Executing/Compiling Equations
CS560 Classs Notes

Alan LaMielle & Andy Stone
scribes for Sanjay Rajopadhye

Consider a formula for the Fibonacci sequence:

$$\text{fib}[i] = \begin{cases} i <= 1 & : 1 \\ i > 1 & : \text{fib}[i-1] + \text{fib}[i-2] \end{cases} \tag{1}$$

$$\text{Out}[] = \text{fib}[n] \tag{2}$$

defined over the domains, $D_{\text{fib}} = \{i \mid 0 \leq i \leq N\}$ and $D_{\text{Out}} = \{ \mid \}$ (i.e., the domain of Out is the empty domain ($\mathbb{Z}^0$)). In other words, Out is a scalar variable.

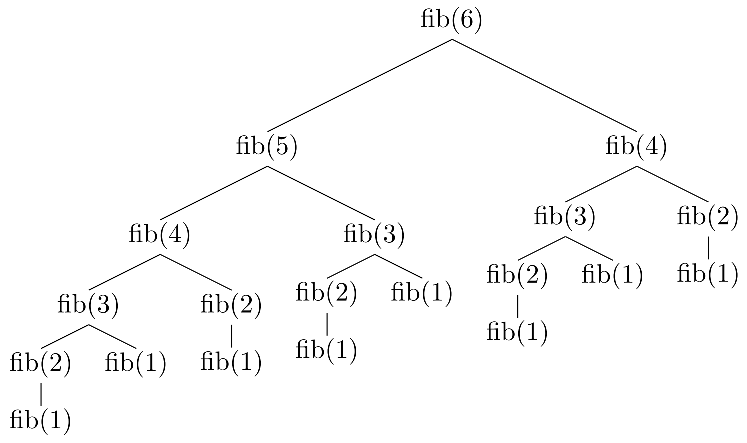
How would one write a program that implements[1] this system of recurrence equations? An obvious approach is through recursion. The following pseudo-code is one possibility:

Algorithm 1 `fib`(int i)

```
if $i <= 1$ then
    return 1
else
    return fib($i-1$)+fib($i-2$)
end if
```

However, there are obvious limitations to this naïve implementation. It has exponential running time, with numerous duplicated calculations. The following tree isllustrates this fact:

[1]Note that there is a closed form solution to Fibonacci: $F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$, so no one would really want to write a program to compute Fibonacci, except for pedagogical purposes.



Notice the repeated computations. For example, fib(3) is called three times and fib(2) is called five times. Thus the purely recursive approach is impractical.

Assuming we are working with the recursive approach, we can certainly do better. Notice from the above Fibonacci call graph that even though we are repeating computations, only n distinct arguments are ever used for fib(n), and thus only n distinct computations are actually needed. Utilizing this fact, we can employ a common technique called *memoization*. Memoization is the act of remembering or "caching" already computed values for use at a later time and is so named because we are keeping a memo of already computed values. Here is pseudo-c code that utilizes this technique:

```
// initialize an array of specified size where all elements
// are set to initialVal, except elements 0 and 1 which are set
// to 1.
int *initializeFibArray(int size, int initialVal)
{
  int *newArray;
  newArray = (int*)malloc(sizeof(int) * size);
  for(int i = 2; i < size; i++) { newArray[i] = initialVal; }
  newArray[0] = newArray[1] = 1;
  return newArray;
}
// memoized fibanocci
int fib(int i)
{
  static const int NOT_COMPUTED = -1;
  static int *cache = initializeFibArray(MAX, NOT_COMPUTED);
  if(cache[i] == NOT_COMPUTED)
    cache[i] = fib(i-1) + fib(i-2);
  return cache[i];
}
```

The basic structure of this code is as follows. Initialize an array of n values to a special value signifying that each value has not yet been computed. In fib(n) check to see if the value for n has been computed. If it has, return the stored value and otherwise compute the value, store it, and return it. We can use this strategy for compiling equations!

