Lecture Notes on Heap-Free Functional Programs

15-417/817: HOT Compilation
Frank Pfenning

Lecture 21
April 10, 2025

1 Introduction

Most of the material in this lecture is contained in Hofmann [2000a,b] (with further material in Hofmann [2003]). Hofmann defines a (mostly) affinely-typed functional language with three key properties:

1. Functions are polynomial time
2. Functions are not size-increasing
3. Functions can be implemented imperatively (in C) without using the heap

Furthermore, the language LFPL is expressive enough to express all non-size-increasing polynomial time functions. The techniques have since been refined and applied, for example, for automatic amortized resource analysis (AARA).

2 Primitive Recursion and Iteration

In order for functions to be polynomial time, they also need to be terminating. There is a long history of defining patterns of recursion such that every function is terminating. We review several that are of particular significance.

Primitive Recursion. We first consider functions on natural numbers. The schema of primitive recursion assumes that we have already defined functions g and h and define a new function f by

$$\begin{array}{lcl} f(0, y_1, \ldots, y_n) & = & g(y_1, \ldots, y_n) \\ f(\mathsf{succ}(x), y_1, \ldots, y_n) & = & h(x, f(x, y_1, \ldots, y_n), y_1, \ldots, y_n) \end{array}$$

One can easily show by induction on the first argument that if g and h are terminating, so is f.

It is a bit unpleasant to carry the extra parameters $y_1, \ldots, y_n$. Also, this schema is not particularly linear or affine. For example, both h and f can use x, as well as $y_1, \ldots, y_n$. As a first step we can allow the result type to be *functional* (rather than just the type of natural numbers). This

increases the expressive power of the language considerably, but does not jeopardize termination. Then we could write

$$\begin{array}{lcl} f(0) & = & \lambda y_1. \ldots \lambda y_n. g(y_1, \ldots, y_n) \\ f(\mathsf{succ}(x)) & = & \lambda y_1. \ldots \lambda y_n. h(x, f(x, y_1, \ldots, y_n), y_1, \ldots, y_n) \end{array}$$

Now it is possible to enforce that the function that is returned is affine, so that the y_i cannot be freely used more than once. Therefore, we have the simpler schema

$$\begin{array}{lcl} f(0) & = & g \\ f(\mathsf{succ}(x)) & = & h(x, f(x)) \end{array}$$

If we write the schema as a higher-order function **primrec**, then we can type it as

$$\mathbf{primrec}_A\ (g : A)\ (h : \mathsf{nat} \to A \to A)\ (x : \mathsf{nat}) : A$$

parameterized by an arbitrary type A.

This schema is also significant in the sense that it captures the functions that are extracted from constructive proofs using mathematical induction, because in

$$A(0) \to (\forall x{:}\mathsf{nat}.\, A(x) \to A(\mathsf{succ}(x))) \to \forall x{:}\mathsf{nat}.\, A(x)$$

we see that the proof of $A(0)$ becomes g, the proof of the induction step becomes h, and the final quantification over x just becomes the argument we named x.

However, if we want to program in an affine type system we see that there is still one obstacle: x is used by h and also by the recursive call $f(x)$.

Iteration. We can further simplify the schema to a form of *iteration* that does not give h access to x.

$$\begin{array}{lcl} f(0) & = & g \\ f(\mathsf{succ}(x)) & = & h(f(x)) \end{array}$$

This schema is not affine in g or h, but it is affine in the argument. So as long as g and h are closed and affine, the resulting function f is also affine.

We can enforce this by allowing only top-level definitions to be recursive, and checking the schematic forms of the recursion to be iterations. But we can also define iteration constructs directly as terms [Hofmann, 2003].

These schemas can be uniformly extended to general inductive types. We show it only for lists, because Hoffmann used lists of Boolean's as his representation of binary numbers. Using binary numbers is critical if we want to capture complexity classes of functions accurately.

$$\begin{array}{lcl} f(\mathsf{nil}) & = & g \\ f(\mathsf{cons}(x, l)) & = & h(x, f(l)) \end{array}$$

Here, we give h access to x but not the tail l of the list. Doing so would give use the schema for primitive recursive functionals over lists.

