`rsc`: an extensible compiler

Russ Cox
Frans Kaashoek
Eddie Kohler

PDOS Group Meeting
January 24, 2006

`http://swtch.com/~rsc/talks/`

Problem (big picture)

Programs are buggy.

Because programming languages don’t work for you.

You work for them.

How do you resolve mismatch between programming task and language?

Outline

Problem 5 slides

Specific Extensions 10 slides

How to Implement `rscc` 10 slides
- parser
- internal representation

Please interrupt with questions.

Good Languages?

Some languages make some bugs easier to avoid / find.

- good abstractions
- type systems

Never good enough.
People extend languages any way they can.

- C preprocessor (easy, but rarely good)
- custom preprocessors (good, but rarely easy)
- custom checkers

Useful Preprocessors

Click (Kohler)
language to describe module properties, router configs

Tame (Krohn)
more convenient syntax for libasync
(Even *I* can write tame programs!)

Ace (Gosling)
syntax-based preprocessor, used for bitblt

QT Rewriter
- used for simpler event syntax in QT library

Useful Checkers

Magik (Engler)

checking of flow-sensitive conditions, using xgcc
more recent work about guessing things to check

Uno (Holzmann)

another static checker, actually available.

Sparse (Torvalds)

add annotations to Linux source for lock/unlock pairing, kernel/user address spaces

Implementation Techniques

Start from scratch

lots and lots of work, requires knowing about compilers

Edit an existing compiler

lots of work, requires knowing about compilers, requires *knowing* that particular compiler

Problem (recap)

Programs are buggy.

Because languages aren't good for programming.

Lots of good can come from tailoring the language.

But it's far too much work for most programmers.

How can we lower the bar?

Solution: Extensible Compiler

Extensible compiler

- let programmers add extensions (`.so`files)
  to the compiler during compilation.

- extensions are separate from programs that use them
  (think media-player codecs)

- extensions add new language features, new checking

- make it easy to write these extensions

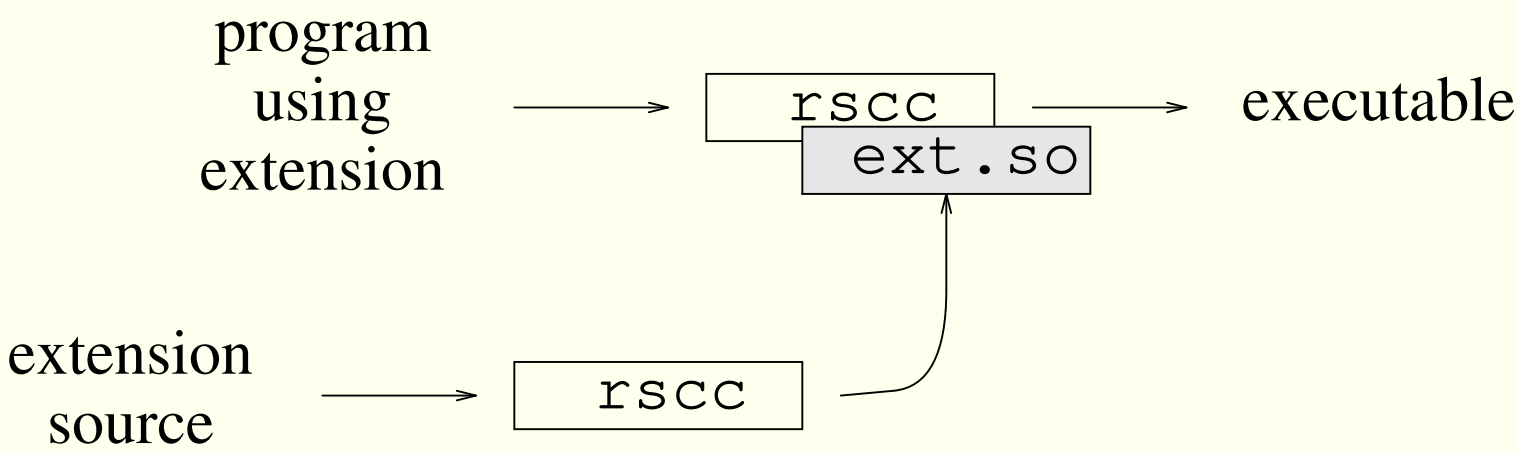
- hide compiler when possible

- make compiler easy to understand

Solution: Extensible compiler

Extensible compiler, without extensions:

Extensible compiler, with one extension:



Expected Uses

Extensions are usually small (and can start small).

