REPLACING RECURSION WITH ITERATION

STEFAAN HIMPE, FRANCKY CATTHOOR, GEERT DECONINCK

1. INTRODUCTION[1]

Recursion is often used for elegant specification of algorithms. We are looking for methods to replace the recursion with iteration, to make them more suitable for analysis, parallelisation and optimization. Given the limited length of this abstract, not many details can be given. The interested reader is encouraged to contact the author Stefaan Himpe (`stefaan.himpe@esat.kuleuven.ac.be`) to get more details.

2. MODELING THE ALGORITHM

To enable rigorous mathematical derivation of the transformation from a recursive to an iterative implementation, we need to model the algorithm. The algorithms we have studied, can all be rewritten to fit in the scheme of algorithm 1.

In the scheme of algorithm 1 the following model elements are seen to appear:

- a *non-mutual recursive* function f taking N arguments $\overrightarrow{x} = (x_0, x_1, \ldots, x_{N-1})$
- *base case conditions* bc_i and *base case calculations* bcc_i, both functions optionally taking the N arguments $\overrightarrow{x}$
- *recursive case conditions* rc_i optionally taking the N arguments $\overrightarrow{x}$, and optionally taking M additional arguments $\overrightarrow{y} = (y_0, y_1, \ldots, y_{M-1})$
- *intermediate functions* g_{ij} optionally taking the N arguments $\overrightarrow{x}$
- *argument transformation functions* $\overrightarrow{A_{ij}}(\overrightarrow{x}) = \big(A_{ij0}(\overrightarrow{x}), A_{ij1}(\overrightarrow{x}), \ldots, A_{ij(N-1)}(\overrightarrow{x})\big)$. Each argument transforming function A_{ijk} takes the N arguments $\overrightarrow{x}$, and returns a single number.
- *return values* r_{ij}
- *return value combination functions* cmb_i, taking the B return values from a recursive case as parameter, and returning a result.

The fact that each recursive case rc_i seems to have the same amount of recursive calls to f does not decrease the generality of the approach, because a recursive algorithm with multiple recursive cases can always be rewritten to have a constant number of recursive calls in all recursive cases (proof omitted).

In the most general case, the argument transforming functions, as well as the intermediate functions may have any number of side-effects. After transforming from recursive to iterative form, everything will still work correctly, because the call trace is preserved. As a consequence the implementations of the model elements need not be known. Extra optimizations or trade-offs are made possible if we make some more assumptions, e.g. related to lack of side-effects or independence of result from parameter values.

3. BASIC METHOD

Instead of presenting the mathematics here, we will explain the concepts intuitively. Details of the basic method explained here have been published in [1]. Consider the simple algorithm 2.

The model elements are: $\overrightarrow{x} = (n, x)$, $bc_0(n, x) = (n == 0)$, $rc_0(n, x) = (n \neq 0)$, $A_{00}(n, x) = (n - 1, x + 2)$, $A_{01}(n, x) = (n - 1, 2 \cdot x)$, $A_{02}(n, x) = (n - 1, x + 1)$. There are no return values, no combination functions, no extra arguments $\overrightarrow{y}$, and no intermediate functions g_{ij}.

As can be seen from the algorithm or the call tree, each invocation of function f results in three recursive calls to f (each time numbered 0, 1 and 2 in the call tree), unless the base case is reached. Each arrow in the call tree represents a (recursive) call, and an argument transformation step. During recursive execution of the algorithm, the call tree is traversed in depth-first order, until all leaves have been visited, and the corresponding base case calculations have been executed. The leaves in the figure from algorithm 2 have been numbered in order of

[1] Stefaan Himpe and Geert Deconinck are from K.U. Leuven, Francky Catthoor is from IMEC

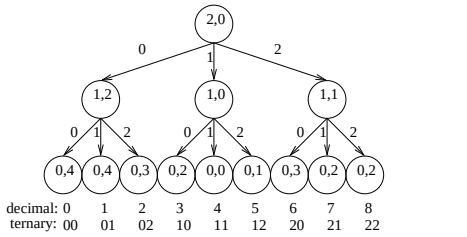
Algorithm 1 Modeling the algorithm

```
f(x⃗){
  if bc_0(x⃗){bcc_0(x⃗)};
  else if bc_1(x⃗){bcc_1(x⃗)};
  else if ...
  else if bc_{P-1}(x⃗){bcc_{P-1}(x⃗)};
  else if rc_0(x⃗){
     g_00(x⃗);
     r_00 = f(A_00(x⃗));
     g_01(x⃗);
     r_01 = f(A_01(x⃗));
      ...
     r_oi = g_0i(x⃗);
     f(A_0i(x⃗));
      ...
     g_0(B-1)(x⃗);
     r_0(B-1) = f(A_0(B-1)(x⃗));
     g_0B(x⃗);
    return cmb_0(r_00, r_01, ..., r_0(B-1));
} else if ...
} else if rc_{Q-1}(x⃗){
      g_(Q-1)0(x⃗);
      r_(Q-1)0 = f(A_(Q-1)0(x⃗));
      g_(Q-1)1(x⃗);
      r_(Q-1)1 = f(A_(Q-1)1(x⃗));
       ...
      g_0(B-1)(x⃗);
      r_(Q-1)(B-1) = f(A_(Q-1)(B-1)(x⃗));
      g_(Q-1)B(x⃗);
     return cmb_{Q-1}(r_(Q-1)0, r_(Q-1)1,
                      ..., r_(Q-1)(B-1));
}
```

