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

Martin Hofmann

LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK mxh@dcs.ed.ac.uk

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 [15], the present method provides guaranteed bounds on heap space which will be of use for applications such as languages for embedded systems or 'proof carrying code' [18].

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)
```

[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.

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 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 [15]. For instance,

```
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, in order to develop and verify functions as opposed to concrete computations doing so will largely be unnecessary.

This is 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 6.

While experiments with "hand-compiled" examples 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.

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 [12] for a survey) 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 overwriting such heap space does not affect the result.

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

When invoking a recursive constructor function such as `cons()` or `node()` one must supply an appropriate number of arguments of type $\diamond$ to provide the required heap space. Conversely, if in a recursion 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 engineering effort our system could be turned into a usable programming language for the abovementioned applications.

2 Functional programming with C

Before presenting the language we show how the translated code will look like by way of some direct examples.

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 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 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 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.

Finally, let us 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 = left;
  *(tree_t *)d2 = right;
  res.left = (tree_t *)d1;
  res.right = (tree_t *)d2;
  return res;
}
```

followed by

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

and

```
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 become available when we decompose a tree.

To implement `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` with prototype

```
list_t br_aux(listtree_t l)
```

by essentially mimicking the functional definition above (the complete code is omitted for lack of space) and 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.

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. The full paper shows how this can be accommodated in a controlled fashion. 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`.

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 [13]). 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.

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

More type formers such as sum types, records, and variants can easily be added.

A first-order type is an expression of the form $T = (A_1, \ldots, A_n) \to B$ 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 $\Gamma, x{:}A$ 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 $\mathsf{L}(-), \mathsf{T}(-), \diamond$ 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.

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

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

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

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

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

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

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

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

$$\Gamma \vdash_\Sigma \mathsf{nil}_A : \mathsf{L}(A) \quad (\textsc{Nil})$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \quad (\textsc{Cons})$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\Diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B} \quad (\textsc{List-Elim})$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{leaf}(e) : \mathsf{T}(A)} \quad (\textsc{Leaf})$$

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

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

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 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.

Programs A *program* consists of a signature Σ and for each symbol $f : (A_1, \ldots, A_n) \to B$ 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 purely functional meaning of programs we introduce a set-theoretic interpretation as follows: types are interpreted as sets by

$[\![\mathsf{N}]\!] = \mathbf{Z}$
$[\![\diamond]\!] = \{0\}$
$[\![\mathsf{L}(A)]\!] =$ finite lists over $[\![A]\!]$
$[\![\mathsf{T}(A)]\!] =$ binary $[\![A]\!]$-labelled trees
$[\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!]$

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

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

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 $[\![e]\!]_{\eta,\rho} \in [\![A]\!] \cup \{\bot\}$ 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. 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 $\eta(x_i) = v_i$.

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 our language 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:

$\texttt{rev_aux} : (\mathsf{L}(\mathsf{N}), \mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})$
$\texttt{reverse} : (\mathsf{L}(\mathsf{N})) \to \mathsf{L}(\mathsf{N})$
$e_{\texttt{rev_aux}}(l, acc) = \mathsf{match}\ l\ \mathsf{with}$
$\quad \mathsf{nil} \Rightarrow acc$
$\quad |\mathsf{cons}(d, h, t) \Rightarrow \texttt{rev_aux}(t, \mathsf{cons}(d, h, acc))$
$e_{\texttt{reverse}}(l) = \texttt{rev_aux}(l, \mathsf{nil}_\mathsf{N})$

Insertion sort

$\texttt{insert} : (\diamond, \mathsf{N}, \mathsf{L}(\mathsf{N}))\rightarrow\mathsf{L}(\mathsf{N})$
$\texttt{sort} : (\mathsf{L}(\mathsf{N}))\rightarrow\mathsf{L}(\mathsf{N})$
$e_{\texttt{insert}}(d, a, l) = \mathsf{match}\ l\ \mathsf{with}$
$\quad \mathsf{nil}\Rightarrow\mathsf{nil}$
$\quad |\mathsf{cons}(d', b, l)\Rightarrow\mathsf{if}\ a \leq b$
$\quad\quad \mathsf{then}\ \mathsf{cons}(d, a, \mathsf{cons}(d', b, l))$
$\quad\quad \mathsf{else}\ \mathsf{cons}(d, b, \texttt{insert}(d', b, l))$
$e_{\texttt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with}$
$\quad \mathsf{nil}\Rightarrow\mathsf{nil}$
$\quad |\mathsf{cons}(d, a, l)\Rightarrow\texttt{insert}(d, a, \texttt{sort}(l))$

