is like H elsewhere.

$$H[h \mapsto_A s](h+i) = \phi_A(s)_i, \text{ when } 0 \leq i < \texttt{sizeof}(A)$$
$$H[h \mapsto_A s](h') = H(h'), \text{ when } h' \notin \{h, \ldots, h+\texttt{sizeof}(A)-1\}$$

The semantics of statements and expressions is always relative to a *runtime environment* mapping free program variables to stack values of appropriate type and also depends implicitly on prior function definitions. In a given runtime environment the semantics of a statement is a partial function from heap configurations to heap configurations. For example, the semantics of the statement $*(A*)\texttt{h} = \texttt{s}$; is the function sending H to $H[h \mapsto_A s]$ assuming that h, s are the stack values associated by the runtime environment with the variables $\texttt{h}, \texttt{s}$.

The semantics of an expression in a given runtime environment is a partial function mapping heap configurations to pairs of stack values and heap configurations. For example, the semantics of the expression $*(A*)h$ is the function mapping H to $H_A(h)$.

The semantics of an n-ary function is independent of the runtime environment (if we disregard global variables) and consists of a partial function $\mathcal{H} \times \mathcal{S}^n \rightharpoonup \mathcal{H} \times \mathcal{S}$.

For example, let us apply the ternary function `cons` from the Introduction to heap configuration H and arguments d, h, t where d is an address, h is an integer, t is a stack value of type `list_t`. The result of this will be the stack value of type `list_t` whose `kind` field is `CONS`, whose `hd` field is h and whose `tl` field is d; the resulting new heap configuration is $H[d \mapsto_{list\_t} t]$.

As usual, partiality arises from nontermination.

4.3 The correctness theorem

For the rest of this section we fix a program $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$.

Definition 4.2 For each zero-order type A occurring in P the set $\mathcal{V}(A)$ is given by the set of pairs (v, R) where v is a `C`-stack-value of type $\nu(A)$ (under the definitions $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$) and R is a region in the heap (a set of addresses). We use the symbol $\uplus$ to denote union of disjoint sets.

For each heap configuration H and zero-order type A occurring in P we define a relation $\Vdash^H_A \subseteq \mathcal{V}(A) \times [\![A]\!]$ inductively as follows:

- $(n, \emptyset) \Vdash^H_{\mathsf{N}} n'$, if $n = n'$,
- $(h, R) \Vdash^H_{\diamond} 0$, if $R = \{h, h+1, \ldots, h+M-1\}$ where $M = \max\{\texttt{sizeof}(\nu(A)) \mid A \in \mathrm{types}(P)\}$ is the size of the largest type occurring in P,
- $(v, R) \Vdash^H_{A \otimes B} (a, b)$ if $R = R_1 \uplus R_2$ and $(v\texttt{.fst}, R_1) \Vdash^H_A a$ and $(v\texttt{.snd}, R_2) \Vdash^H_B b$,
- $(v, R) \Vdash^H_{A+B} \mathsf{inl}(a)$ if $v\texttt{.kind} = \texttt{INL}$ and $(v\texttt{.body.inl}, R) \Vdash^H_A a$,
- $(v, R) \Vdash^H_{A+B} \mathsf{inr}(b)$ if $v\texttt{.kind} = \texttt{INR}$ and $(v\texttt{.body.inr}, R) \Vdash^H_B b$,

- $(v, \emptyset) \Vdash^H_{\mathsf{L}(A)}$ nil if v.`kind` = `NIL`,
- $(v, R) \Vdash^H_{\mathsf{L}(A)}$ $\mathsf{cons}(h, t)$, if v.`kind` = `CONS` and $R = R_d \uplus R_h \uplus R_t$ and $(v.\mathtt{tl}, R_d) \Vdash^H_\diamond 0$ and $(v.\mathtt{hd}, R_t) \Vdash^H_A h$ and $(H_{\nu(\mathsf{L}(A))}(v.\mathtt{tl}), R_t) \Vdash^H_{\mathsf{L}(A)} t$,
- $(v, R) \Vdash^H_{\mathsf{T}(A)}$ $\mathsf{leaf}(a)$ if v.`kind` = `LEAF` and $(v.\mathtt{label}, R) \Vdash^H_A a$,
- $(v, R) \Vdash^H_{\mathsf{T}(A)}$ $\mathsf{node}(a, l, r)$ if v.`kind` = `NODE` and $R = R_{d1} \uplus R_{d2} \uplus R_a \uplus R_l \uplus R_r$ and $(v.\mathtt{left}, R_{d1}) \Vdash^H_\diamond 0$ and $(v.\mathtt{right}, R_{d2}) \Vdash^H_\diamond 0$ and $(v.\mathtt{label}, R_a) \Vdash^H_A a$ and $(H_{\nu(\mathsf{T}(A))}(v.\mathtt{left}), R_l) \Vdash^H_{\mathsf{T}(A)} l$ and $(H_{\nu(\mathsf{T}(A))}(v.\mathtt{right}), R_r) \Vdash^H_{\mathsf{T}(A)} r$.