Programmers write compiler extensions
while writing the programs that use them.

Project leaders write compiler extensions
to make big project easier for others.

Net work should be less than not using extensions.

Make possible the language changes mentioned earlier.

Example Extension: Alef

```
void
primetask(chan(int) c)
{
    chan(int) nc;
    int i, p;

    p = <-c;
    print("%d\n", p);
    alloc nc;
    task primetask(nc);
    for(;;){
        i = <-c;
        if(i%p)
            nc <-= i;
    }
}
```

Alef original

```
void
primethread(void *arg)
{
    Channel *c, *nc;
    int i, p;

    c = arg;
    p = recvul(c);
    print("%d\n", p);
    nc = chancreate(sizeof(ulo
    threadcreate(primethread,
    for(;;){
        i = recvul(c);
        if(i%p)
            sendul(nc, i);
    }
}
```

C transliteration

Example Extension: Alef

Alef	C + libthread
`chan(int) nc;`	`Channel *nc;`
`alloc nc;`	`nc = chancreate(sizeof(ulon`
`task primetask(nc);`	`threadcreate(primethread, n`
`nc <-= i;`	`sendul(nc, i);`

Benefits of Alef:

- more concise, more readable.
- type-checked channels.
- good thread creation syntax.
- good communication syntax (`alt`)

Alef is dead.

- compiler author moved on.
- those left behind switched to C.
- gave up the benefits for ease of maintenance

Example Extension: Tame

```
TAME(static void
  try_connect(str h,
    int port))
{
  VARS { int fd; }
  BLOCK{ tcpconnect(h,
       port, @(fd)); }

  if(fd < 0)
    warn << h << "\n";
  exit(0);
}
```

Tame TCP client.

```
static void cb1(str, int);
static void try_connect(str
    int port)
{
  tcpconnect(h, port,
      wrap(cb1, h));
}
static void cb1(str h, int
{
  if(fd < 0)
    warn << h << "\n";
  exit(0);
}
```

Equivalent libasync

Example Extension: Tame

```
TAME(static void
  try_connect(str h,
    int port))
{
  VARS { int fd; }
  BLOCK{ tcpconnect(h,
       port, @(fd));

  if(fd < 0)
    warn << h << "\n";
  exit(0);
}
```

Tame TCP client.

```
static void cb1(str h, int
{
   if(fd < 0)
      warn << h << "\n";
   exit(0);
}
static void try_connect(str
    int port)
{
   tcpconnect(h, port,
      wrap(cb1, h));
}
```

Libasync in the wild.

Example Extension: Tame

```
VARS { int fd; }      static void cb1(void) {exit
BLOCK{ tcpconnect(h, tcpconnect(port, wrap(cb1))
    port, @(fd)); }
exit(fd);
```

Benefits of Tame:

- more concise, more readable
- does not encourage upside-down programming
- easy to handle multiple parallel threads-of-control
- can write structured programs again (`if`, `for`, ...)

Example Extension: Plan 9 Kernel Checker

Plan 9 kernel has many conventions to remember:
(So does any large program.)

- lock/unlock pairing
- no sleeping in interrupt handlers
- interrupt handlers must use ilocks
- exception stack for error handling
- kernel/user pointers
- no dereferencing user pointers in interrupt handlers
- ...

Many are checked at run-time.
No fundamental reason not to check at compile time.