Breadth-first search

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

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

Remark 31 *It can be shown that all definable functions are non-size-increasing, e.g., if* $f : (\mathsf{L}(\mathsf{N}))\rightarrow\mathsf{L}(\mathsf{N})$ *then, semantically,* $|f(l)| \leq |l|$*. 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.*

3.4 Compilation into C

By following the pattern of the examples in the introduction it is possible to associate a piece of C-code $[\![P]\!]^{\mathtt{C}}$ to each program $P = (\Sigma, (e_f)_{f\in\mathrm{dom}(\Sigma)})$ in such a way that

1. To each zero-order type A occurring in P a unique C identifier $\nu(A)$ is associated and $[\![P]\!]^{\mathtt{C}}$ contains an appropriate type definition of this identifier along with appropriately typed helper functions, e.g. $\nu(A)\texttt{_cons}, \nu(A)\texttt{_list_destr}$ when $A = \mathsf{L}(\dots)$.
2. For each function symbol $f : (A_1, \dots, A_n)\rightarrow B$ defined in P the code $[\![P]\!]^{\mathtt{C}}$ contains a corresponding definition $[\![f]\!]^{\mathtt{C}}$ of a function f with prototype
   $\nu(B)\ \ f\texttt{(}\nu(A_1)\ \ x_1\texttt{, } \dots\texttt{, } \nu(A_n)\ \ x_n\texttt{)}$
3. Whenever $\Gamma \vdash_\Sigma e : A$ then we can exhibit a C expression $[\![e]\!]^{\mathtt{C}}$ of type $\nu(A)$ and involving the identifiers in Γ and in Σ.

The details of this translation are omitted for lack of space; its gist is, however, contained in the examples from the introduction.

3.5 Correctness of the translation

We now have to show that the translation $[\![P]\!]^{\mathtt{C}}$ of a program P computes the partial functions defined by the set-theoretic interpretation ρ of P. Since we have not given all details of the translation we must content ourselves with a sketch of the correctness theorem and its proof which should hopefully allow the inclined reader to reconstruct it in full.

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

For example, an element of $\mathcal{V}(\mathsf{L}(\mathsf{N}))$ consists of a stack-value of

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

i.e., a triple $v = (k, h, t)$ where k, h are (4 byte) integers and t is a memory address together with a set H of memory addresses. This set of memory addresses is meant to, but at this point not required to, comprise all addresses reachable from t by iterated dereferencing.

Next, we inductively define a relation $\Vdash_A \subseteq \mathcal{V}(A) \times [\![A]\!]$ which singles out the values which "implement" or "correspond to" a given semantic value.

- $(n, \emptyset) \Vdash_{\mathsf{N}} n'$, if n encodes n'
- $(p, H) \Vdash_{\diamond} 0$, if H is a contiguous region of size $\max\{\mathtt{sizeof}(\nu(A)) \mid A \text{ occurs in } P\}$ and p points to the beginning of H.
- $(v, H) \Vdash_{A \otimes B} (a, b)$ if $H = H_1 \dot\cup H_2$ and $v.\mathtt{fst}, H_1 \Vdash_A a$ and $v.\mathtt{snd}, H_2 \Vdash_B b$.
- $(v, \emptyset) \Vdash_{\mathsf{L}(A)} \mathsf{nil}$ if $v.\mathtt{kind} = \mathtt{NIL}$.
- $(v, H) \Vdash_{\mathsf{L}(A)} \mathsf{cons}(h, t)$, if $v.\mathtt{kind} = \mathtt{CONS}$ and $H = H_d \dot\cup H_h \dot\cup H_t$ and $(v.\mathtt{tl}, H_d) \Vdash_{\diamond} 0$ and $(v.\mathtt{hd}, H_t) \Vdash_A h$ and $(v.\mathtt{tl}, H_t) \Vdash_{\mathsf{L}(A)} t$,
- $(v, H) \Vdash_{\mathsf{T}(A)} \mathsf{leaf}(a)$ if $v.\mathtt{kind} = \mathtt{LEAF}$ and $(v.\mathtt{label}, H) \Vdash_A a$,
- $(v, H) \Vdash_{\mathsf{T}(A)} \mathsf{node}(a, l, r)$ if $v.\mathtt{kind} = \mathtt{NODE}$ and $H = H_{d1} \dot\cup H_{d2} \dot\cup H_a \dot\cup H_l \dot\cup H_r$ 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 $(v.\mathtt{left}, H_l) \Vdash_{\mathsf{T}(A)} l$ and $(v.\mathtt{right}, H_r) \Vdash_{\mathsf{T}(A)} r$