□

Notice that whenever A is heap-free and $(v, R) \Vdash^H_A a$ for some a then $R = \emptyset$ and H is irrelevant.

Theorem 4.3 *Assume the following:*

- *a well typed expression $\Gamma \vdash_\Sigma e : A$ involving only the types in P,*
- *for each $x \in \Gamma$ a value $(v_x, R_x) \in \mathcal{V}(\Gamma(x))$ such that $R_x \cap R_y = \emptyset$ whenever $x \neq y$,*
- *a heap configuration H,*
- *a set-theoretic valuation η of Γ such that $(v_x, R_x) \Vdash^H_{\Gamma(x)} \eta(x)$ for each $x \in \mathrm{dom}(\Gamma)$,*

Let ρ be the set-theoretic interpretation of P.

Then the evaluation of $[\![e]\!]^{\mathsf{C}}_{[x \mapsto x | x \in \mathrm{dom}(\Gamma)]}$ in heap configuration H and runtime environment which maps $x \in \mathrm{dom}(\Gamma)$ to stack value v_x and interprets function calls according to $[\![P]\!]^{\mathsf{C}}$ will either not terminate or result in a `C` *stack value v and heap configuration H' such that $(v, R) \Vdash^{H'}_A [\![e]\!]_{\eta,\rho}$ for some subset $R \subseteq \biguplus_{x \in \mathrm{dom}(\Gamma)} R_x$ and moreover the part of the heap outside of $\biguplus_{x \in \mathrm{dom}(\Gamma)} R_x$ will be left unaffected by the evaluation, i.e., $H'(h) = H(h)$ for $h \notin \biguplus_{x \in \mathrm{dom}(\Gamma)} R_x$.*

Proof. Induction on the lexicographic product of evaluation time (if desired formalised using the operational semantics of `C` as well) and length of typing derivations with second priority. This second factor is needed only to cater for rule Contr.

With all the previous definitional apparatus the proof itself is straightforward. We show four representative cases.

Suppose that the last typing rule applied was Pair. Then $A = A_1 \otimes A_2$ and $e = e_1 \otimes e_2$ and $\Gamma = \Gamma_1, \Gamma_2$ and the induction hypothesis applies to $\Gamma_i \vdash e_i : A_i$ for $i = 1, 2$. This subdivision of contexts partitions our heap $\biguplus_{x \in \mathrm{dom}(\Gamma)} R_x$ into two portions $\biguplus_{x \in \mathrm{dom}(\Gamma_i)} R_x$ for $i = 1, 2$ in which the computations of $[\![e_1]\!]^{\mathsf{C}}_{\sigma_1}$ and $[\![e_2]\!]^{\mathsf{C}}_{\sigma_2}$ take place. Accordingly, these can be carried out one after the other without interfering which each other or destroying each others results so that — without any further heap operations — the respective results can be put together to form the result of $[\![e]\!]^{\mathsf{C}}_\sigma$.

Now suppose that the last rule applied was CONS. Again, the computations of the subexpressions are carried out in disjoint parts of the heap; the results of the computation will also point to disjoint parts of the heap by the induction hypothesis. Therefore, we can put them together to form a valid list value.

Suppose that the last rule applied was SIG, i.e., $e = f(e_1, \ldots, e_n)$. Suppose furthermore, that $\Gamma_f \Vdash^H e_f : B$. The induction hypothesis applied to e_i and an appropriate splitting of η similar to the case $\otimes$ yields an environment η_f for Γ_f. Applying the induction hypothesis to $[\![e_f]\!]^{\mathsf{C}}_{\eta_f}$ yields the result. Notice that since e_f may itself contain calls to f we cannot rely on induction on typing derivations alone, but also need to induct on evaluation time.

