Lecture Notes on Optimizations

15-417/817: HOT Compilation
Frank Pfenning

Lecture 6
January 30, 2025

1 Introduction

One important aspect of compilation is *optimization*, which refers to rewriting programs in a semantics-preserving manner so that they execute more efficiently. In Lecture 4 we have already seen two: in Sax, a **cut** composed with the identity **id** as either subterm can be reduced by substitution. While one would be unlikely to write such Sax programs by hand, a straightforward translation from ND will introduce such patterns. Rather then trying to complicate the translations, it is generally a better idea keep translations simple, followed by equally simple transformations. In fact, in the first version of my research compiler for ND I tried to avoid such patterns, leading to unnecessarily complex code that, in the end, couldn't avoid all instances of cut/id.

In today's lecture we cover an optimization that fundamentally relies on linearity (or at least the absence of contraction). The semantics expresses that once we have read a memory cell, we can deallocate it because it can no longer be referenced in the running program. This, by itself, isn't an optimization, simply an aspect of the dynamics of Sax that isn't visible in ND because memory isn't explicit. But instead of deallocating it we can look for opportunities to reuse it in the same lexical context. If we do, the Sax code becomes even more imperative in flavor than it already is. Algorithms mutate data in place instead of frequently allocating and freeing memory as we are used to from functional programs.

For Sax and its variants, this has been investigated by Ng in his Honors Thesis [Ng, 2024], but it is not the first time this idea has been pursued in several forms [Lorenzen et al., 2023, 2024]. There is also a dynamic, opportunistic variant where memory is actually deallocated if a reference count hits zero, but then picked up again by temporally (rather than lexically) close allocation [Reinking et al., 2021] with a similar effect.

2 Vertical Reuse

There are some nontrivial interactions between parallelism and reuse, so we begin with a relatively robust case: *vertical reuse*. We begin with an example.

```
1 type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
2
3 proc inc (d : bin) (x : bin) =
4     read x {
5     | 'b0(y) => write d 'b1(y)
```

```
    | 'b1(y) => cut z : bin
                    call inc z y
                write d 'b0(z)
    | 'e(u) => cut z : bin
                   write z 'e(u)
               write d 'b1(z)
    }
```

There is no allocation in the first branch of the **read**, so let's focus on the second branch. When we encounter the **cut**, we have just read from the address x. This means the memory cell containing the small value of x could be deallocated. Now we observe that, immediately, we are allocating another cell for z of the same type. So instead of deallocating x and allocating a fresh z, we can just reuse x! So the optimized code becomes:

```
proc inc (d : bin) (x : bin) =
    read x {
    | 'b0(y) => write d 'b1(y)
    | 'b1(y) => reuse z = x : bin
                    call inc z y
                write d 'b0(z)
    | 'e(u) => cut z : bin
                   write z 'e(u)
               write d 'b1(z)
    }
```

Now let's shift our attention to the third branch. Again, we are allocating a fresh z after just having read x, so we can turn this into

```
proc inc (d : bin) (x : bin) =
    read x {
    | 'b0(y) => write d 'b1(y)
    | 'b1(y) => reuse z = x : bin
                    call inc z y
                write d 'b0(z)
    | 'e(u) => reuse z = x : bin
                   write z 'e(u)
               write d 'b1(z)
    }
```

Perhaps surprisingly, this code does not perform any allocations at all! In other words, binary increment works (almost) "in place", except possibly for the top-level destination that might have to be allocated before calling `inc`. We see that x is freed in the case of `'b0`, so perhaps it is not quite "in place" unless we could call it with the original source as a destination. That, however, wouldn't work in the other two branches.

The general principle at work is that if we read from an address a then in the scope of that read the address a can be reused. In Sax's somewhat unrealistic memory model, all data take the same amount of space, so all reuse would be permitted. Slightly more robust would be to rule out address of type $\mathbf{1}$, because the runtime system may want to avoid allocating and writing something of that type. Even more robust would be to only reuse addresses at the same type. Because of the way functional algorithms work on linear data, this is surprisingly frequently applicable.

In order to capture this formally, we modify the typing judgment and rules to track, throughout its scope, whether a variable has been used. The additive type checking rules do not actually accomplish that because we collect information when moving back up the abstract syntax tree rather than when going down. For now we maintain the property that every variable lexically in

scope appears in Γ, but we annotate each variable with 0 (not used yet) and 1 (used already). In the latter case it may be reused.

$$\frac{\Gamma, x^1 : \mathbf{1} \vdash P :: \delta \;/\; \Xi}{\Gamma, x^0 : \mathbf{1} \vdash \mathbf{read}\ x\ (\,) \Rightarrow Q :: \delta \;/\; \Xi\ ;\ (x : \mathbf{1})}\ \mathbf{1}L$$

$$\frac{\Gamma, z^1 : A \otimes B, x^0 : A, y^0 : B \vdash Q :: \delta \;/\; \Xi}{\Gamma, z^0 : A \otimes B \vdash \mathbf{read}\ z\ (x, y) \Rightarrow Q :: \delta \;/\; (\Xi \setminus x \setminus y)\ ;\ z : A \otimes B}\ \otimes L$$

$$\frac{(\Gamma, x^1 \oplus \{\ell : A_\ell\}_{\ell \in L}, y_\ell^0 : A_\ell \vdash Q_\ell :: \delta \;/\; \Xi_\ell \quad \Xi = \Xi_\ell \setminus y_\ell) \quad (\forall \ell \in L)}{\Gamma, x^0 : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ x\ \{\ell(y_\ell) \Rightarrow Q_\ell\}_{\ell \in L} :: \delta \;/\; \Xi\ ;\ x : \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus L$$

