mc: A Compiler for the "Mini" Language

CPE431 Final Report / Jacob Hladky / June 3, 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:

```
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
    }
```

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.

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 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 an 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

Copy Propagation

Copy propagation is run in the `copyProp` module.

Dead Code Removal

Dead Code Removal is run in the `stripDeadCode` module.

Local Value Numbering

Local Value Numbering is run in the `lvNumbering` module.

Code Performance