Finally, consider rule CONTR. We want to apply the induction hypothesis with $R_x = R_y = R_x$. In general, this would not be possible since $R_x \cap R_y \neq \emptyset$. In the case where the type A of the variables x, y is heap-free, we have $R_x = R_y = \emptyset$ and hence R_x, R_y are disjoint as required. □

It follows by specialising to the defining expressions e_f that a program computes its set-theoretic interpretation.

5 Expressivity

In a certain trivial sense LFPL is Turing complete because every partial recursive function can be represented as a function of type $(\mathsf{N}) \to \mathsf{N}$. Of course, this is unrealistic, as the C-type `int` only ranges over 32 bit integers (at the time of writing!). In another extreme sense the system is no more powerful than a finite-state machine, since in a real computer even stack and heap are finite.

However, it appears to be a reasonable abstraction to view the type N as finite, e.g., to consist of 32 bit words (or indeed to replace it by a type of booleans) and to view stack and heap as potentially infinite (unbounded). This means that unbounded input data must be presented using a heap-allocated type such as $\mathsf{L}(\mathsf{N})$. In this case, we have the following expressivity result:

Theorem 5.1 *Let $f : \mathbb{N} \to \mathbb{N}$ be a non size-increasing function, i.e., $f(|x|) \leq |x|$ where $|x| = \lceil \log_2(x+1) \rceil$. The following are equivalent:*

- *f is a computable in linear space (linear in the length of the input).*
- *There exists a tail-recursive program containing a symbol $\mathtt{f} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$ such that $[\![\mathtt{f}]\!](u(x)) = u(f(x))$ when $u : \mathbb{N} \to \{0,1\}^*$ is a reasonable encoding of natural numbers as lists of 0s and 1s.*

Proof. If $f(x)$ is computable in space cn where $n = |x|$ then we use the type $T = \mathsf{L}(\mathsf{N} \otimes \ldots \otimes \mathsf{N})$ with c factors to store memory configurations. We obtain $\mathtt{f}$ by iterating a one-step function of type $(T)\to T$ and composing with an initialisation function of type

$(\mathsf{L}(\mathsf{N}))\to T$ and an output extraction function of type $(T)\to\mathsf{L}(\mathsf{N})$ all of which are readily seen to be implementable in LFPL.

For the other direction we simulate computations in LFPL on a space-bounded Turing machine which operates on an encoding of the heap using an algebraic notation for data structures. For example a binary tree labelled 0 with leaves labelled 1 and 2 might be represented as the string $N0L1L2$. It is easy to see that the size of this encoding is linear in the size of the input.

We note that the translation to `C` itself cannot be used to prove this direction. The reason is that each node requires space $O(\log(n)$ because under the assumption of a potentially infinite heap we must allow space $O(\log(n))$ to store a pointer. □

In order to assess the expressive power of the full system we use the following result [7] due to Stephen Cook[3] [7].

Theorem 5.2 (Cook) *The following are equivalent for a function $f : \mathbb{N} \to \mathbb{N}$*

i) $f(x)$ is computable in time $O(2^{c|x|})$ for some c,

ii) $f(x)$ is computable by a $O(|x|)$-space bounded Turing machine having as extra resource an unbounded stack.

The "stack" can e.g. be formalised as a one side infinite tape undergoing the restrictions that all fields to the left of the read-write head must be blank.

Theorem 5.3 *Let $f : \mathbb{N} \to \mathbb{N}$ be a non size-increasing function. f is representable in the sense of Theorem 5.1, i.e., with integers encoded as lists, if and only if $f(x)$ is computable on a Turing machine in time $O(2^{c|x|})$.*

Proof. For the "only if" direction we notice that we can evaluate arbitrary programs on a $O(n)$ space-bounded Turing machine with unbounded stack using the encoding of heap contents described above and the usual encoding of general recursion as iteration with a stack. Cook's theorem then furnishes the result.

For the "if" direction we assume an $C2^{c|x|}$ time-bounded Turing machine M computing f. For integers x, t, a with t (time) and a (address) below $C2^{c|x|}$ we introduce the function

$$\pi(x, t, a) = (x, t, a, q, b)$$

where q is the state reached by M after t steps on input x and b is the symbol on the tape at position a after t steps on input x.

