A lean retargetable
C compiler

Chris Fraser, Bell Labs
Dave Hanson, Princeton
Optimize *our* time

◆ Minimize source code
◆ Compile fast
◆ Emit satisfactory code
◆ One literate program emits two outputs:
One source

The string table is an array of 1,024 hash buckets:

```
<<data>>=
static struct string {
    char *str;
    int len;
    struct string *link;
} *buckets[1024];
@ Each bucket heads a list of strings that share a hash value.
```
Sizes

◆ 12K lines target-independent
◆ Plus 1K lburg
◆ Plus ~700 lines per target:
  – tree grammar
  – code for proc entry/exit, data ...
◆ 400KB code segment includes 3 real targets + 2 for debugging.
Compile/execution times

- Compiles itself in half the time of gcc
- Emitted code generally within 20% of gcc’s
Code generation interface: Dags

- Shared data structures
- 36 base opcodes:
  - ADD INDIR JUMP ...
- 9 base types:
  - I D C ...
- but only 108 combos:
  - ADDI INDIRC ...
Interface functions

- begin/end module, function, block
- select/emit code
- define symbol
- emit initialized data
- change segment
typedef struct interface {
    unsigned little_endian:1;
    void *(defsymbol)(Symbol);
    ...
}

lcc -Wf-target=x86-linux foo.c
Code generation specs

◆ Tree grammars match IR and emit asm code
◆ Sample rules:
  \[
  \text{reg: ADDI}(\text{reg,con})
  \]
  \[
  \text{“addu } \%c,\%0,\%1\n” \quad 1
  \]
  \[
  \text{addr: ADDI}(\text{reg,con}) \text{ “} \%1(\%0)\text{”} \quad 0
  \]
◆ Specs: ~200 rules
◆ Hard-coded, bottom-up, optimal tree matchers, ~2000 lines
Twists

- Link-time CG: Fernandez
- Run-time CG: Poletto, Engler, Kaashoek
- Emit Java, even C: Fraser, Huelsbergen
- Debuggers: Hanson, Ramsey, Raghavachari
- Optimize battery life: Tiwari
More twists

- Compress code: Fraser, Proebsting
- Program directors: Sosic
- Browse code: Fraser, Pike
- Audit trees: Proebsting
Code compression
Proebsting and Fraser

◆ Accept a C program
◆ Emit:
  – a custom interpreter
  – postfix bytecodes
◆ Suits ROM, Java, optimizing linkers?
Organization

program to compress "i+1"

trees as ASCII "ADDI(., CNSTI[1])"

tree patterns "ADDI(*, CNSTI[*])"

trees as C initializer

driver

code generator

instruction-set generator

interpreter and interpretive code
Assigning opcodes

◆ Enumerate all trees:
  – ADDI(INDIRI(ADDRGP[i]),CNSTI[1])

◆ Patternize, up to some limit:
  – ADDI(*,CNSTI[*])
  – ADDI(*,CNSTI[1]) ...

◆ Generate a huge code generator
• Assign codes to all IR ops used by the program at hand
• With leftover codes, pick pattern that saves the most, then loop
Results

![Figure 2](image-url)
Run-time CG
Poletto, Engler, Kaashoek

◆ Construct code to sum n int args:
void cspec ConstructSum(int n) {
    int k, cspec c = `0;
    for (k = 0; k < n; k++) {
        int vspec v = (int vspec) param(k, TC_I);
        c = `(c + v);
    }
    return `{return @c};
}
Translate C to Java
Huelsbergen, Fraser

class FromLCC {
    public static int _main() {
        int pc = 0;
        M.sp -= 16;
        while(true) switch (pc) {
            ...
            i=0    case 3: M.putint((M.sp+4), 0);
            case 6: M.putint(((M.getint(rows[i]=1 (M.sp+4))<<2)+_rows), 1);
            case 7: M.putint((M.sp+4), i++ (M.getint((M.sp+4))+1));
            if (M.getint((M.sp+4)) < 8) {
                if(i<8)goto case 6 pc=6; continue; }
        }
    }
}
Program directors

Sosic

◆ Mix interpretive, compiled code

◆ Interpreter sends a (filtered) stream of events from the executor to the director
  – time, pc, result, ...

◆ Director watches and ...
  – animates calls,
  – watches for corrupt state, ...
Audit trees

Proebsting

◆ Some trees make no sense:
  – INDIRC(ADDF(*,*)

◆ One “back end” emits only Yes or No but matches with a grammar that specifies the valid trees.  *We* run it.
Big mistakes

- Need ASTs
- Need flow graphs
- “Economized” on long and void* metrics for too long
- Need interface pickle (now plural)
- Need better modularization:
  - Half the patches create a new error. See Dave’s coming book.
Smaller mistakes

- A graph-coloring register allocator
- Instruction scheduling
- Peephole optimization
What we like

◆ Simple and thus good infrastructure
◆ Fast
◆ Portable
◆ Complete
◆ Validated and kept that way
◆ We’d miss flexibility and fast compiles more than global opts