In-place update with linear types
or
How to compile functional programs into malloc()-free C

Martin Hofmann*

Synopsis: We show how linear typing can be used to write C functions which modify their heap allocated arguments in a functional way without explicit "pointer surgery."

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.

A number of examples including list quicksort and breadth-first traversal are given. Comparisons with existing compilers are described.

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 C is notoriously cumbersome, error prone, and difficult to teach.

Suppose that a type of lists has been defined 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 node.

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 labels(tree_t t)
```

which constructs the list of labels of tree t 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 [].

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

let rec insert a l = match l with
  [] -> [a]
| b::l -> if a<=b
            then a::b::l
            else b::(insert a l)

let rec sort l = match l with
  [] -> []
| a::l -> insert a (sort l)
```

*LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK. E-mail: mxh@dcs.ed.ac.uk

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

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

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.

An aside: a critic may argue that a skilled programmer would write the corresponding `C` code in the same time if not faster than the functional code. I would like to compare this with doing lengthy arithmetic or algebraic calculations with pencil and paper as opposed to using a pocket calculator or a computer algebra system. Although some individuals develop considerable skill at the former activity and may enjoy developing ingenious shortcuts here and there one must agree that this will invariably distract the mind from more creative tasks. (end of aside.)

The main idea of the proposed approach is that the programmer should pay for the use of constructor functions like `cons()`, `node()`, etc., as these require the allocation of heap space.

This is done by introducing a special type $\diamond$ or `dia_t` pointing to free heap space of appropriate size to store one list or tree node. When invoking a recursive constructor function such as `cons()` or `node()` one must supply an appopriate number of arguments of this type to provide the required heap space. Conversely, if in a recursion an argument of list or tree type is decomposed these $\diamond$-values become free. Linear typing then ensures that overwriting the heap space pointed to by these $\diamond$-values is safe.

2 Functional programming with C

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

```
typedef void * dia_t;

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

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;
}

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

list_destr_t list_destr(list_t l) {
  list_destr_t res;
  if (res.kind = l.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 free heap location capable of storing a list node (`d`).

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

```
list_t rev_aux(list_t l0, list_t acc) {
  list_destr_t l = list_destr(l0);
  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 `malloc()` statement is used.

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 l0) {
```

```
  list_destr_t l = list_destr(l0);
  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 l0) {
  list_destr_t l = list_destr(l0);
  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.

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 functions `dia_t new()` and `void disp(dia_t d)` which allocate and deallocate heap space in portions of one list node:

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

void disp(dia_t d) {
  free((list_t *)d);
}
```

This allows us to write an input function

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

and a printing function

```
void printlist(list_t l0) {
  list_destr_t l = list_destr(l0);
  switch(l.kind) {
  case NIL:
    printf("[]");break;
  case CONS:
    disp(l.d);
    printf("%d :: ", l.hd);
    printlist(l.tl);break;
  }
}
```

Notice that the `disp()` statement is safe at any time in the `CONS` case of `printlist()`. The three components of `l` are completely independent of each other.

Alternatively, we can write a print function of type

```
list_t printlist(list_t l);
```

which returns its argument rather than destroying it.

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 must use its argument at most once.

For instance, the following code which attempts to double the size of its argument would be incorrect:

```
list_t twice(list_t l0) {
  list_destr_t l = list_destr(l0);

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

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`.

As a last example let us look at binary `int`-labelled trees. We define

```
typedef struct tnode {
  kind_t kind;
  int label;
  struct tnode * left;
  struct tnode * right;
} tree_t;
tree_t leaf(int label) {
  tree_t res;
  res.kind = LEAF;
  res.label = label;
  return res;
}

tree_t node(dia_t d1, dia_t d2, int la-
bel, tree_t left, tree_t right) {
  tree_t res;

  res.kind = NODE;
  res.label = label;

  *(tree_t *)d1 = left;
  *(tree_t *)d2 = right;
```

```
  res.left = (tree_t *)d1;
  res.right = (tree_t *)d2;
  return res;
}

typedef struct {
  kind_t kind;
  int label;
  dia_t d1, d2;
  tree_t left;
  tree_t right;
} tree_destr_t;

tree_destr_t tree_destr(tree_t t) {
  tree_destr_t res;

  res.label = t.label;
  if(res.kind = t.kind == NODE) {
    res.d1 = (dia_t)t.left;
    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* $\diamond$s in order to build a tree node. In exchange, two $\diamond$s will become available when we decompose a tree.

We can now define a function `list_t labels(dia_t d, tree_t t)` which computes the list of labels of a tree in the depth first order. Notice that we must provide one `dia_t` argument to balance the definition.