Now, clearly, $\pi(x, t+1, a)$ can be expressed in terms of $\pi(x, t, a-1), \pi(x, t, a), \pi(x, t, a+1)$ thus giving rise to a definition in LFPL.

Alternatively, we could use an encoding of stack computations as recursive functions and appeal again to Cook's theorem. □

[3]Cook's original result asserts more generally that Turing machines with unbounded stack and $O(s(n))$ bounded tapes are equipotent to $O(2^{cs(n)})$ time-bounded Turing machines provided that $s(n) \geq \log(n)$.

6 Extensions

Dynamic allocation As it stands there is no way to create a value of type $\diamond$, so in particular, it is not possible to create a non-nil constant of list type. The examples show that in order to define functions this is often not needed. Sometimes, however, dynamic allocation and deallocation may be required and to this end we can introduce functions $\mathsf{new} : ()\to\diamond$ and $\mathsf{disp} : (\diamond)\to\mathsf{N}$ whose interpretation is given by C-functions

```
ν(◇) new()
  {return malloc(sizeof(T);}
ν(N) disp(ν(◇) d)
  {free((T *)d);return 0;}
```

where T is a `union` with one entry for each $A \in \mathrm{types}(P)$.

An example, where this might come in useful is the definition of a function which arranges the labels of a tree in a list of lists containing one entry for each depth level of the tree. It is readily seen, that this requires extra space in the order of the depth of the tree which is best obtained dynamically. If this list of lists is subsequently flattened to compute the list of labels in breadth-first order then this space becomes available again and could be deallocated. Of course, programs involving `new` come with no guarantee on their space usage. Obtaining verifiable space bounds for programs involving this construct is left for future research.

Nondestructive computation, I/O We can include functions `print_int` : $(\mathsf{N})\to\mathsf{N}$ and `read_int` : $()\to\mathsf{N}$ with the obvious semantics. From these we can write recursive I/O functions for other datatypes.

It is then advisable to type a list-printing function as of type $(\mathsf{L}(A))\to\mathsf{L}(A)$ so as to circumvent destruction. A similar typing should be used for other non destructive functions such as computing the length of a list, i.e., `length` : $(\mathsf{L}(A))\to\mathsf{N}\otimes\mathsf{L}(A)$, etc. A perhaps more convenient alternative would be a refined type system which allows one to express that a function does not destroy its argument. It appears that ideas from O'Hearn-Reynolds-Tennent's SCI type system [22, 20] could be of use there.

Polymorphism, higher-order functions We can extend the language with polymorphism (with two kinds of type variables ranging over zero- and first order types) and higher-order functions, both linear (written $A \multimap B$) and nonlinear (written $!A \multimap B$). Recursive functions would then be defined using a single constant

$$\mathsf{rec} : \forall X.!(!X \multimap X) \multimap X$$

where X ranges over first-order types.

The innermost ! signifies that more than one recursive call is allowed; the outermost ! stipulates that the body of a recursively defined function must not depend on linear variables.

To handle mutual recursion a slightly more complicated syntax involving a `let rec` construct is needed.

It is then possible to compile every first-order term—even if it contains higher-order functions as subexpressions—into equivalent `C`-code, e.g., by β-reducing all nonlinear applications and type applications and then applying the existing translation. The advantage of such an extension is that schematic programs such as `append` for arbitrary type or `maplist` for arbitrary function would need to be written only once.

If we were to allow recursion with higher-order result type X not containing ![4] then we would have to hold intermediate results of functional type (closures) in the heap. While the *evaluation* of linear such terms takes place in linear time and constant space the size of these terms in the course of a recursion is not bounded by the input. The complexity-theoretic strength of such a system is open. We remark, however, that if we limit recursion to *structural recursion* on lists and trees then the results in [10] guarantee in this case a polynomial bound on evaluation time (and space).

