YAGL

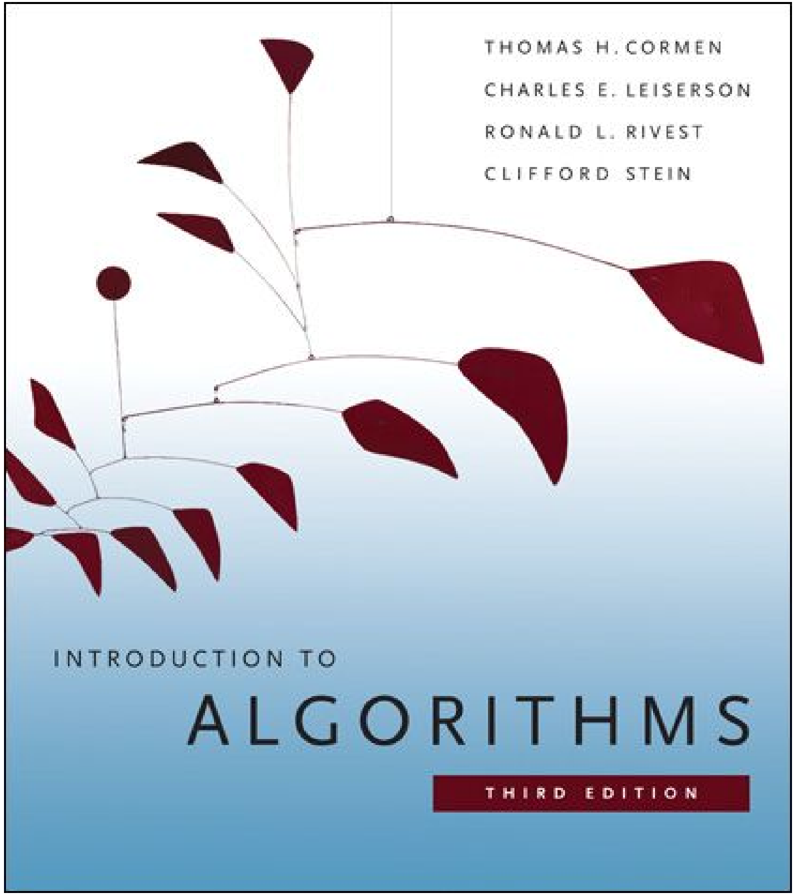
Anthony Alvarez and David Ding

Presentation Outline

1. Motivation
2. Sample Code (Dijkstra’s Algorithm)
3. Unique YAGL syntax
4. How to make this easy for the programmer
5. Aspects of Code Generation
6. Lessons Learned

Motivation

- Understanding graphs is a common skill
- Manipulating them and implementing graph algorithms has lots of overhead
- There has to be a better way!!!!



Dijkstra’s Algorithm from Wikipedia

```
create vertex set Q

for each vertex v in Graph:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q

dist[source] ← 0

while Q is not empty:
    u ← vertex in Q with min dist[u]
    remove u from Q

    for each neighbor v of u:
        alt ← dist[u] + length(u, v)
        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
```

Dijkstra’s Algorithm in YAGL

```
def minDistU( vertices ){
   minVertex = 0
   qCounter = 0
   minDistSoFar = INF
   forKeyValue( i, v, vertices ){
      thisDist = v.dist
      if( thisDist < minDistSoFar ) {
         minDistSoFar = thisDist
         minVertex = qCounter
      }
      qCounter = qCounter + 1
   }
   return( minVertex )
}
```

```
def dijkstras( G, source ){
  Q = []
  forKeyValue( label, v, v( G ) ){
      v.dist = INF
      v.prev = NULL
      append( v(G)[ label ], Q )
  }
  v(G)[source].dist = 0
  while( not isEmpty( Q ) ){
    u = remove( minDistU(Q), Q )
    forKeyValue( i, v, adj( G, u ) ) {
      alt = u.dist + e(G)[ edgeLabel( u, v )].length
      if ( alt < v.dist ) {
        v.dist = alt
        v.prev = u
      }
    }
  }
}
```

Unique YAGL Syntax

Native Alias

```
INF > 0
-INF < 0
false == 0
true == 1
```

Float/Int Interchangeability

```
4/2 == 2.000
1 == 1.0
false == 0.0
true == 1.0
```

Print Returns the value

```
Return( print( 10 ) )
```

Unique YAGL syntax

Map Access

```
a = {| 'key1' := 1, 'key2' := 'two'
|}
a.key3 = ['three']
a.key4 = {| 'key5' := '5' |}
a.key4.key5 = '6'
print( a.key1 )
print( a.key2 )
print( a.key3[0] )
print( a['key4'].key5 )
```

Unique YAGL syntax

Map Access Extends to Graph Property Access

```
forKeyValue( label, v, v( G ) ){
      v.dist = INF
      v.prev = NULL
}


i = 0
forKeyValue( label, edge, e( G ) ){
      edge.weight = i
      edge.capacity = 10
      edge.randomAttribute = []
      i = i + 1
}
```

Unique YAGL Syntax

forKeyValue

```
a = ['zero','one','two']
b = {||}
forKeyValue( k, v, a ) {
  b[v] = k
}
isEqual( b ,{|'zero' := 0, 'one' := 1, 'two' := 2 |}
)
k == 2
v == 'two'
```

How to make this easy for the programmer

- Dynamic Type
    - User does not have to declare type
- Native Graph
    - Native graph type
    - Easy access of vertices & edges
    - Arbitrary attributes possible → very diverse set of capabilities
- Pass by reference to functions
    - Allows functions to behave as they do in CLRS
- Standard Library
    - Standard library encapuslates common use cases isEmpty, enqueue/dequeue, push/pop, deep copy, isIn
    - Frees user to perform higher level operations

Code Generation: Dynamic Typing

- LLVM is not dynamically typed. How does one implement a dynamic language with it?
- Our idea: make all expressions, as well as function arguments and return, have a single type: a struct which contains pointers to the different YAGL types.
  - numbers
  - strings
  - lists
  - maps
  - graphs
- No more than one pointer in a struct should be non-null at any time
- When accessing a struct in an expression, find the pointer that isn't null. If its type isn't valid within the expression, abort with a type error message.
-
  ```
  ERROR: Expected item of type:
  0
  ERROR: found item of type:
  1
  {| 'NULL' := -1, 'Num' := 0, 'String' := 1, 'List' := 2, 'Map' := 3, 'Graph' := 4 |}
  ```

Code Generation: Variable Scope

- Variables in YAGL are scoped within a function, but not within blocks. For example, if a variable is defined in an if/else block, it will still be defined afterwards outside of the while block.
- To generate code for this, we separated every function in LLVM into two main blocks:
    - Variable allocation block, which breaks to the
    - Block for everything else
- Maintain two builders: one for the variable allocation block, and one for everything else. Whenever we see a variable assignment expression for a variable we have not yet seen, allocate space for it in the variable allocation block. Otherwise, use the normal builder.
- In some sense, this is similar to C’s notion of defining all locals in a function first.

Lessons Learned

- Implementing dynamic typing is not easy! Ultimately, it just pushes type checking, which is relatively easy in Ocaml, to LLVM, which makes it a lot harder.
- We should have invested more time into directly writing helper functions like type checkers in C and LLVM, rather than try to generate code for them in Ocaml
- Implementing a compiler in 3-4 weeks is ***really*** hard