```
list_t labels(tree_t t0) {
  tree_destr_t t = tree_destr(t0);

  return
  t.kind == LEAF ?
    cons(d,t.label,nil()) :
    append(labels(t.d1,t.left),
      cons(d,t.label,labels(t.d2,t.right)));
}
```

We omit the definition of `list_t append(list_t l, list_t r)`.

Of course, in the presence of both lists and trees we must change the definitions of `new()` and `disp()` to something like

```
typedef union {
  list_t a;
  tree_t b;
} max_t;

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

void disp(dia_t d){
  (max_t *)free(d);
}
```

3 A linear functional programming language

We will now introduce a linearly typed functional metalanguage 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 [12]). 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 metalanguage 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 zero-order types are given by the following grammar.

$$A ::= \mathsf{int} \mid \diamond \mid \mathsf{list}(A) \mid \mathsf{tree}(A) \mid A_1 \otimes A_2 \mid A_1 + A_2$$

A first-order type is an expression of the form

$$T = (A_1, \ldots, A_n)B$$

where $A_1 \ldots A_n, B$ are zero-order types.

A signature Σ is a partial 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 $\Gamma, x\colon A$ 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 of a rule below it is implicitly understood that these disjointness conditions are met.

Types not including $\mathsf{list}(-), \mathsf{tree}(-), \diamond$ are called *heap-free*, e.g. int and $\mathsf{int} \otimes \mathsf{int}$ are heap-free.

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

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \quad \text{(VAR)}$$

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

$$\frac{\Gamma, x\colon A, y\colon A \vdash_\Sigma e : B \qquad A \text{ heap-free}}{\Gamma, x\colon A \vdash_\Sigma e[x/y] : B} \quad \text{(CONTR)}$$

$$\frac{c \text{ a } \mathsf{C} \text{ integer constant}}{\Gamma \vdash_\Sigma c : \mathsf{int}} \quad \text{(CONST)}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{int} \qquad \Delta \vdash_\Sigma e_2 : \mathsf{int} \quad \star \text{ a } \mathsf{C} \text{ infix opn.}}{\Gamma, \Delta \vdash e_1 \star e_2 : \mathsf{int}} \quad \text{(INFIX)}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{int} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, e'' : A} \quad \text{(IF)}$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \quad \text{(PAIR)}$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}\, A, y{:}\, B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\, e \,\mathsf{with}\, x \otimes y \Rightarrow e' : C} \quad \text{(SPLIT)}$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{inl}_{A,B} : A + B} \quad \text{(INL)}$$

$$\frac{\Gamma \vdash_\Sigma e : B}{\Gamma \vdash_\Sigma \mathsf{inr}_{A,B} : A + B} \quad \text{(INR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e : A + B \\ \Delta, x{:}\, A \vdash_\Sigma e' : C \qquad \Delta, x{:}\, B \vdash_\Sigma e'' : C\end{array}}{\Gamma \vdash_\Sigma \mathsf{match}\, e \,\mathsf{with}\, \mathsf{inl}(x) \Rightarrow e' \mid \mathsf{inr}(x) \Rightarrow e'' : C} \quad \text{(SUM-ELIM)}$$

$$\Gamma \vdash_\Sigma \mathsf{nil}_A : \mathsf{list}(A) \quad \text{(NIL)}$$

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

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

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

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

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

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.

As usual, we omit type annotations wherever possible. 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 SUM-ELIM. Here is how we typecheck $x + x$ when $x{:}\,\mathsf{int}$. First, we have $x{:}\,\mathsf{int} \vdash x : \mathsf{int}$ and $y{:}\,\mathsf{int} \vdash y : \mathsf{int}$ by VAR. Then $x{:}\,\mathsf{int}, y{:}\,\mathsf{int} \vdash x{+}y : \mathsf{int}$ by INFIX and finally $x{:}\,\mathsf{int} \vdash x{+}x$ by rule CONTR. It follows by standard type-theoretic techniques that typechecking for this system is decidable in linear time.

Programs A *program* consists of a signature Σ and for each symbol $f : (A_1, \ldots, A_n)B$ a term

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

3.2 Set-theoretic interpretation

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

$$\begin{array}{l}[\![\mathsf{int}]\!] = \mathbf{Z} \\ [\![\Diamond]\!] = \{0\} \\ [\![\mathsf{list}(A)]\!] = \text{finite lists over } [\![A]\!] \\ [\![\mathsf{tree}(A)]\!] = \text{binary } [\![A]\!]\text{-labelled trees} \\ [\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!] \\ [\![A + B]\!] = \{\mathsf{inl}(a) \mid a \in [\![A]\!]\} \cup \{\mathsf{inr}(b) \mid b \in [\![B]\!]\} \\ [\![(A_1, \ldots, A_n)B]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!] \rightharpoonup [\![B]\!]\end{array}$$