Queues The program for breadth-first search could be made more efficient using queues with constant time enqueuing. In LFPL we can achieve this effect very easily by adding a new type former $\mathsf{Q}(A)$ which is interpreted as:

```
typedef struct {
  ν(L(A)) qu;
  ν(L(A)) * end;
} ν(Q(A));
```

We introduce (formally with typing rules) basic functions:

$\mathsf{nil} : () \to \mathsf{Q}(A)$
$\mathsf{cons} : (\Diamond, A, \mathsf{Q}(A)) \to \mathsf{Q}(A)$
$\mathsf{snoc} : (\Diamond, \mathsf{Q}(A), A) \to \mathsf{Q}(A)$
$\mathsf{append} : (\mathsf{Q}(A), \mathsf{Q}(A)) \to \mathsf{Q}(A)$
$\mathsf{dequeue} : (\mathsf{Q}(A)) \to \mathsf{N} + \Diamond \otimes A \otimes \mathsf{Q}(A)$

The set-theoretic interpretation is given by $[\![\mathsf{Q}(A)]\!] = [\![\mathsf{L}(A)]\!]$. We have $(v, H) \Vdash^H_{\mathsf{Q}(A)} q$ iff $(v\,.\,\mathsf{qu}, H) \Vdash^H_{\mathsf{L}(A)} q$ and $v\,.\,\mathtt{end}$ points to the end of v or equals `NULL` in case $q = []$. If desired this can be turned into a formal inductive definition.

The semantics of $\mathsf{dequeue}$ is given by

$[\![\mathsf{dequeue}]\!]([]) = \mathsf{inl}(0)$
$[\![\mathsf{dequeue}]\!](\mathsf{a} :: \mathsf{l}) = \mathsf{inr}(0, \mathsf{a}, \mathsf{l})$

[4] Allowing recursion with arbitrary result type means that we can use nonlinear functions throughout and thus place no restriction whatsoever on our programs.

It should be clear from this how the basic functions associated with queues are interpreted and how the correctness proof extends.

Notice that the linear typing allows us again to modify the data structure in place while allowing for equational reasoning.

In a similar way we can provide other familiar data structures provided they admit an ADT-like functional interface.

Tail recursion The type system does not impose any restriction on the size of the *stack*. If a bounded stack size is desired, all we need to do is restrict to a tail recursive fragment and translate the latter into iteration.

More challenging would be some automatic program transformation which translates the existing (non-tail recursive) definition of `breadth` and similar functions into iterative code. To what extent this can be done systematically remains to be seen.

Recursive types We can extend the type system and the compilation technique to arbitrary (even nested) first-order recursive types. To that end, we introduce (zero order) type variables and a new type former $\mu X.A$ which binds X in A. Elements of $\mu X.A$ would be introduced and eliminated using fold and unfold constructs

$$\frac{\Gamma \vdash_\Sigma e : A[(\diamond \otimes \mu X.A)/X]}{\Gamma \vdash_\Sigma \mathsf{fold}(e) : \mu X.A} \qquad (\textsc{Fold})$$

$$\frac{\Gamma \vdash_\Sigma e : \mu X.A}{\Gamma \vdash_\Sigma \mathsf{unfold}(e) : A[(\diamond \otimes \mu X.A)/X]} \qquad (\textsc{Unfold})$$

This together with coproduct and unit types allows us to *define* lists and trees as recursive datatypes. Notice that this encoding would also charge two $\diamond$s for a tree constructor. Unfortunately, the type of queues mentioned above would still have to be introduced as a primitive. The definition of a generic syntax allowing one to specify such non-free datatypes is left to future research.

The translation of a type $\mu X.A$ would be obtained as the translation of the type A where the variable X is translated as `void *`. The definition of the auxiliary functions requires the definition of "functorial strength", i.e., a way of lifting a function $X \to Y$ to a function $A(X) \to A(Y)$. Such construct can be readily obtained by induction on A.

Additive rule for conditionals It is somewhat annoying that the LFPL type system insists that no variables be shared between guard and branches of a conditional. We will now discuss the implications of the following laxer typing rule which allows such sharing:

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{N} \qquad \Gamma \vdash_\Sigma e' : A \qquad \Gamma \vdash_\Sigma e'' : A}{\Gamma \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \qquad (\textsc{If-Add})$$

This would allow us to use the program for insertion sort without changes for arbitrary (not necessarily heap-free) type of entries. Notice also that such typing rule would allow us to encode the recursive definition of the function π from the proof of Cook's theorem directly.

