Show and Tell

1. Plan 9 Things (brief)
2. An Extensible Compiler for Systems Programming

Russ Cox
rsc@plan9

1127 Show and Tell
April 19, 2005

who am i

‘‘Neighborhood kid’’

1995
summer hacking Excel (jlb)

1995-1997
cable modems (nls, tom)

1997-1999
annoying Plan 9 user

1999
summer doing Plan 9 graphics (rob, jmk)

1999-present
assorted Plan 9 hacking

VBE

- use BIOS to set up VGA modes
- requires switching into real mode and back

Venti

- reworked significantly
- aggressive caching, prefetching, batching, delayed writes
- Bloom filter to avoid index misses

Plan 9 from User Space (plan9port)

- port bulk of Plan 9 software to Unix systems
- Linux, FreeBSD, NetBSD, SunOS, Mac OS X

An Extensible Compiler for Systems Programming

Russ Cox
Frans Kaashoek
Eddie Kohler

l@pdos.csail.mit.edu

Outline .....

Why bother?

What could it do?

How could it work?

Ground rules:

- interrupt with questions
- work in progress
- show and tell, not a job talk

Man vs. Machine – Linguistic Tensions .....

The most important readers of a program are people.

- ‘‘We observe simply that a program usually has to be read several times in the process of getting it debugged. The harder it is for *people* to grasp the intent of any given section, the longer it will be before the program becomes operational.’’
  — Kernighan and Plauger, 1974.

- ‘‘Programs are meant to be read by humans, and only incidentally for computers to execute.’’
  — Donald Knuth

- ‘‘Write programs for people first, computers second.’’
  — Steve McConnell

Why (not) use C? .....

Low-level execution model close to hardware

- gives programmer lots of power, control
- with great power comes great responsibility
- who wants all that responsibility?

Why (not) use ___ ? .....

(for ___ in Perl, Python, C++, ML, etc.)

High-level execution model lets you ignore the hardware

- makes it easier to think at a high level of abstraction
- cannot think at other levels, both higher and lower

Really want a language that lets you work at the level of abstraction you want

- instead of the level the language designer chose

The Extensible Compiler Approach.....

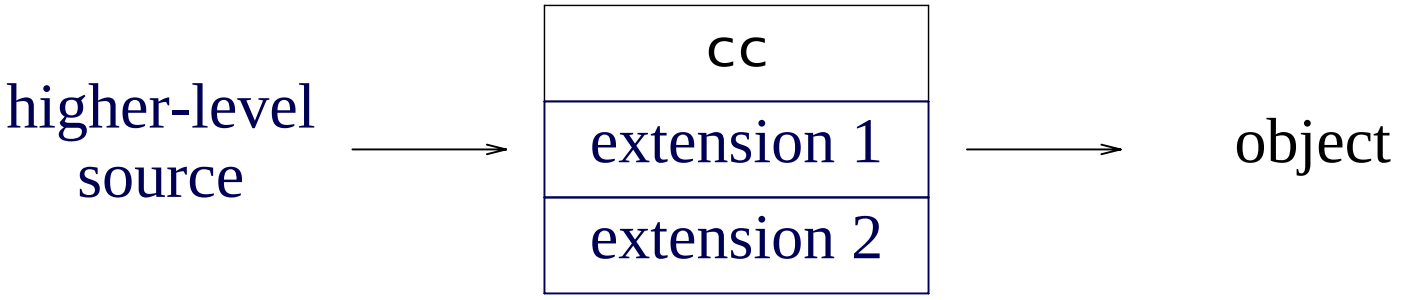
C compiler as base.

Extension modules loaded into compiler dynamically.

- rewrite high-level code into lower-level constructs
- back end is standard C compiler
- users can supply extensions themselves

Object files remain the *lingua franca* of the system.

The Extensible Compiler Approach.....



C compiler as base.

Extension modules loaded into compiler dynamically.

- rewrite high-level code into lower-level constructs
- back end is standard C compiler
- users can supply extensions themselves

Object files remain the *lingua franca* of the system.

Pointer qualifiers

```
int copyin(void*, void user*, int);
int copyout(void user*, void*, int);
int foo(void *);
int bar(void user*);
```

- cannot implicitly or explicitly convert `void*` to `void user*`
- need to add syntax, type representation, and type conversion rules

Extensions .....

Anonymous structure elements (a la Ken)

```
struct Foo
{
    Lock;
    int x;
};
void lock(Lock*);

Foo *foo;

lock(foo);
```

- need to add compilation rules

Extensions .....

Good thread creation syntax

```
void threadcreate(void (*fn)(void*), void *arg);
spawn { print("hello\n"); print("goodbye\n"); }
spawn f(x,y,z);
```

- need to add syntax, compilation rules
- compiler must determine which arguments to copy into argument for new thread

Checking `printf` format strings

```
printf("hello, %s\n", 12345);
```

- need to handle all calls to functions named `printf`

Structure of Extensible Compiler .....

Extensible syntax

- add new tokens or reserved words
- add new grammar rules
- all while compiler is running

Extensible compiler data types

- simple classes for abstract syntax, types, etc.

Extensible functions

- define how the new data types get handled
- like Lisp’s generic functions

Extensible lexer

- assume `cpp` rules for tokenization
- augment with table of tokens + ++ += etc.
- augment with table of words `for` `break` `default` etc.
- can edit these tables on the fly

```
yylexnewsym($G, "user");
yylexdelsym($G, "user");

cppaddtok(cpp, "<-");
```

Extensible Syntax .....

Extensible lexer

Extensible parser

- new `libyacc` builds LALR(1) parsing tables on the fly
- incremental compilation of a large NDFA
- full compilation isn't much more expensive

```
void add(void *vout, void *vin) {
    double *in=vin, *out=vout;
    *out = in[0] + in[2];
}

/* N: N + N */
addrule(g, add, "N", "N", "+", "N", nil);
```

Extensible Syntax .....