Here $X \rightharpoonup Y$ is the set of partial functions from X to Y.

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 $f \in \mathrm{dom}(\Sigma)$. It is valid if it interprets the constructors and destructors for lists and trees by the eponymous set-theoretic operations

To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ (i.e., formally, we define a partial

function from valuations to values) 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.

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

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

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

where $\eta(x_i) = v_i$.

This definition makes sense because—as is readily seen—$\rho \mapsto [\![e]\!]_{\eta,\rho}$ is a monotone functional.

3.3 Examples

Reverse:

$$
\begin{array}{l}
\mathtt{rev_aux} : (\mathsf{list}(\mathsf{int}),\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \\
\mathtt{reverse} : (\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \\
e_{rev_aux}(l,\mathit{acc}) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow \mathit{acc} \\
\qquad \mid \mathsf{cons}(d,h,t)\Rightarrow \mathtt{rev_aux}(t,\mathsf{cons}(d,h,\mathit{acc})) \\
e_{reverse}(l) = \mathtt{rev_aux}(l,\mathsf{nil}_{\mathsf{int}})
\end{array}
$$

Flattening

$$
\begin{array}{l}
\mathtt{append} : (\mathsf{list}(\mathsf{int}),\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \\
\mathtt{flatten} : (\mathsf{list}(\mathsf{list}(\mathsf{int})))\mathsf{list}(\mathsf{int}) \\
e_{\mathtt{append}}(l,r) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow r \\
\qquad \mid \mathsf{cons}(d,h,t)\Rightarrow \mathsf{cons}(d,h,\mathtt{append}(t,r)) \\
e_{\mathtt{flatten}}(l) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}()\Rightarrow \mathsf{nil}() \\
\qquad \mid \mathsf{cons}(d,h,t) = \mathtt{append}(h,\mathtt{flatten}(t))
\end{array}
$$

This example presents a waste as the $\Diamond$ d is not used.

Quicksort:

$$
\begin{array}{l}
\mathtt{append} : (\mathsf{list}(\mathsf{int}),\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \\
\mathtt{split} : (\mathsf{int},\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \otimes \mathsf{list}(\mathsf{int}) \\
\mathtt{sort} : (\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int}) \\
\\
e_{\mathtt{append}}(l,r) = \text{as above} \\
e_{\mathtt{split}}(p,l) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow \mathsf{nil}\otimes\mathsf{nil} \\
\qquad \mid \mathsf{cons}(d,h,t)\Rightarrow \mathsf{match}\ t\ \mathsf{with} \\
\qquad\quad u\otimes v\Rightarrow \mathsf{if}\ h{<}{=}p \\
\qquad\qquad \mathsf{then}\ \mathsf{cons}(d,h,u)\otimes v \\
\qquad\qquad \mathsf{else}\ u\otimes\mathsf{cons}(d,h,v) \\
\\
e_{\mathtt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow \mathsf{nil} \\
\qquad \mid \mathsf{cons}(d,p,t)\Rightarrow \mathsf{match}\ \mathtt{split}(h,t)\ \mathsf{with} \\
\qquad\quad u\otimes v\Rightarrow \mathtt{append}(\mathtt{sort}(u),\mathsf{cons}(d,p,\mathtt{sort}(v)))
\end{array}
$$

Breadth-first search

$$
\begin{array}{l}
\mathtt{snoc} : (\Diamond,\mathsf{list}(\mathsf{tree}(\mathsf{int})),\mathsf{tree}(\mathsf{int}))\mathsf{list}(\mathsf{tree}(\mathsf{int})) \\
\mathtt{breadth} : (\mathsf{list}(\mathsf{tree}(\mathsf{int})))\mathsf{list}(\mathsf{int}) \\
e_{\mathtt{snoc}}(d,l,t) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow \mathsf{cons}(d,t,\mathsf{nil}()) \\
\qquad \mid \mathsf{cons}(d',t',q)\Rightarrow \mathsf{cons}(d',t,\mathtt{snoc}(d,q,t)) \\
e_{\mathtt{breadth}}(q) = \mathsf{match}\ q\ \mathsf{with} \\
\qquad \mathsf{nil}\Rightarrow \mathsf{nil} \\
\qquad \mid \mathsf{cons}(d,t,q) = \mathsf{match}\ t\ \mathsf{with} \\
\qquad\quad \mathsf{leaf}(a)\Rightarrow \mathsf{cons}(d,a,\mathtt{breadth}(q)) \\
\qquad\quad \mathsf{node}(d_1,d_2,a,l,r)\Rightarrow \mathsf{cons}(d,a, \\
\qquad\qquad \mathtt{breadth}(\mathtt{snoc}(d_2,\mathtt{snoc}(d_1,q,l),r)))
\end{array}
$$

Remark 3.1 *It can be shown that all definable functions are non-increasing, e.g. if* $f : (\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int})$ *then* $|\rho(f)|(l) \le |l|$. *This would not be the case if we would omit the* $\Diamond$ *argument in* cons *even if we keep linearity. We could then define for instance the function* $f(l) = \mathsf{cons}(0,l)$.

3.4 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 $\mathrm{types}(P)$ be the (finite) set of zero-order types mentioned in P. To each $A \in \mathrm{types}(P)$ we assign a unique identifier $\nu(A)$. Next, for each $A \in \mathrm{types}(P)$ we introduce a piece of code $[\![A]\!]^{\mathsf{C}}$ defining a C type $\nu(A)$ and some functions associated with A.

The code $[\![\Diamond]\!]^{\mathsf{C}}$ is: `typedef void * ν(◇);`

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