Unfortunately, the relatively straightforward translation of LFPL into `C` code does not carry over to programs typed with this more relaxed rule. Nevertheless, we can evaluate such programs on a linearly space bounded Turing machine with unbounded stack as follows. In order to evaluate a conditional expression $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$ where non-heap free variables are shared between e and the branches e', e'' we first save on the stack a copy of the portion of the heap corresponding to the shared variables. Thereafter, we evaluate e which may work on this portion of the heap, possibly destroying its contents, but will eventually release it since its (heap-free) result value does not refer to the heap. We can then restore the old contents of the heap using the information saved previously on the stack and proceed to evaluate either e or e' according to the outcome of e (which of course we should have stored in the finite control, i.e., a register, rather than on the stack). This demonstrates that the rule If-Add does not increase the expressive power of the system.

One should note, however, that saving the content of the heap on the stack is a rather expensive operation and therefore use of the rule If-Add should be used with care, especially, when the program under consideration is otherwise tail recursive or can be transformed into one such and therefore does not use the stack much.

7 Experiments

A prototype version of the translation has been implemented by Nick Brown; an interactive version is accessible from the author's WWW page (`www.dcs.ed.ac.uk/home/mxh`).

The `C` code thus resulting from the above example programs was then compared with equivalent functional programs written in Standard ML of New Jersey (110.0.6) and Objective Caml 3.00. While these functional programs would run slightly faster than the generated `C`-code they use about 10 times as much space. Quite surprisingly, however, native code generated with the optimising Objective Caml compiler OCAMLOPT ran faster and used even less space than the generated `C`-code.

This seems to be due to clever use of caching on the one hand, and to the certain overhead in our translation (many function calls, unnecessary copying, recursion instead of iteration) which could be improved in a practical implementation tuned towards efficiency as opposed to clarity and compositionality.

The benchmark chosen was an application of the abovedefined breadth-first traversal function to full binary trees of depths 12-15.

Also the lazy functional language Clean slightly improves upon the generated `C` code (and OCAMLOPT!), however, due to lazyness the tree to be traversed is never laid out in the heap in full so that we are not really comparing like with like.

I stress that these experiments are not meant to demonstrate the usefulness of the present approach as a new optimisation technique, but that the translated programs can compete with state-of-the-art functional programming languages while guaranteeing static bounds on heap size and being evaluable without runtime garbage collection.

8 Conclusion

We have defined a linearly typed first-order language which gives the user explicit control over heap space in the form of a resource type.

A translation of this system into `malloc()`-free `C` is given which in the case of simple examples such as list reversal and quicksort generates the usual textbook solutions with in-place update. (To strictly agree with "textbooks" we have to represent the empty list as a `NULL` pointer which as explained above we have not done for the sake of uniformity.)

We have shown the correctness of this compilation with respect to a standard set-theoretic semantics which disregards linearity and the resource type and demonstrated the applicability by a range of small examples.

The main selling points of the approach are

1. that it achieves in place update of heap allocated data structures while retaining the possibility of equational reasoning and induction for the verification and

2. that it generates code which is guaranteed to run in a heap of statically determined size.

This latter point should make the system interesting for applications where resources are limited, e.g. computation over the Internet, programming of smart cards and embedded systems. Of course further work, in particular an integration with a fully-fledged functional language and the possibility of allocating a fixed amount of extra heap space will be required. Notice, however, that this latter effect can already be simulated by using input of the form $\mathsf{L}(\Diamond \otimes A)$ as opposed to $\mathsf{L}(A)$.

Also, a type inference system relieving the user from having to explicitly move around the $\Diamond$-resource might be helpful although the present system has the advantage of showing the user in an abstract and understandable way where space is being consumed. And perhaps some programmers might even enjoy spending and receiving $\Diamond$s.

9 Related work

While the idea of translating linearly typed functional code directly into `C` seems to be new there exist a number of related approaches aimed at controlling the space usage of functional programs.

Tofte-Talpin's region calculus [23] tries to minimise garbage collection by dividing the heap into a list of regions which are allocated and deallocated according to a stack discipline. A type systems ensures that the deallocation of a region does not destroy data which is still needed; an inference system [24] generates the required annotations automatically for raw ML code.

The difference to the present work is not so much the inference mechanism (see above) but the fact that even with regions the required heap size is potentially unbounded whereas the present system guarantees that the heap will not grow. Also in place update does not take place in.