Algorithm 2 Simple algorithm

```
f(n,x) {
 if (n==0)
  printf("Base case x=%d",x);
 else {
  f(n-1,x+2);
  f(n-1,2*x);
  f(n-1,x+1);
 }
```

Simple recursive algorithm, with one base case and one recursive case.



Call tree showing argument values for $n = 2$, $x = 0$.

execution. This same numbering is also shown in ternary representation (integers with number base 3, where 3 is the amount of recursive calls in the recursive case).

Observe how the numbering of the leaves, written in ternary representation, define the path (and hence the order of argument transforms) from root node to leaf through the call tree. This observation leads to the following method for making an iterative version: count in number base 3 from 0 to $3^2 - 1 = 8$. (3 is the number of branches in the recursive case, 2 is the depth of the call tree, and hence 3^2 is the number of leaves in the call tree, we number them 0 to $3^2 - 1$). The digits of the ternary representation show what argument transformations must be applied (and in which order) to find the argument values in the base case. This only works correctly if the argument transformation functions have no side-effects.

A second observation is that each leaf can be calculated independently of the others; just take its ternary number representation and apply the correct argument transformation steps given by the digits in the representation. This observation makes this method promising for parallelizing the algorithm afterwards (as long as side-effects of the base case calculations don't inhibit parallelization).

4. EXTENSIONS

The basic method presented in section 3 seems to be of limited use, due to the fact that it was derived from a simplistic example (one recursive case, all leaves live at the same depth in the call tree, no g_{ij} functions, no return values, no side-effects). All of these shortcomings, however, can be overcome by small extensions to the basic method. We have proposed, implemented and tested theoretical and practical extensions to the basic method until the general algorithm 1 could be transformed from a recursive to an equivalent iterative algorithm, *provided that an upper bound for the maximal depth of the recursion can be found at run-time*. The main practical problem that needed to be solved was to keep the overhead of bookkeeping calculations as low as possible.

Theoretical extensions, without further explanation, include:

- handling non constant depth of the call tree leaves
- handling multiple base cases, multiple recursive cases

- handling the intermediate functions g_{ij}(also in the presence of side-effects, or when they need argument values)
- handling return values and combination functions
- handling argument transforming functions with side-effects

5. TRADE-OFFS

While replacing recursion with iteration, one can simply insert a stack data structure, and iteratively "simulate" the recursion. Why then go through all the trouble of what is described in this abstract? Inserting a stack data structure is only one way to transform the recursion to iteration.

By applying our method, it becomes possible to make an iterative equivalent avoiding the storage normally needed for storing the stack frame (under the assumption that the argument transforming functions have no side-effects), but at the expense of up to $\frac{D \cdot B^{D-1}(B-1)}{B^D-1}$ times more argument transformation calculations (proof omitted), where B is the amount of recursive calls in the recursive case, D is the depth of the call tree and the root node has depth 0.

This extra amount of calculations can be exchanged for memory in a systematic way by applying memoization to the argument transforming functions. Memoization is a technique in which you store results of calculations instead of redoing the calculations. This is the first systematic trade-off that can be made: *calculations needed* versus *memory used*. If all results are stored in separate memory locations (static-single assignment), full parallelizability, as mentioned in section 3, is still possible. The amount of memory needed to allow full parallelization equals the amount of nodes in the call tree (times a constant), and becomes quickly too large to be practical.

Full parallelizability, however, is usually not needed, depending on the platform. The amount of memory needed can be systematically reduced (without increasing the amount of argument transforming calculations) at the expense of parallelizability, by applying in-place optimization. The in-place optimization exploits life-times of calculation results, and reduces memory requirements by reusing memory locations. The exploitation of life-times creates dependencies which inhibit full parallelization: calculation results have to be available before you can use them, and should not be used after they have be overwritten. This is a second systematic trade-off that can be made: *memory usage* versus *parallelizability*.

When applying full memoization and full in-place optimization to the iterative implementation of the recursive algorithm, the resource requirements become the same for the iterative version and the recursive version, but the opportunities for parallelization are lost.

6. CONCLUSIONS

We have presented an overview of our recent work on replacing recursion with iteration. At present the reasonably general case from algorithm 1 can be handled. Depending on the assumptions that can be made, systematic trade-offs can be made and additional optimizations can be done.

REFERENCES

[1] Stefaan Himpe, Francky Catthoor, and Geert Deconinck. Control flow analysis for recursion removal. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, Lecture Notes in Computer Science. Springer, September 2003. To appear.