```
typedef int ν(int);
```

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

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

```
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);
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 $[\![\mathsf{list}(A)]\!]^{\mathsf{C}}$ is:

```
typedef struct {
  kind_t kind;
  ν(A) hd;
  void * tl;
} ν(list(A));
ν(list(A)) ν(list(A))_nil(void){
  ν(list(A)) res;
  res.kind = NIL;
  return res;
}
ν(list(A)) ν(list(A))_cons
(ν(◇) d, ν(A) hd, ν(list(A)) tl){
  ν(list(A)) res;
  res.kind = CONS;
  res.hd = hd;
  *(ν(list(A)) *)d = tl;
  res.tl = d;
  return res;
}
```

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

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

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

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

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

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

```
ν(B) f(ν(A1) x1, ..., ν(An) xn)
```

The interpretation of the signature $[\![\Sigma]\!]^{\mathsf{C}}$ is given as the concatenation of $[\![f]\!]^{\mathsf{C}}$; (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 η satisfying Γ we associate a C expression $[\![e]\!]^{\mathsf{C}}_\eta$ of type $\nu(B)$. The definition is by induction on typed expressions as follows:

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

$$[\![f(e_1, \ldots, e_n) : A]\!]^{\mathsf{C}}_\eta = f\,([\![e_1]\!]^{\mathsf{C}}_\eta,\ \ldots,\ [\![e_n]\!]^{\mathsf{C}}_\eta)$$

$$[\![c : \mathsf{int}]\!]^{\mathsf{C}}_\eta = c$$

$$[\![e_1 \star e_2 : \mathsf{int}]\!]^{\mathsf{C}}_\eta = ([\![e_1]\!]^{\mathsf{C}}_\eta \star [\![e_2]\!]^{\mathsf{C}})$$

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

$$[\![e \otimes e' : A \otimes B]\!]^{\mathsf{C}}_\eta = \nu(A \otimes B)\texttt{_pair}([\![e]\!]^{\mathsf{C}}_\eta,\ [\![e']\!]^{\mathsf{C}}_\eta)$$

$$[\![\mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e']\!]^{\mathsf{C}}_\eta = [\![e']\!]_{\eta'}$$
where $\eta' = \eta[x \mapsto [\![e]\!]_\eta\texttt{.fst}][y \mapsto [\![e]\!]_\eta\texttt{.snd}]$

$$[\![\mathsf{in}|_{A,B}(e) : A + B]\!]^{\mathsf{C}}_\eta \nu(A + B)\texttt{_inl}([\![e]\!]_\eta)$$