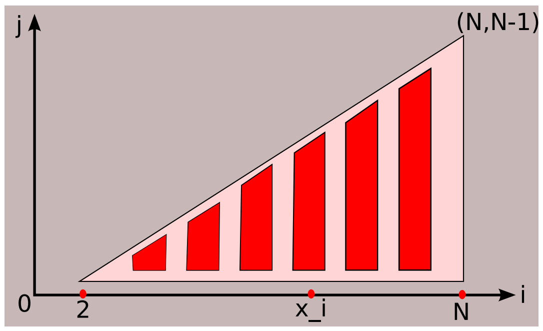


Figure 1: Forward Substitution Slices

As a second example, consider the all to familiar forward substitution formula (find x given L and b in $Lx = b$):

$$x_i = \begin{cases} i = 1 & : b_i \\ i > 1 & : b_i - \sum_{j=1}^{i-1} L_{i,j} x_j \end{cases} \tag{3}$$

How would we go about compiling this equation? We expect to do $O(n^2)$ work since we are computing n values of x over a summation of an average of $\frac{n}{2}$ values. See Figure 1 for a graphical representation of this computation. Each vertical slice of the triangle consists of a number of multiplications ($L_{i,j}x_j$). All of these must be summed. The result of the reduction is stored at x_i.

What might the code that performs this computation look like?

```
// initialize an array of specified size where all elements
// are set to initialVal, except elements 0 and 1 which are set
// to 1.
int *initializeFSArray(int size, int initialVal)
{
  int *newArray;
  newArray = (int*)malloc(sizeof(int) * size);
  for(int i = 2; i < size; i++) { newArray[i] = initialVal; }
  return newArray;
}
// memoized Forward Substitution
int compute_x(int i,int **L, int *b)
{
  static const int NOT_COMPUTED = -1;
  static int *cache = initializeFSArray(MAX, NOT_COMPUTED);
  if(cache[i] == NOT_COMPUTED)
  {
    if(i==1)
      cache[i]=b[i];
    else
    {
      int res=0;
      for(int j=1;j<i;j++)
        res+=L[i,j]*compute_x(j,L,b);
      res=b[i]-res;
      cache[i]=res;
    }
  }
  return cache[i];
}
```

An addition to this may be to mark in the `cache` array that the value is currently *being* computed. This allows us to detect circular dependencies and let the programmer know.

This works for finite domains, but what about infinite domains such as those that occur in signal processing? We cannot allocate a `cache` array in this case (`MAX` is infinite). In this case, do not memoize and incur the cost of repeated evaluations. There may be ways to improve this naïve fallback and we may see them later.

What are the problems with the code generation strategy that utilizes memoization?

1. It is slow. The speed loss comes from a nnumber of reasons:
   - The conditionals in the loop are one source of slowdown.
   - Recursion is expensive: stack frames maintenance is not free and potential context switching comes at a price as well.

   Note that this slowdown is within a constant factor of the total computation (it is manageable: better than an exponential time program for a linear work algorithm, but usually still unacceptable).
2. Memory Hog: The "memory hog" problem could be really severe if the language was not rich enough to express/compile reductions (this was the case with the original definition of Alpha). Moreover, if some of the later analysis questions are difficult/impossible for programs with reductions, then we may be forced to somehow "remove" reductions. MMAlpha has a transformation called `serializeReduce` that does just this.
3. No parallelism.

To better understand the memory problem, let's see how we would write the forward substitution program without reductions. Define a new variable, `Temp`, defined over the domain of the computation.

$$\mathrm{Temp}_{i,j} = \begin{cases} j = 0 & : 0 \\ j > 0 & : \mathrm{Temp}_{1,j-1} + L_{i,j}x_j \end{cases} \tag{4}$$

$$x_i = \begin{cases} i = 1 & : b_i \\ i > 1 & : b_i - \mathrm{Temp}_{i,i-1} \end{cases} \tag{5}$$

The standard memoized program for this will require quadratic storage (for the entire domain of Temp).

Further Optimizations How do we make this computation faster? If we can figure out the "direction" (i.e., a schedule) for the computation, such that dependences in the original program are "respected" (i.e., the schedule is legal), can we increase performance? Yes, if our program execution order follows the schedule, we do not have to check if something was previously computed. We are optimizing away the "demand-driven" computation. Once we have a schedule, we can figure out when a value is no longer needed (based on its lifetime), do a reuse analysis, and only allocate space for the maximum number of simultaneously live values, rather than the whole range of values. Different analyses include scheduling, value reuse, and memory reuse analysis.