mc: A Compiler for the "Mini" Language

CPE431 Final Report / Jacob Hladky / June 8, 2015

Mc is a compiler for the "mini" toy language, targeting the amd64 architecture on OS X and GNU/Linux. Mc is written entirely in SML, and depends on one external SML module: "SML-JSON", which provides callbacks that assist in parsing. Mc produces either an ELF or a MachO object file that is assembled and linked by the GNU assembler and linker.

The program "mc" is in fact a bash script which handles command-line argument parsing and which calls the SML "compiler" binary along with any other required programs. The "compiler" binary is itself compiled with mlton.

Architecture

Mc is input with a source file containing a JSON-representaion of the AST for a mini program, and parses this JSON into an internal AST with the `json2Ast` function. Then it scans through the AST to collect various information about the program. This information is collected into the symbol table using the `mkSymbolTable` function.

The AST and symbol table are then passed into the static checker. The `staticCheck` function runs a series of tests on the program to make sure that it does not violate the semantics of the source language. All type checking is done in this step. The static checker can fail; upon failure the checker prints an error message and compilation stops. After the static checker finishes compilation is guaranteed.

The AST and symbol table are then converted to the ILOC intermediary assembly using the `ast2Iloc` function. The ILOC is then run through several optimizing steps, detailed below. After being optimized the ILOC is converted to amd64 assembly using virtual registers with the `iloc2Amd64` function. Register allocation is then performed by the `regAlloc` function.

This completes the compilation process. The resulting amd64 assembly is outputted using the `programToStr` function. The resulting object file is then assembled and linked.

Internal Structures

AST

The structure of the AST is defined by the JSON input file. The only notable feature of mc's AST is that all nodes preserve line numbers so that they can be presented to the user if there is an error from static analysis.

Symbol Table

The symbol table is one of the most important data structures in mc. It consists of two custom data-types with the following definitions:

The `types` field is used by the static checker to verify type correctness. It is also used when converting the AST to the ILOC to calculate structure access offsets. The `funcs` field is used determine whether it is necessary to add spill space to functions when converting the ILOC to amd64. The remaining fields are only used by the static checker.

```
datatype func_info =
    FUNC_INFO of {
        params: Ast.typ list,
        returnType: Ast.typ,
        calls: string list
    }

datatype symbol_table =
    ST of {
        types: (string, (string, Ast.typ) HashTable.hash_table)
            HashTable.hash_table,
        globals: (string, Ast.typ) HashTable.hash_table,
        funcs: (string, func_info) HashTable.hash_table,
        locals: (string, (string, Ast.typ) HashTable.hash_table)
            HashTable.hash_table
    }
```

Figure 1: Datatypes used in the symbol table.

Control Flow Graph

The control flow graph is implemented generically, and is used to represent the ILOC and the amd64 assembly. It implements several of the higher-order functions, including map, fold, and apply. The CFG significant use of references and thus mutation, and although it purports a functional interface, it is not implemented functionally. This implementation, although ugly, is simpler and solves several issues when optimizing the ILOC.

Interference Graph and Virtual-To-Real map

The interference graph is undirected and was significantly easier to implement. Like the CFG, it supports the higher-order functions, but it is not implemented generically, because it is only used for register allocation. The Virtual-To-Real map is a hash table that is used to keep track of how virtual registers map to real registers. It is used in conjunction with the interference graph during register allocation: the interference graph is "deconstructed" into a stack and "reconstructed" into the virtual-to-real map.

ILOC and Amd64

The ILOC and Amd64 are each written similarly. Each contains an "instruction" datatype and an "opcode" datatype. However the ILOC represents registers as integers, while the amd64 represents them as a separate datatype. This is necessary in order to maintain the virtual/real register distinction.

Optimizations

The following optimizations are implemented by mc:

- Copy Propagation (`copyProp.sml`)
- Local Value Numbering (`lvNumbering.sml`)
- Dead Code Removal (`stripDeadCode.sml`)

The copy propagation and dead code removal optimizations (and the register allocation algorithm) share a common datatype, the `dataflow_analysis` structure (`dfa.sml`). This structure standardizes running the dataflow analysis parts of those algorithms. All functions that use the dfa struct follow the exact same sequence:

1. Take in a CFG representing the function

2. Initialize any global data needed before-hand, e.g. all copies made in the function, or all definitions
3. Map the basic_block CFG to a dataflow_analysis CFG, and pass this in to the `buildDFAs` function, provided by the dfa struct.
4. Map the resulting dataflow_analysis CFG back into a basic_block, in the process the replacement step.