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

$$[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x)\Rightarrow e' \mid \mathsf{inr}(x)\Rightarrow e'' : C]\!]^{\mathtt{C}} =$$
$$(\,[\![e]\!]_{\eta}\mathtt{.kind==INL\ ?}$$
$$[\![e']\!]_{\eta[x\mapsto [\![e]\!]_{\eta}\mathtt{.body.inl}]}\ :$$
$$[\![e'']\!]_{\eta[x\mapsto [\![e]\!]_{\eta}\mathtt{.body.inr}]}\,)$$

$$[\![\mathsf{nil}_A : \mathsf{list}(A)]\!]^{\mathtt{C}}_{\eta} = \nu(\mathsf{list}(A))_\mathtt{nil}\,()$$

$$[\![\mathsf{cons}(e_d, e_h, e_t)]\!]^{\mathtt{C}}_{\eta} = \nu(\mathsf{list}(A))_\mathtt{cons}\,([\![e_d]\!]^{\mathtt{C}}_{\eta}, [\![e_h]\!]^{\mathtt{C}}_{\eta}, [\![e_t]\!]^{\mathtt{C}}_{\eta})$$

$$[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil}\Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d,h,t)\Rightarrow e_{\mathsf{cons}} : B]\!]^{\mathtt{C}}_{\eta} =$$
$$([\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.kind==NIL\ ?}\ [\![e_{\mathsf{nil}}]\!]^{\mathtt{C}}_{\eta}\ :\ [\![e_{\mathsf{cons}}]\!]^{\mathtt{C}}_{\eta'})$$
where $\eta' = \eta[d \mapsto [\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.tl}, h \mapsto [\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.hd},$
$t \mapsto \mathtt{*}\,(\nu(\mathsf{list}(A))\,\mathtt{*})\,[\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.tl}]$

$$[\![\mathsf{leaf}(e) : \mathsf{tree}(A)]\!]^{\mathtt{C}}_{\eta} = \nu(\mathsf{tree}(A))_\mathtt{leaf}\,([\![e]\!]^{\mathtt{C}}_{\eta})$$

$$[\![\mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r)]\!]^{\mathtt{C}}_{\eta} =$$
$$\nu(\mathsf{tree}(A))_\mathtt{node}\,([\![e_{d1}]\!]^{\mathtt{C}}_{\eta}, [\![e_{d2}]\!]^{\mathtt{C}}_{\eta}, [\![e_a]\!]^{\mathtt{C}}_{\eta}, [\![e_l]\!]^{\mathtt{C}}_{\eta}, [\![e_r]\!]^{\mathtt{C}}_{\eta})$$

$$[\![\ \mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a)\Rightarrow e_{\mathsf{leaf}}\ ]\!]^{\mathtt{C}}_{\eta} =$$
$$\mid \mathsf{node}(d_1, d_2, a, l, r)\Rightarrow e_{\mathsf{node}} : B$$
$$([\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.kind==NIL\ ?}\ [\![e_{\mathsf{leaf}}]\!]^{\mathtt{C}}_{\eta[a\mapsto [\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.label}]}\ :\ [\![e_{\mathsf{node}}]\!]^{\mathtt{C}}_{\eta'}$$
where $\eta' = \eta[d_1 \mapsto [\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.left}, d_2 \mapsto [\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.right},$
$a \mapsto [\![e]\!]^{\mathtt{C}}\mathtt{.label},$
$l \mapsto \mathtt{*}\,(\nu(\mathsf{tree}(A))\,\mathtt{*})\,[\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.left},$
$r \mapsto \mathtt{*}\,(\nu(\mathsf{tree}(A))\,\mathtt{*})\,[\![e]\!]^{\mathtt{C}}_{\eta}\mathtt{.left}]$

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

i. the interpretation of the types: $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$,

ii. the function prototypes: $[\![\Sigma]\!]^{\mathtt{C}}$,

iii. for each $f \in \mathrm{dom}(\Sigma)$ a function definition
```
[[f]]^C {
  return [[e]]^C_{x1↦x1,...x1↦x1};
}
```

3.5 Correctness of the translation

The following is proved by induction on typing derivations.