The identity rule now requires that y has not been used yet.

$$\frac{}{\Gamma, y^0 : A \vdash \mathbf{id}\ x\ y :: (x : A)\ /\ (y : A)}\ \mathsf{id}$$

With the rules so far it should be clear that certain type errors may now be caught a bit earlier in the term traversal of the typechecker, since we propagate some usage information down. The version of cut that allocates remains intact, just notes the fresh variable hasn't been used yet.

$$\frac{\Gamma \vdash P :: (x : A)\ /\ \Xi_1 \quad \Gamma, x^0 : A \vdash Q :: \delta \;/\; \Xi_2}{\Gamma \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta \;/\; \Xi_1\ ;\ (\Xi_2 \setminus x)}\ \mathsf{cut}$$

Now the most interesting rule on reuse.

$$\frac{\Gamma \vdash P :: (x : A)\ /\ \Xi_1 \quad \Gamma, x^0 : A \vdash Q :: \delta \;/\; \Xi_2}{\Gamma, y^1 : A \vdash \mathbf{reuse}\ x = y\ P\ Q :: \delta \;/\; \Xi_1\ ;\ (\Xi_2 \setminus x)}\ \mathsf{cut}$$

It is not clear what should happen to the hypothesis $y^1 : A$. Leaving it as $y^1 : A$ would allow another reuse, even though it is not available. Reverting it to $y^0 : A$ would suggest that it may be used, which is also not correct because it has already been read. We decided simply to remove it, even though it is still lexically in scope. Perhaps there are better solutions?

Do we also need to change the axioms? As for the identity rule, it is clearly possible to do so by requiring that variables have not yet been used.

$$\frac{x^0 : A \in \Gamma, y^0 : B \in \Gamma}{\Gamma \vdash \mathbf{write}\ z\ (x, y) :: z : A \otimes B\ /\ (x : A)\ ;\ (y : B)}\ \otimes X \qquad \frac{}{\Gamma \vdash \mathbf{write}\ x\ (\,) :: (x : \mathbf{1})\ /\ (\cdot)}\ \mathbf{1}X$$

$$\frac{(k \in L) \quad y^0 : A_k \in \Gamma}{\Gamma \vdash \mathbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ /\ (y : A_k)}\ \oplus X$$

3 Another Example: List Reversal in Place for Free

Static reuse applies surprisingly frequently, giving us efficient implementations of algorithms from pure functional program. Reversing a linear list is another example. Consider:

```
1  type list = +{'nil : 1, 'cons : bin * list}
2
3  proc reverse (d : list) (xs : list) (acc : list) =
4    read xs {
5    | 'nil(u) => read u ()
6                 id d acc
7    | 'cons(p) => read p (x, xs)
8                  cut q : bin * list
9                     write q (x, acc)
10                 cut acc1 : list
11                    write acc1 'cons(q)
12                 call reverse d xs acc1
13   }
```

Note that at the cut of q we have just read p of the same type, at the cut of $acc1$ we have read xs earlier, again of the same type. So under our optimization, this code transforms to

```
1  type list = +{'nil : 1, 'cons : bin * list}
2
3  proc reverse (d : list) (xs : list) (acc : list) =
4    read xs {
5    | 'nil(u) => read u ()
6                 id d acc
7    | 'cons(p) => read p (x, tl)
8                  reuse q = p : bin * list
9                     write q (x, acc)
10                 reuse acc1 = xs : list
11                    write acc1 'cons(q)
12                 call reverse d tl acc1
13   }
```

We see there is no actual allocation in this recursive functions, although a single new cell may need to be allocated for the destination before the outermost call to $reverse$.

We may notice something else here, that's not currently part of our research compiler. In the second branch, we know that xs is a address paired with the tag `'cons`. But then we unnecessarily write the tag `'cons` again. Similarly, the first component of the pair at address p is the address x, but then we write x in the same place in line 9. Since writing to memory can be expensive, optimizing aways such unnecessary write operations may be noticeable in terms of overall performance if it happens frequently.

4 Horizontal Reuse

The static reuse optimizations above propagate information about usage into the scope of read operations. However, it is also possible that in a **cut** $(x : A)\ P\ Q$ there is a variable used in P that should then be available for reuse in Q. This can be tracked with a *subtractive* version of resource management during typechecking [Cervesato et al., 2000].

This works fine in a sequential semantics but, as Ng [2024] points out, can lead to a classic deadlock situation under a parallel semantics. The basic observation is that a reuse command for y in Q may have to wait until P actually reads y. Reuse then becomes a possibly blocking command.

We have not run any experiments to assess how much horizontal reuse adds to vertical reuse over a representative set of programs. Both optimizations clearly rely on linearity, or at least on

the type system to be affine so there is at most one reference to any cell.

References

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. FP^2: Fully in-place functional programming. In *International Conference on Functional Programming (ICFP 2023)*, Proceedings on Programming Languages, pages 275–304. ACM, August 2023.

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. The functional essense of imperative binary search trees. In *Programming Language Design and Implementation (PLDI 2024)*, volume 8 of *Proceedings on Programming Languages*, pages 518–542. ACM, January 2024.

Daniel Ng. Memory reuse in linear functional computation. Honors thesis, Carnegie Mellon University, May 2024. URL http://www.cs.cmu.edu/~fp/courses/15417-s25/misc/Ng24.pdf.

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: Garbage free reference counting with reuse. In *42nd International Conference on Programming Language Design and Implementation (PLDI 2021)*, pages 96–111. ACM, June 2021.