Making C Safely Extensible

Robert Grimm
New York University
The Problem

- Complexity of modern systems is staggering
  - Increasingly, a seamless, global computing environment
- System builders continue to use C (or C++)
  - Inadequate basis for coping with this complexity
    - There simply is too much code
  - As a result, software quality and security suffer
    - Think critical updates for Windows or Mac OS X
- *Metaprogramming* holds considerable promise
  - Basic idea: Automatically compute (parts of) programs instead of manually writing them
The Power of Metaprogramming

- libasync [Mazieres, Usenix ’01]
  - Provides extensions for asynchronous programming
    - Helps write straight-line code in presence of callbacks
- Capriccio [von Behren et al., SOSP ’03]
  - Provides a highly scalable threading package
    - Uses stack depth analysis to reduce memory overhead
- MACEDON [Rodriguez et al., NSDI ’04]
  - Provides DSL for specifying overlay protocols
    - Compiles state machine specification to runnable system

→ How to give this power to other system builders?
  - Make language and compiler extensible through macros!
**The Four Requirements**

- *Expressive* enough for new definitional constructs
  - Domain-specific (e.g., closures, state machines)
  - General (e.g., modules, objects, generics, …)

- *Safe* to statically detect program errors
  - Not only macro hygiene, but also new typing constraints
    - Avoid string of obscure error messages (e.g., C++ templates)

```c
char* host = "www.example.com";
char* path = "/index.html"

dnslookup(host, void closure(ip_t addr) {
    tcpconnect(addr, 80, void closure(int fd) {
        ... write(fd, path, strlen(path)); ...
    });
});
```
The Four Requirements (cont.)

- **Fine-grained** to compose and reuse extensions
  - Track dependencies
  - Detect conflicts
- **Efficient** code as compiler output
  - Specialized expansions (e.g., `foreach` for arrays)
  - Domain-specific optimizations
    - E.g., stack depth analysis for Capriccio
- No existing macro system meets all requirements
  - Little work on extensible type systems, macro composability
Enter xtc (eXTensible C)

- Macro system for C
  - Grammar modification rules
  - Abstract syntax tree (AST) transformation rules
  - Typing rules
  - Macro (meta-)module system

- Toolkit for extensible source-to-source transformers
  - *Rats!*, our extensible parser generator
  - AST representation and transformation framework

- Traditional compiler (gcc)
  - Conventional optimizations and binary code generation
Why C?

- Still popular for operating and distributed systems
  - E.g., Linux, BSDs, Apache, MySQL, BerkeleyDB
- Unlike Java or C#, no need for a virtual machine
  - Which, ironically, is often implemented in C
- Unlike C++, Objective-C, Java, C#, no object model
  - C++’s object system is very complex
  - Java’s model is much simpler, but has performance issues
  - More powerful models can be quite useful
    - Think predicate dispatch [Millstein, OOPSLA ’04]
Talk Outline

- Motivation and requirements
- Our parser generator and prototype framework
- The xtc macro system
  - Including interesting/open research issues
- Conclusions
Our Parser Generator and Prototype Framework

- Focused on extensibility and ease-of-use
  - Easy to change grammars, transformers
    - Changes should be localized
- Written in Java
  - Represents temporary engineering trade-off
    - Simple object system, GC, collections framework, reflection
  - Provides us with a first, large-scale test case
    - As soon as xtc is functional, will make it self-hosting
- Released as open-source
  - Currently at version 1.3, version 1.4 coming “soon”
Basic strategy: Packrat parsing

- Originally described by Birman [Phd ‘70]
- Rediscovered by Ford [ICFP’02, POPL ’04]
  - Pappy: A packrat parser generator for Haskell
- Parses top-down (like LL)
- Orders choices (unlike CFGs, LR, and LL)
- Treats every character as a token (unlike LR and LL)
- Supports unlimited lookahead through syntactic predicates
- Performs backtracking, but memoizes each result
  - Linear time performance
  - One large table: characters on x-axis, nonterminals on y-axis
Rats! vs. Pappy

- More concise grammar specifications
  - Automatic deduction of semantic values
  - Automatic construction of AST (using “generic nodes”)
- Better support for debugging and error reporting
  - Pretty-printing of grammars and parsers
  - Automatic annotation of AST with source locations
  - Automatic generation of error messages
- More aggressive optimizations
  - Faster (~5 times), less memory (~4 times) than Pappy
  - Still slower than ANTLR and JavaCC (2-6 times)
  - But reasonable absolute performance (67 KB in 0.3 sec)
Our Framework So Far