Lemma 3.2 *If $\Gamma \vdash_\Sigma e : A$ and $\eta(x)$ is a* C*-expression of type $\nu(\Gamma(x))$ for each $x \in \mathrm{dom}(\Gamma)$ then $[\![\Gamma]\!]^{\mathtt{C}}_{\eta}$ is a* C*-expression of type $\nu(A)$ when it appears after the declarations $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$ and $[\![\Sigma]\!]^{\mathtt{C}}$.*

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. We now want to show that the computed functions agree with the set-theoretic semantics.

Definition 3.3 For each zero-order type A the set $\mathcal{V}(A)$ is given by the set of pairs (v, H) where v is a C-stack-value of type $\nu(A)$ (under the definitions $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$) and H is a region in the heap (a set of addresses).

A relation $\Vdash_A \subseteq \mathcal{V}(A) \times [\![A]\!]$ is defined inductively as follows:

- $(n, \emptyset) \Vdash_{\mathsf{int}} n'$, if n encodes n'
- $(p, H) \Vdash_{\diamond} 0$, if H is a contiguous region of size $\max\{\mathtt{sizeof}\,(\nu(A)) \mid A \in \mathrm{types}(P)\}$ and p points to the beginning of H.
- $(v, H) \Vdash_{A\otimes B} (a, b)$ if $H = H_1 \uplus H_2$ and $v\mathtt{.fst}, H_1 \Vdash_A a$ and $v\mathtt{.snd}, H_2 \Vdash_B b$.
- $(v, H) \Vdash_{A+B} \mathsf{inl}(a)$ if $v\mathtt{.kind} = \mathtt{INL}$ and $(v\mathtt{.body.inl}, H) \Vdash_A a$,
- $(v, H) \Vdash_{A+B} \mathsf{inr}(b)$ if $v\mathtt{.kind} = \mathtt{INR}$ and $(v\mathtt{.body.inr}, H) \Vdash_B b$,
- $(v, \emptyset) \Vdash_{\mathsf{list}(A)} \mathsf{nil}$ if $v\mathtt{.kind} = \mathtt{NIL}$.
- $(v, H) \Vdash_{\mathsf{list}(A)} \mathsf{cons}(h, t)$, if $v\mathtt{.kind} = \mathtt{CONS}$ and $H = H_d \uplus H_h \uplus H_t$ and $(v\mathtt{.tl}, H_d) \Vdash_{\diamond} 0$ and $(v\mathtt{.hd}, H_t) \Vdash_A h$ and $(\mathtt{*}\,(\nu(\mathsf{list}(A))\,\mathtt{*})\,v\mathtt{.tl}, H_t) \Vdash_{\mathsf{list}(A)} t$,
- $(v, H) \Vdash_{\mathsf{tree}(A)} \mathsf{leaf}(a)$ if $v\mathtt{.kind} = \mathtt{LEAF}$ and $(v\mathtt{.label}, H) \Vdash_A a$,
- $(v, H) \Vdash_{\mathsf{tree}(A)} \mathsf{node}(a, l, r)$ if $v\mathtt{.kind} = \mathtt{NODE}$ and $H = H_{d1} \uplus H_{d2} \uplus H_a \uplus H_l \uplus H_r$ and $(v\mathtt{.left}, H_{d1}) \Vdash_{\diamond} 0$ and $(v\mathtt{.right}, H_{d2}) \Vdash_{\diamond} 0$ and $(v\mathtt{.label}, H_a) \Vdash_A a$ and $(\mathtt{*}\,(\nu(\mathsf{tree}(A))\,\mathtt{*})\,v\mathtt{.left}, H_l) \Vdash_{\mathsf{tree}(A)} l$ and $(\mathtt{*}\,(\nu(\mathsf{tree}(A))\,\mathtt{*})\,\mathtt{v.right}, H_r) \Vdash_{\mathsf{tree}(A)} r$

Notice that whenever A is heap-free and $(v, H) \Vdash_A a$ for some a then $H = \emptyset$.

Theorem 3.4 *Assume the following:*

- *a program $P = (\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$,*
- *a well typed expression $\Gamma \vdash_\Sigma e : A$,*
- *for each $x \in \Gamma$ a value $(v_x, H_x) \in \mathcal{V}(\Gamma(x))$ such that $H_x \cap H_y = \emptyset$ whenever $x \neq y$,*
- *a set-theoretic valuation η of Γ such that $(v_x, H_x) \Vdash_{\Gamma(x)} \eta(x)$ for each $x \in \mathrm{dom}(\Gamma)$,*

Let ρ be the set-theoretic interpretation of P.