Hughes and Pareto's system of sized types annotates list types with their length, e.g. the reversal function would get type $\forall n.\mathsf{L}_n(A) \rightarrow \mathsf{L}_n(A)$. While this system allows one to estimate the required heap and stack size it does not perform in place update either (and cannot due to the absence of linear types).

In a similar vein Crary and Weirich [8] have given a type system which allows one to formalise and certify informal reasoning about time consumption of recursive programs involving lists and trees. Their language is a standard one and no optimisation due to heap space reuse is taken into account.

The relationship between linear types and garbage collection has been recognised as early as '87 by Lafont [15], see also [11, 1, 25, 17] and [4] for a similar approach not based on the syntax of linear logic.

But again, due to the absence of $\diamond$-types, these systems do not provide in-place update but merely deallocate a linear argument immediately after its use.

This effect, however, is already achieved by traditional reference counting which may be the reason why linear functional programming hasn't really got off the ground, see also [6]. While the runtime advantages of the present approach are also realised by state-of-the-art garbage collection techniques (see the above section on experiments) the distinctive novelty lies in the fact that one can *guarantee* bounded heap size and obtain a simple `C` program realising it which can be run on any machine or system supporting `C`.

The type system itself is very similar to the system described by the author in [10] which in turn was inspired by Caseiro's analysis of recursive equations [5] and bears some remote similarity with Bounded Linear Logic [9]

Mention should also be made of Baker's Linear LISP [2, 3] which bears some similarity to LFPL. It does not contain the resource type $\diamond$ or a comparable feature, thus it is not clear how the size of intermediate data structures is limited, cf. Remark 3.1. Similar ideas, without explicit mention of linearity are also contained in Mycroft's thesis [18]

Mycroft and Sharp [19] have introduced a functional language which guarantees static memory bounds by restricting to tail-recursion and to stack allocated datatypes such as bytes, characters, and integers. They have found intriguing applications to hardware design.

Remotely related works are [12, 21].

Acknowledgement I would like to thank Samson Abramsky for helpful comments and encouragements, David Aspinall for proofreading and commenting, Neil Jones for pointing me to Cook's result and for encouragements, and Peter Selinger for spotting a shortcoming in an earlier version of this paper. Two anonymous referees made careful and detailed comments on presentation which have shaped this final version.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Henry Baker. Lively Linear LISP—Look Ma, No Garbage. *ACM Sigplan Notices*, 27(8):89–98, 1992.

[3] Henry Baker. A Linear Logic Quicksort. *ACM Sigplan Notices*, 29(2):13–18, 1994.

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

[5] Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions*. PhD thesis, University of Oslo, 1997. Available by ftp from `ftp.ifi.uio.no/pub/vuokko/0adm.ps`.

[6] J. Chirimar, C. Gunter, and J. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2), 1995.

[7] Stephen A. Cook. Linear-time simulation of deterministic two-way pushdown automata. *Information Processing*, 71:75–80, 1972.

[8] K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*, pages 184–198. ACM, 2000.

[9] J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.

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

[11] Sören Holmström. A linear functional language. In *Proceedings of the Workshop on Implemenation of Lazy Functional Languages*. Chalmers University, Göteborg, Programming Methodology Group, Report 53, 1988.

[12] Paul Hudak and Chih-Ping Chen. Rolling your own mutable adt — a connection between linear types and monads. In *Proc. Symp. POPL '97, ACM*, 1997.

[13] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ACM). Paris, September '99.*, pages 70–81, 1999.

[14] Kelley and Pohl. *A book on C, third edition*. Benjamin/Cummings, 1995.

[15] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[16] Xavier Leroy. The Objective Caml System, documentation and user's guide. Release 2.02. http://pauillac.inria.fr/ocaml/htmlman, 1999.

[17] P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Proc. LICS 1992, IEEE*, 1992.

[18] Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs.* PhD thesis, Univ. Edinburgh, 1981.

[19] Alan Mycroft and Richard Sahrp. A Statically Allocated Parallel Functional Language . In Ugo Montanari et al., editor, *Automata, languages, and programming (Proc. ICALP 2000)*, Lecture Notes in Computer Science 1853. Springer Verlag, 2000.

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

[21] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

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

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

[24] Mads Tofte and Lars Birkedal. Region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.

[25] D. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, 1999.