Compiler extension translates *yacc*-like syntax into calls to lex, yacc library.

```
yacc(g){
    yaccsym token NUMBER "\n";
    ...

    yaccsym left "+" "-";
    yaccsym left "*" "/" "%";
    yaccsym left "UNARYMINUS";
    ...

    expr:
        NUMBER
    |   expr "+" expr   { $$ = $1 + $3; }
    |   expr "*" expr   { $$ = $1 * $3; }
    |   "-" expr   %prec UNARYMINUS   { $$=-$2; }
    |   "(" expr ")"    { $$ = $2; }
    ;
}
```

Extensible Functions .....

Want to change existing functions, add new ones

Doesn’t necessarily fit C++ or other models

- change behavior of existing functions
- define new methods for existing data types
- case analysis and data types not necessarily aligned
- always fall back to default

Extensible functions - implementation.....

Implement extensible functions as chains of handlers.

```
List *handlers;
void
compile(Node *n)
{
     int (*fn)(Node*);
     List *l;

     for(l=handlers; l; l=l->tl){
          fn = l->hd;
          if(fn(n) == Handled)
               return;
     }
     /* default behavior here */
}

int
compileprintcheck(Node *n)
{
     if(isprintcall(n)){
          /* check print arguments, emitting warnings */
     }
     return NotHandled;
}
handlers = mklist(handlers, compileprintcheck);
```

Extensible functions - compiler help.....

Easier with explicit language support.

```
extensible
void
compile(Node *n)
{
     /* default behavior here */
}

extend
void
compile(Node *n)
{
     if(isprintcall(n)){
          /* check print arguments, emitting warnings */
     }
     default;
}
```

Extensible data types.....

Could do by hand.

```
struct Node
{
     ...
     int typetag;
};

struct YaccNode
{
     Node;    /* using typetag==TypeYaccNode */
     int yaccinfo;
};

struct OtherNode
{
     Node; /* using typetag==TypeOtherNode */
     char otherinfo;
};

if(node->typetag == TypeYaccNode){ ... }
```

Extensible data types.....

Better with help from the language.

```
extensible struct Node
{
     ...
};

struct YaccNode extend Node
{
     int yaccinfo;
};

struct OtherNode extend Node
{
      char otherinfo;
};

if(istype(node, YaccNode)) ...
```

Pointer qualifiers

- Add new syntax

```
void
xinit(Ygram *g)
{
    yacc(g){
        yaccsym <vval> qname;
        yaccsym term "user";
        qname: "user" { $$ = BUSER; };
    }
}
```

Pointer qualifiers

- Add new syntax
- Add new type checking rules

```
extend
int
canimplcast(Node *n, Type *t2)
{
    if(isuserptr(n->type) && !isuserptr(t2)){
        werrstr("cannot discard user qualifier");
        return 0;
    }
    default;
}
```

Anonymous structure elements

- Add new syntax

```
void
xinit(Ygram *g)
{
    yacc(g){
        yaccsym <type> type;
        yaccsym term sudecl ";";

        sudecl: type ";"
            {
                declare($1, nil, 0);
            }
        ;
    }
}
```

Anonymous structure elements

- Add new syntax
- Add new handler in type phase

```
extend
Type*
lookstruct(Type *t, char *name, int *offset)
{
    Type *tt;

    if((tt = oldlookstruct(t, name, offset)) != nil)
        return tt;
    /* for each anonymous element in struct/union */ {
        if((tt = lookstruct(anon, name, offset)) != nil){
            *offset += /* anon offset in t */;
            return tt;
        }
    }
    return nil;
}
```

Good thread creation syntax

- Add new syntax

```
void
xinit(Ygram *g)
{
    yacc(g){
        yaccsym <node> stmt expr;

        stmt: "spawn" expr ";"
            {
                $$ = new(OSPAWN, $2, Z);
            }
        ;
    }
}
```

Good thread creation syntax

- Add new syntax
- Add handler

```
extend
void
compile(Node *n)
{
    if(n->op != OSPAWN)
        default;
    /* lift n->left into its own function */
    /* emit code to construct arguments */
    /* emit threadcreate(newfn, arguments); */
    return;
}
```

Checking `printf` format strings

- Add handler

```
extend
void
compile(Node *n)
{
    if(isprintcall(n)){
        /* check print arguments, emitting warnings */
    }
    default;
}
```

Details I Skipped.....

Constants and new types

- `BUSER`, `OSPAWN` etc. must get defined in a meaningful way.

Code transformations

- code lifting and a library of other useful transformations.

Code generation

- need good syntax to generate programs
- Lisp wins hands down

Status.....

Implemented as translator from extended C to normal C

- using `gcc` to compile to machine code
- eventually do entire compilation
- can compile itself, relies heavily on `yacc` extension
- necessary `gcc`isms implemented as extensions

Extensibility being fleshed out

- extensible syntax implemented, works well
- still working out reparsing
- adding extensible data types, functions now

Related work.....

Lisp, Scheme, ‘‘Macros for C’’, etc.

- somewhat solid syntax story
- not much story for changing other aspects of compilation

Future .....

Get compiler up and running

Get new users

- use to compile Plan 9 C on Unix
- Aegis processor group (keep memory spaces separate)
- Asbestos operating system (make handles more palatable)

Closing.....

Not going to save the world.

- ‘‘Whatever language you write in, your task as a programmer is to do the best you can with the tools at hand. A good programmer can overcome a poor language or a clumsy operating system, but even a great programming environment will not rescue a bad programmer.’’

  — Kernighan and Pike, *The Practice of Programming*