Then the evaluation of $[\![e]\!]^{\mathtt{C}}_{[x_1\mapsto x_1,\ldots,x_n\mapsto x_n]}$ in a runtime environment which maps $x \in \mathrm{dom}(\Gamma)$ to v_x will result in a value v such that $(v, H) \Vdash_A [\![e]\!]_{\eta,\rho}$ for some subset $H \subseteq \bigcup_{x\in\mathrm{dom}(\Gamma)} H_x$ and moreover the part of the heap outside of $\bigcup_{x\in\mathrm{dom}(\Gamma)} H_x$ will be left unaffected by the evaluation.

Proof. Straightforward lexicographic induction on evaluation time and length of typing derivations. Details are omitted for lack of space. □

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

4 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 this is often not needed. Sometimes, however, dynamic allocation and deallocation may be required and to this end we can introduce functions $\mathsf{new} : ()\Diamond$ and $\mathsf{disp}(\Diamond)\mathsf{int}$ whose interpretation is given by C-functions

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

here 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 list containing one entry for each depth level of the list. 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.

Nondestructive computation, I/O We can include functions `print_int` : $(\mathsf{int})\mathsf{int}$ and `read_int` : $()\mathsf{int}$ 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 $(\mathsf{list}(A))\mathsf{list}(A)$ 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., `depth` : $(\mathsf{list}(A))\mathsf{int} \otimes \mathsf{list}(A)$, 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 [17, 16] 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 and nonlinear. 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. The advantage of such an extension would be that schematic programs such as `append` for arbitrary type or `maplist` for arbitrary function would need to be written only once.

A prototype implementation of such an extension is underway.

If we were to allow recursion with not necessarily first-order type X then we would have to hold intermediate functional terms 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. Consider, e.g., the following definition of a functional $F : \mathsf{list}(\mathsf{int}) \multimap (\mathsf{list}(\mathsf{int}) \multimap \mathsf{list}(\mathsf{int})) \multimap \mathsf{list}(\mathsf{int}) \multimap \mathsf{list}(\mathsf{int})$ where $f, g : \mathsf{list}(\mathsf{int}) \multimap \mathsf{list}(\mathsf{int})$ are function symbols.

$$F(l,u) = \mathsf{if}\ a(l)\ \mathsf{then}\ F(b(l), \lambda x.f(u(x)))\ \mathsf{else}\ F(c(l), \lambda x.u(g(x)))$$

It is clear that for appropriate choice of a, b, c the evaluation of $F(l_1, \lambda x.x)(l_2)$ will require to store an arbitrary large lambda expression involving f, g on the heap. One could consider charging a $\Diamond$ for the construction of a lambda term, but that would destroy structural properties such as currying and uncurrying being an isomorphism and thus would require further research.

Queues The program for breadth-first search in the examples section is very inefficient for various reasons. One of them lies in the linear runtime of `snoc`. A seasoned C-programmer would implement breadth-first search using linked lists which include a pointer to their end. Enqueuing and also appending can then be performed in constant time.

In our language we can achieve this effect very easily by adding a new type former $\mathsf{queue}(A)$ which is interpreted as:

```
typedef struct {
  ν(list(A)) qu;
  ν(list(A)) * end;
} ν(queue(A));
```

We introduce (formally with typing rules) basic functions:

$$\begin{aligned}
&\mathsf{nil} : ()\mathsf{queue}(A)\\
&\mathsf{cons} : (\Diamond, A, \mathsf{queue}(A))\mathsf{queue}(A)\\
&\mathsf{snoc} : (\Diamond, \mathsf{queue}(A), A)\mathsf{queue}(A)\\
&\mathsf{append} : (\mathsf{queue}(A), \mathsf{queue}(A))\mathsf{queue}(A)\\
&\mathsf{dequeue} : (\mathsf{queue}(A))\mathsf{int} + A \otimes \mathsf{queue}(A)
\end{aligned}$$

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

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 Even with real queues in place the program for breadth-first search is still rather inefficient as the form of its recursion requires a runtime stack of size proportional to the input tree. An iterative program would require no stack at all.

A straightforward solution would be to recognise tail recursive definitions and to translate them into iterative C-code. For

instance, the code for `rev_aux` or a tail recursive equivalent to `breadth` would then be mapped to iterative code.

More challenging would be some automatic program transformation which translates the existing definition of `breadth` and similar functions into iterative code. To what extent this can be done systematically remains to be seen. It seems that at least for linear recursion (only one recursive call) such transformation might always be possible.