The following block of code from the copyProp module demonstrates this procedure:

```
fun optFunc (id, cfg) =
  let
      val copies = Cfg.fold findCopies (empty ()) cfg
      val dfas = buildDFAs propagate (Cfg.map (bbToDFA copies) cfg)
  in
      (id, Cfg.map replaceCopies dfas)
  end
```

Figure 2: Top-level function for the copy propagation optimization

The local value numbering optimization does not use dataflow analysis. It optimizes each basic block by numbering values and replacing redundant computations with moves. Local value numbering is the first optimization run, as it generates several (possibly redundant) move instructions. Copy propagation is then run to eliminate these moves, and dead code removal is run last to clean up any more unnecessary code.

Code Performance

The optimized code generated by mc was compared to unoptimized code also generated by mc, as well as to optimized and unoptimized code generated by clang. Code was compared based on run-time.

Optimization Notes

- The following benchmarks were modified to make them run longer:
  - BenchMarkishTopics
  - OptimizationBenchmark
  - hailstone
  - wasteOfCycles

  The run-times of these benchmarks with the benchmarks was measured to be 0 seconds by *time* for all cases. By making the code run longer the optimized and unoptimized code could be better compared.
- clang comparisons were performed against version "Apple LLVM 6.1.0" using -O3 for optimized code.
- All run-time results listed are an average of 10 trials and are in seconds. Only user time was measured. Tests were run on an Intel 2.7 GHz Core-i7 CPU.

Results

Table 1 summarizes the results.

Benchmark	**`mc` unopt**	**`mc` opt**	**`clang` unopt**	**`clang` opt**
BenchMarkishTopics	01.9030	01.9350	02.4870	00.5940
TicTac	00.0000	00.0000	00.0000	00.0000
binaryConverter	01.4950	01.3910	02.5910	00.0000
hailstone	00.0000	00.0000	00.0000	00.0000
mile1	00.0100	00.0100	00.0100	00.0000
programBreaker	00.0000	00.0000	00.0000	00.0000
wasteOfCycles	06.2440	06.5360	06.3320	06.3030
Fibonacci	02.2390	02.2370	01.7580	01.0620
OptimizationBenchmark	01.9790	01.5100	02.0570	00.0000
bert	00.0000	00.0000	00.0000	00.0000
creativeBenchMarkName	08.6800	08.0130	16.8790	00.0000
hanoi_benchmark	07.0090	06.8410	04.1040	01.9980
mixed	01.1380	00.7170	01.2540	00.0000
stats	00.0000	00.0000	00.0000	00.0000
GeneralFunctAndOptimize	01.3810	01.1260	01.2870	00.1770
biggest	00.0000	00.0000	00.0000	00.0000
fact_sum	00.0000	00.0000	00.0000	00.0000
killerBubbles	02.1250	02.2140	02.2910	00.8940
primes	00.1500	00.1590	00.2230	00.1000
uncreativeBenchmark	00.0000	00.0000	00.0000	00.0000

Table 1: Run-time results

Analysis

Table 1 shows that several benchmarks, despite having their inputs modified, still registered 0 user time. Meaningful comparison for these benchmarks is not possible.