Example Extension: Plan 9 Kernel Checker

Add attributes to document and enforce conventions.

```
void *ialloc(int) attribute(nosleep);
void *malloc(int) attribute(nosleep);
void print(char*, ...);

void*
ialloc(int n)
{
    void *v;

    v = malloc(n);


    return v;
}
```

Example Extension: Plan 9 Kernel Checker

Add attributes to document and enforce conventions.

```
void *ialloc(int) attribute(nosleep);
void *malloc(int) attribute(nosleep);
void print(char*, ...);

void*
ialloc(int n)
{
    void *v;

    v = malloc(n);
    if(v == nil)
        print("ialloc returning nil\n");
    return v;
}
```

Compiler can tell programmer he made a mistake.

Front end only
- compile to low-level C and invoke `gcc`

Written using itself

Extensible LR(1) parser

Extensible grammar notation

Destructuring syntax

Restructuring syntax

Other extensions as convenient

Extensible LR(1) parser

Shift-reduce parsers run an NFA on input.

NFA says shift or reduce at each step

Parse `x * x + x`

```
x           * x + x   shift
x *         x + x     shift
x * x       + x       reduce x => x*x
x           + x       shift
x +         x         shift
x + x                 reduce x => x+x
x
```

Extensible LR(1) parser

Shift-reduce parsers run an NFA on input.
- NFA says shift or reduce at each step
- NFA corresponds exactly to grammar rules

`Yacc` precomputes, hard-codes equivalent DFA.

New parser for rscc:

- run determinization on the fly, caching result.
- if grammar changes, throw away DFA built so far.

- normal C library interface

```
yynewsym(g, "\"**\"", 0);
yylexnewsym(g, "\"**\"");
yyrulestr(g, yyaction, "expr", "expr", "\"**\""
```

Grammar Syntax

Use an extension to present a better parser interface.

```
yacc {
    yaccsym right "**" %prec "*";

    expr: expr "**" expr;
}
```

Compiles into appropriate function calls.

Provides default `yyaction` rule that builds an abstract syntax tree node.

Destructuring Syntax

Need to make abstract syntax tree nodes accessible.

Don’t want extension writer to need to know internal represen

Provide matching operator ~ for destructuring syntax.

```
Node *a, *b, n;

if(n ~ `expr{ \a**\b })
    warn("using **: base=%N exponent=%N", a, b);
```

Restructuring Syntax

Same syntax as destructuring, but used as expression instead of next to ~.

```
Node *n, *a, *b;

if(n ~ `expr{ \a**\b })
    n = `expr{ pow(\a, \b) };
```

The `a` and `b` are substituted as units, not as token streams.

```
int x;
Node *n, *nn;

x = 2;
n = `expr{ 1+\x };
nn = `expr{ \n*3 };
```

Extensible Functions

Linked list of function pointers implementing a function.

Each function can handle a call itself, with or without invoking next function in the list.

Syntax codifies the idiom.

```
extend
Node*
compile(Node *n)
{
    Node *a, *b;

    if(n ~ `expr{ \a**\b })
        return compile(`expr{ pow(\a, \b) });
    return default;
}
```

Extensible data structures

Same idea as functions; not sure of implementation.

```
extend
struct Node
{
    int extra;
    int other;
    int field;
};
```

Other extensions: iterators

```
static int start(State *z, Rule *r);
static int next(State *z, Sym **s);
static int end(State*);
iterator(start, next, end);

Rule *r;
Sym *s;

for(s in r)
    something(s);
```

Related Work

Lots of macro systems
- can't write checkers with those.
- often run into scoping problems

Other extensible languages/compilers
- require programmers to know too much about internals
- not easy enough to write extensions

Status

`Rscc` exists.

- uses extensible grammar, yacc syntax
- provides its own grammar action functions
- uses restructuring, not destructuring
- uses extensible functions heavily
- no extensible data structures yet
- uses iterators heavily

- know how to switch over to implicit actions and destructuring
- need to find ways to hide other aspects of compiler internals (types, flow control, ...)

- submit to OSDI

- always looking for more target uses