If we wanted to keep stack size small for instance in order to generate code to be executed on an embedded system we would simply have to write tail recursive (or linear recursive perhaps) functions only. The possibility of using equational reasoning for specification and verification remains unaffected by this restriction.

Expressivity In order to study complexity-theoretic expressivity it seems to be a reasonable abstraction to view the type int as finite, e.g. the set of 32 bit words, and to view the heap as infinite. In this case, we have the following expressivity result:

Theorem 4.1 *If $f : \mathbb{N}\to\mathbb{N}$ is a non-increasing function computable in linear (in $\log(n)$) space then there exists a program containing a symbol $\mathtt{f} : (\mathsf{list}(\mathsf{int}))\mathsf{list}(\mathsf{int})$ such that $[\![\mathtt{f}]\!](\mathtt{u}(\mathtt{x})) = \mathtt{u}(\mathtt{f}(\mathtt{x}))$ when $u : \mathbb{N}\to\{0,1\}^*$ is an encoding of natural numbers as lists of 0s and 1s.*

Proof. By encoding all the worktapes as a single list over a larger alphabet. See [8] for details. □

If we restrict to a tail recursive fragment then programs can also be evaluated in linear space so that we obtain a characterisation.

Recursive types We can extend the type system and the compilation technique to arbitrary (even nested) first-order recursive types. To that end, we would 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

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

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

The translation of such definitions into `C` would be essentially straightforward but would require a formalisation of mapping a function through a type operator ("strength"). Such a construct would allow one to, e.g., define trees as $\mu X.A + A \otimes X \otimes X$ and the presence of two $\diamond$-arguments in the typing of node would then arise as a special case of the typing of fold.

Such recursive types would be most useful in conjunction with (variant) records and pattern matching which could also be added to the language.

Circular data structures We have seen that the (forbidden) multiple usage of $\diamond$-values leads to circular data structures. It might be worth investigating whether this can be turned into a feature allowing functional manipulation of circular data. At present, however, we have no indication as to whether this could be successful.

5 Experiments

The above example programs have been "hand-compiled" into `C` with the aid of a prototype implementation which generates $[\![\mathrm{types}(P)]\!]^{\mathtt{C}}$. The resulting code was then compared with genuine functional programs. As should be expected it vastly outperforms interpreted and also bytecode-compiled ML code (NJ-SML, OCAMLC) [14]. Somewhat surprisingly, ML-code compiled with the native code compiler OCAMLOPT performs better than the `C`-code both in terms of runtime and memory usage. I do not know at present whether this effect is due to a static analysis (which then would probably be similar to the one proposed in this paper) or whether it's simply the efficiency of the garbage collection and clever use of caching.

More experiments with the ML KIT compiler incorporating Talpin-Tofte's [18] region analysis and with the HASKELL compiler are underway.

6 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.

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

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

ii. 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 or in 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 $\mathsf{list}(\diamond \otimes A)$ as opposed to $\mathsf{list}(A)$.

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.

7 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 [18] 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 [19] 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.

Hughes and Pareto's system of sized types [11] annotates list types with their length, e.g. the reversal function would get type $\forall n.\mathsf{list}_n(A) \rightarrow \mathsf{list}_n(A)$. 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).

The relationship between linear types and garbage collection has been recognised as early as '87 by Lafont [13], see also [9, 1, 20, 15]. 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 might also be realised through reference counting (and indeed seem to be by the OCAMLOPT compiler) 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 [8] which in turn was inspired by Caseiro's analysis of recursive equations [5] and bears some remote similarity with Bounded Linear Logic [7]

Mention should also be made of Baker's Linear LISP [2, 3] which bears some similarity to our language. It does not contain the resource type $\diamond$ or a comparable feature thus it is not clear how the size of intermediate data structure is limited, cf. Remark 3.1. Unfortunately, loc. cit. is too brief as to clarify this point.

Other related approaches are *uniqueness types* in Clean [4], linear ADTs and monads [10] which will be compared in the full paper.

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

[8] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*. IEEE, Computer Society Press, 1999. to appear.

[9] 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.

[10] 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.

[11] J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ml programming. In *Proc. ICFP '99*, 1999. to appear.

[12] Kelley and Pohl. *A book on C*.

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

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

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

[16] 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.

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

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

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

[20] D. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 1999. to appear.