- Three main abstractions
  - Abstract syntax tree (AST) nodes to represent code
  - Visitors to walk and transform AST nodes
    - Methods selected through reflection (with caching)
  - Utilities to store cross-visitor state
    - Analyzer for analyzing grammars
    - Printer for pretty printing source code

- Two axes of extensibility
  - New visitors to represent new compiler phases
  - New AST nodes to represent new language constructs
    - `process()` methods specified with new AST nodes
The xtc Macro System

- xtc implemented as pipeline ending with gcc
  - Ensures wide-spread availability
- xtc macros consist of four kinds of rules
  - Grammar rules for expressiveness
  - AST transformation rules for expressiveness, efficiency
  - Typing rules for safety, expressiveness
  - Dependency rules for composability
Macro Rules in Detail

- Grammar rules specify syntax
  - Add or remove alternatives in choices
    - May specify “before” and “after” constraints on named alternatives
  - Add new productions
- AST transformation rules reduce language, optimize
  - Expressed through templates
    - Literal syntax mixed with pattern expressions
  - Selected based on constraints
    - Nonterminals, kinds of types, individual types
    - Builds on Maya’s multiple dispatch [Baker & Hsieh, PLDI ‘02]
- Supplemented by AST analysis and transformation library
Macro Rules in Detail (cont.)

- Typing rules express safety constraints
  - Introduce new types
  - Express type compatibility (e.g., subtyping)
  - Deduce type of compound expressions, statements
  - Supported by symbol table and type records
    - Represented as collections of (nested) name/value pairs

- Dependency rules express inter-macro constraints
  - Specify other macros used by macro
    - E.g., closure macro building on object system
  - Group several macros into a larger extension
    - E.g., objects, GC, exceptions, threading/synchronization in one
The Beginnings of a Class Macro

source {
    class $(id:Identifier) { $(members:MemberDeclarations) } 
} target {
    typedef struct $(gensym(id, "object")) { 
        void *vtable;
        $(select("/FieldDeclaration", members))
    } * $id;
    ...
} newtype {
    kind = "Object",
    class = id,
    fields = createmap(
        select("/FieldDeclaration/Identifier", members),
        typeof(select("/FieldDeclaration/Type", members)),
    methods = ...
}
The Beginnings of a Class Macro (cont.)

- Starting to define supporting library functions
  - AST querying and (re-)construction (e.g., `select()`)
    - Not shown: Splicing implicit pointer to `self` into methods
  - Typing (e.g., `createmap()`, `typeof()`)
    - Not shown: Explicit symbol table access
  - Program analysis (e.g., `freevars()`)
    - Not shown: Making implicit references to `self` explicit

```plaintext
source {
  $(exp:Expression!Object) . $(id:Identifier)
} target {
  $exp -> $id
} type {
  select("/fields/$id", typeof(exp))
}
```
More Research Issues

- **Macro expansion order: Outside-in vs. inside-out**
  - Outside-in supports local macro definitions
    - But requires parsing and macro expansion be interleaved
  - Inside-out required by some macros (e.g., dynamic codegen)
  - Either way, parsing and typing have to be interleaved
    - Need to distinguish between type names and regular identifiers in C

- **Choice of base language**
  - C has many redundant constructs (3 types of loops, ++, --)
    - Tedious to specify domain-specific optimizations
  - CIL provides better basis [Necula et al., CC ‘02]
    - Concise and formally defined language kernel
Even More Research Issues

- Expressiveness of macro module system
  - Direct dependencies are good enough to get us started
  - But we may want to provide more flexibility
    - I.e., there may be more than one way to implement a syntax
      - Think: Object layout, multiple/predicate dispatch
  - We are considering a macro traits facility
    - Macros implement protocols, comparable to interfaces
- What do you need for realizing your domain-specific optimizations in xtce?
Conclusions

- Metaprogramming can help us cope with increasing complexity of (distributed) systems
- A “good” solution needs to meet four requirements
  - Expressiveness, safety, composability, efficiency
- xtc strives to meet these requirements for C
  - Expresses macros as collections of rules
    - Grammar, AST transformation, typing, dependency
  - Builds on our toolkit for source-to-source transformers and on Rats!, our packrat parser generator
  - Leaves conventional optimizations and code generation to gcc