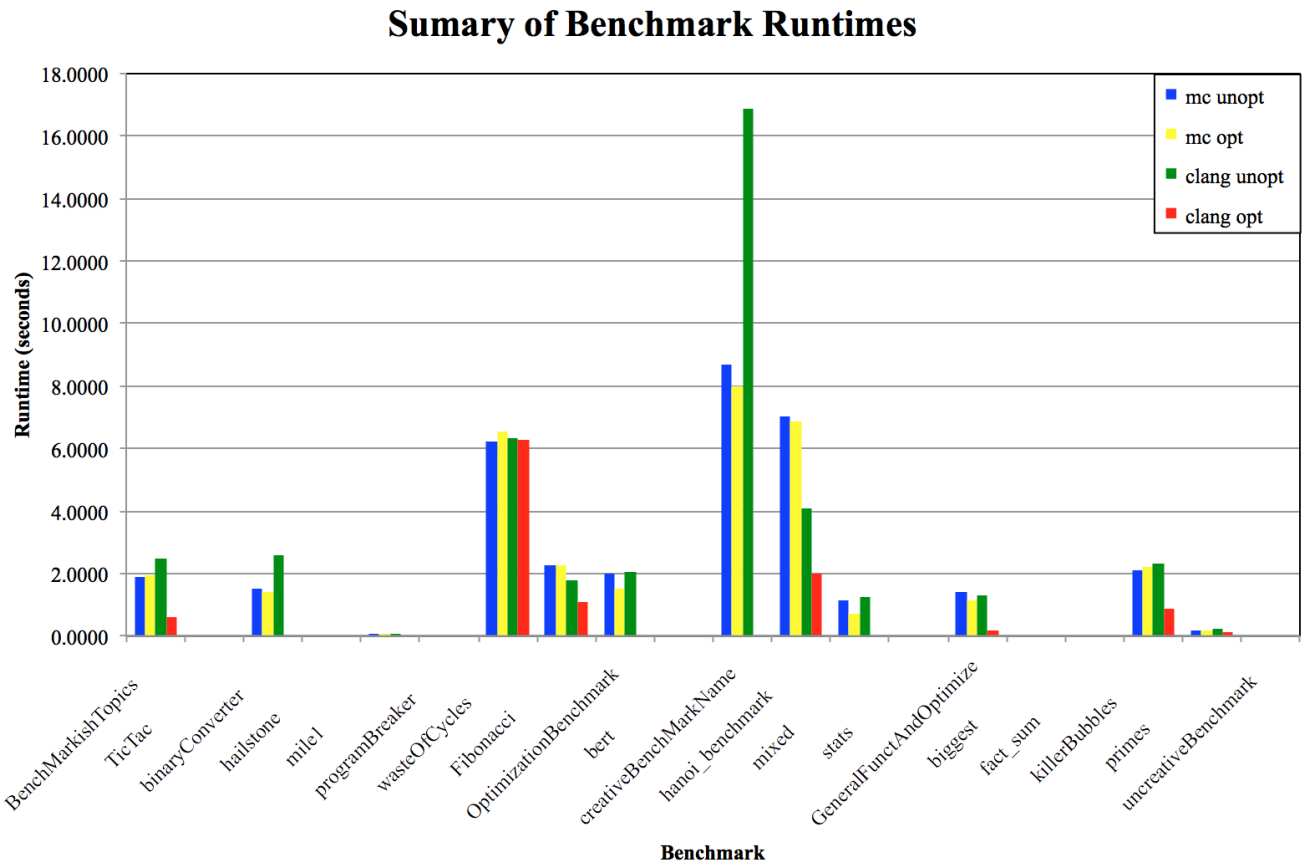
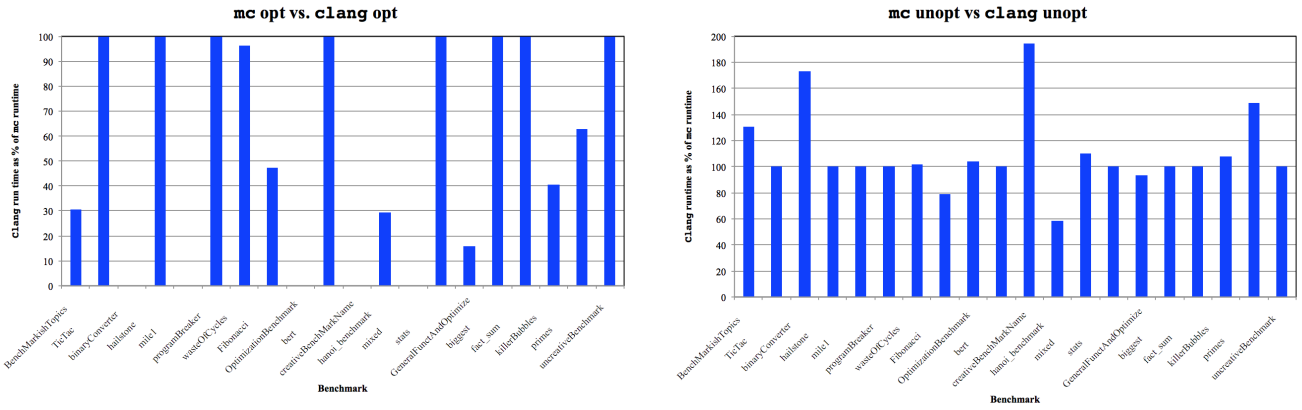


Figure 3: Summary of results



(a) With optimizations.

(b) Without optimizations

Figure 4: `Mc` vs `clang`