Here $H = H_1 \dot\cup H_2$ means that $H = H_1 \cup H_2$ and $H_1 \cap H_2 = \emptyset$.

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

Theorem 32 *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 mapping* η *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 $\mathsf{new} : ()\to\diamond$ and $\mathsf{disp} : (\diamond)\to\mathsf{N}$. The full paper explains how these are translated and used.

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 full paper contains a more detailed discussion of this point.

Queues The program for breadth-first search could be made more efficient using queues with constant time enqueuing. We can easily add a type former $\mathsf{Q}(A)$ (and appropriate term formers) which gets translated into linked lists with a pointer to their end. The correctness proof carries over with only minor changes.

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 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 using continuations.

Expressivity In order to study complexity-theoretic expressivity it seems to be a reasonable abstraction to view the type N 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 41 *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 $\mathtt{f} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$ such that $[\![\mathtt{f}]\!](u(x)) = u(\mathtt{f}(x))$ when $u : \mathbb{N} \to \{0,1\}^*$ is an encoding of natural numbers as lists of 0s and 1s.*

Proof. If $f(n)$ is computable in space $c\log(n)$ then we use the type $T = \mathsf{L}(\mathsf{N} \otimes \ldots \otimes \mathsf{N})$ with c factors to store memory configurations. We obtain 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 our system.

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

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

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

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

. 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.

5 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

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, proof-carrying code, 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.

6 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 [19] 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 [20] 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 annotates list types with their length, e.g. the reversal function would get type $\forall n.\mathsf{L}_n(A) \to \mathsf{L}_n(A)$. 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 [7] 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 [14], see also [10, 1, 21, 16]. 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 [9] which in turn was inspired by Caseiro's analysis of recursive equations [5] and bears some remote similarity with Bounded Linear Logic [8]

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 structures is limited, cf. Remark 31. Similar ideas, without explicit mention of linearity are also contained in Mycroft's thesis [17]

Other related approaches are *uniqueness types* in Clean [4], linear ADTs and monads [11] which will be compared in the full paper.

In a seminar talk in Edinburgh, John Reynolds has reported about ongoing work on using linear types for in-place update. At the time of writing there was no conclusive result, though and his attention seems to have since shifted to using linear types for *reasoning* about *shared* heap allocated data structures. This together with a medium depth literature research leads me to believe that the present article is in fact the first to successfully apply linear types to the problem of functional in-place update.

Acknowledgement I would like to thank Samson Abramsky for helpful comments and encouragements. Thanks are also due to Peter Selinger for spotting a shortcoming in an earlier version of this 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. K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symp. Principles of Prog. Lang. (POPL)*. ACM, 2000. to appear.
8. J.-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.
9. Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*. IEEE, Computer Society Press, 1999. to appear.
10. 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.
11. 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.
12. J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ml programming. In *Proc. International Conference on Functional Programming. Paris, September '99.*, 1999. to appear.
13. Kelley and Pohl. *A book on C, third edition*. Benjamin/Cummings, 1995.
14. Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
15. Xavier Leroy. The Objective Caml System, documentation and user's guide. Release 2.02. http://pauillac.inria.fr/ocaml/htmlman, 1999.
16. P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *Proc. LICS 1992, IEEE*, 1992.
17. Alan Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, Univ. Edinburgh, 1981.
18. George Necula. Proof-carrying code. In *Proc. 24th Symp. Principles of Prog. Lang. (POPL)*. ACM, 1997. to appear.
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. Mads Tofte and Lars Birkedal. Region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.
21. D. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 1999. to appear.