3 Two Small Examples

Before we get to sizes, we can test the usual versions of append and reverse for adherence to the schema of iteration. We have written some redundant parentheses to highlight that the recursive calls are just on the tail of the list, so these are not only linear (and therefore affine) in l but they are overall linear.

```
type A = ...
type list = +{'nil : 1, 'cons : A * list}

defn append (l : list) : list -> list = fun k =>
match l with
| 'nil() => k
| 'cons(hd, tl) => 'cons(hd, (append tl) k)
end

defn rev (l : list) : list -> list = fun acc =>
match l with
| 'nil() => acc
| 'cons(hd, tl) => (rev tl) ('cons(hd, acc))
end

defn reverse (l : list) = (rev l) ('nil())
```

4 Capturing Space

Hofmann's fundamental idea is to introduce a type $\Diamond$ that stands for memory cells. In the context of ND/Sax we can think of it as typing a heap address. When writing code in his source language LFPL, there is no concrete expression that has type $\Diamond$. We can only obtain one by reading some data already in memory. Because data are affine, reading it means that there are no further reference to it, so we can reuse it for new data. This is also the fundamental idea behind reuse in Lecture 6. There, however, we did not make it available to the ND programmer directly but viewed it as a compiler optimization available in Sax. Also, the ND language does not make any wholesale restriction to be affine or use only specific schemas of recursion, so we do not obtain any guarantees as Hofmann obtains in LFPL.

We then enrich the recursive types to account for space by adding a diamond <>.

```
type A = ...
type list = +{'nil : 1, 'cons : <> * A * list}
```

We obtain access to the diamond via a variable introduced by pattern matching which can then be reused to construct new lists. Revisiting the earlier examples:

```
defn append (l : list) : list -> list = fun k =>
match l with
| 'nil() => k
| 'cons(d, hd, tl) => 'cons(d, hd, (append tl) k)     % note d
end

defn rev (l : list) : list -> list = fun acc =>
match l with
| 'nil() => acc
| 'cons(d, hd, tl) => (rev tl) ('cons(d, hd, acc))    % note d
end

defn reverse (l : list) = (rev l) ('nil())
```

We see that the variable d (to be bound to an address) is freed by the match and then used in the construction of a new cons.

There is a small issue here: it looks like we construct the empty list in the definition of `reverse` before calling `rev`. However, this is of fixed size and does not involve any addresses, so the expression is "heap-free" in that it can be allocated on the stack. This is already built into the definition of the list type: a cons requires an address while the empty list does not.

Importantly, we cannot double the elements of a list as follows:

```
defn double (l : list) : list =
match l with
| 'nil() => 'nil()
| 'cons(d, hd, tl) => 'cons(d, hd, 'cons(d, hd, double tl))     % fails
end
```

The problem is two-fold: for one, it is not linear in d, in other words, we do not have enough memory to create two cons cells. It is also not linear in the head of the list. However, if we have binary numbers, then the head of the list should be a Boolean, and Booleans are not recursive. Therefore, Hofmann allows contraction for variable of heap-free type. These are defined inductively from Booleans and pairs; in our case we would also include other non-recursive types like unit and sums.

Fortunately, there is a way around this issue and still double the elements in a list! We have to equip the input list with a sufficient store of diamonds!

```
type bool = +{'false : 1, 'true : 1}    % heap-free
type list_bool = +{'nil : 1, 'cons : <> * bool * list_bool}
type list2_bool = +{'nil : 1, 'cons : <> * (<> * bool) * list_bool}

defn double (l : list2_bool) : list_bool =
match l with
| 'nil() => 'nil()
| 'cons(d1, (d2, hd), tl) => 'cons(d1, hd, 'cons(d2, hd, double tl))
end
```

This now type-checks because we have two variables of type $\Diamond$, and because the type `bool` is heap-free so it doesn't have to be linear.

There is one issue overall: while the functions are guaranteed to be non-size-increasing, how can we ever construct lists to start with? One idea is to give sufficient diamonds to the main function to do that, but a type like $\Diamond^n$ raises some new issues. The other is just to allocate the data on the heap as usual (such functions being checked differently) and being content that the property of being non-size-increasing is confined to the functions.

5 Implementation in C

Hofmann [2000a] gives an implementation of the first-order non-size-increasing functions in C, where diamonds are inhabited by addresses, and values of non-recursive type live on the stack. I recommend you take a look. His programs are malloc-free. In his examples he uses both lists and binary trees, so I have a suspicion his implementation may not be sound if one takes a diamond from a list and uses it to allocated a tree node, unless there is enough space at every heap address for the largest type in the program. Instead of freely casting between pointers, it may also be possible to have different flavors of diamonds.

References

Martin Hofmann. A type system for bounded space and functional in-place update. In G. Smolka, editor, *Proceedings of the European Symposium on Programming (ESOP 2000)*, pages 165–179, Berlin, Germany, March 2000a. Springer LNCS 1782.

Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, November 2000b. URL http://www.cs.cmu.edu/~fp/courses/15417-s25/misc/Hofmann00njc.pdf. A previous version was presented as ESOP 2000.

Martin Hofmann. Linear types and non-size-increasing polynomial type computation. *Information and Computation*, 183(1):57–85, 2003.