Reliable and Automatic Composition of Language Extensions to C

Making language extension practical with AbleC

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk
(University of Minnesota)

October 2017

What do we mean by language extensions?

```
typedef datatype Tree Tree;
datatype Tree {
  Fork ( Tree*, Tree*, const char* );
  Leaf ( const char* );
};
```

```
cilk int count_matches (Tree *t) {
  match ( t ) {
    Fork(t1,t2,str): {
      int res_t, res_t1, res_t2;
      spawn res_t1 = count_matches( t1 );
      spawn res_t2 = count_matches( t2 );
      res_t = (str =~ /foo[0-9]+/) ? 1 : 0;
      sync;
      cilk return res_t1 + res_t2 + res_t ;
    };
    Leaf(str): { return (str =~ /foo[0-9]+/) ? 1 : 0; };
  }
}
```

Language extension research program goals

Expression problem[1] criteria, as applied to AST:

1. Introduce both new syntax and new analysis
2. Static checking of this representation
3. Without modifying the original code
4. Separate compilation

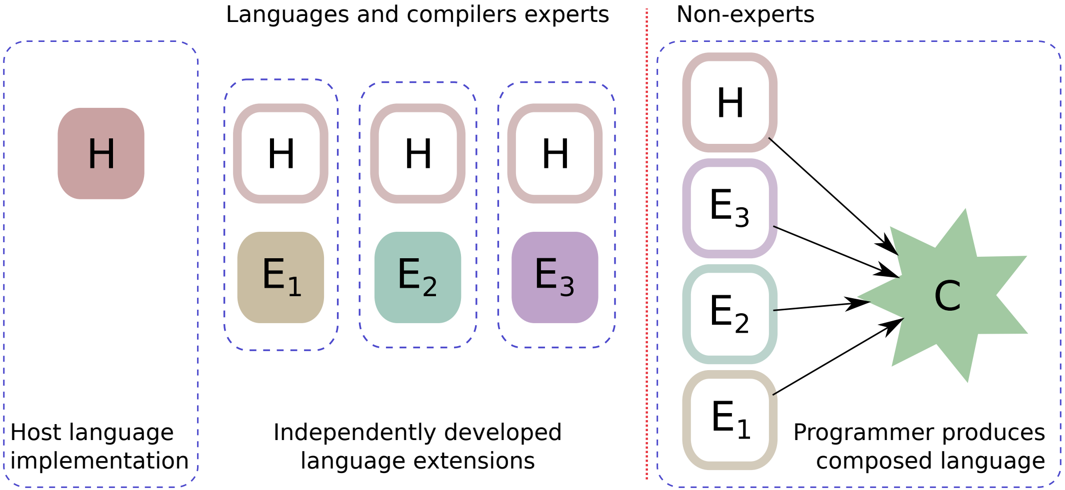
Independent extensibility[2]:

5. “Unordered,” *composable* extensions

Our criteria:

6. Automatic composition, no glue code

Library model of language extension



Distinguishing characteristics

Many approaches to extensible languages.

- ExtendJ (using JastAdd)
- SujarJ (using Spoofax)
- Wyvern/VerseML
- mbeddr
- XTC
- XoC
- object algebras
- and others

What distinguishes this work is how two questions are answered:

1. Who composes the language features?
2. How expressive are the supported features?

Success! Composition without conflict.

- Copper & Silver: tools for creating extensible compilers
- AbleC: our application to C

The general operation of AbleC

`edu:umn:cs:melt:IndData`
`org:bar:cilk`
`com:foo:RegEx`
$\Downarrow$
`Silver` $\Longrightarrow \Longrightarrow$ `my-ableC`
$\Uparrow$
`edu:umn:cs:melt:ableC`

`pts.xc`
$\Downarrow$
`cpp`
$\Downarrow$
`pts.i`
$\Downarrow$
`my-ableC`
$\Downarrow$
`pts.c`
$\Downarrow$
`gcc`
$\Downarrow$
`a.out`

$$\left\{\begin{array}{l} -scanning \\ -parsing \\ -AST\ construction \\ -type\ checking \\ -optimization \\ -C\ code\ generation \end{array}\right.$$

Previous work

Allowing expressive extensions:

Context-aware scanning: safe syntactic overlap between extensions
Forwarding: tool for solving the expression problem

Modular analyses ensure reliable composition:

Modular determinism analysis: no unexpected syntactic conflicts
Modular well-definedness analysis: well-defined attribute grammar

Concurrent work

Coherent non-interference: extensions behave as specified

This talk

- Modular analyses impose restrictions
- Previous work, let’s skip over that

- What kinds of extensions can we build?
- What kinds of extensions can’t we build for plain C?
- What host language modifications allow for more kinds of extensions?

Extensions to AbleC

```
typedef datatype Tree Tree;
datatype Tree {
  Fork ( Tree*, Tree*, const char* );
  Leaf ( const char* );
};
```

```
cilk int count_matches (Tree *t) {
  match ( t ) {
    Fork(t1,t2,str): {
      int res_t, res_t1, res_t2;
      spawn res_t1 = count_matches( t1 );
      spawn res_t2 = count_matches( t2 );
      res_t = (str =~ /foo[0-9]+/) ? 1 : 0;
      sync;
      cilk return res_t1 + res_t2 + res_t ;
    };
    Leaf(str): { return (str =~ /foo[0-9]+/) ? 1 : 0; };
  }
}
```

Extensions to AbleC

```
transform {
  for (unsigned i : m, unsigned j : n) {
    c[i][j] = 0;
    for (unsigned k : p) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
} by {
  split i into (unsigned i_outer,
                unsigned i_inner : (m - 1) / NUM_THREADS + 1);
  parallelize i_outer into (NUM_THREADS) threads;
  tile i_inner, j into (TILE_DIM, TILE_DIM);
  split k into (unsigned k_outer,
                unsigned k_unroll : UNROLL_SIZE,
                unsigned k_vector : VECTOR_SIZE);
  unroll k_unroll;
  vectorize k_vector;
}
```

Extensions to AbleC

```
matlab
(unsigned char pic[height][width][3]) =
  mandelbrot(double xstart, double xend,
             double ystart, double yend, double iter_d)
{
  ...
}
```

Other extensions

- **Sqlite**: describing schemas, writing queries, LINQ-like
- **Tensor/matrix/vector**: more scientific computing applications
- **Go concurrency**: other parallel computing models

Enabling more extensions

GCC extensions

- Something we already have to support
- One in particular especially useful: statement-expressions
- General problem with stratified grammars
- Transformations are local

```
1 + ({ foo x; f(&x); x.val; })
```

Operator overloading

- Cannot be implemented as an extension
  - (Changes meaning of host productions)
- Quite useful in enabling extensions, though
- An alternative non-syntactic “hook” from host into extension

```
matrix(A * x + V)
```

vs

```
A * x + V
```

Lifting declarations

- Extension translation (*“forwarding”*) is local
- Sometimes need non-local transformations

```
lambda (int x) -> (*z = x * y + *z)
```

- Lift out a function declaration
- Lift out a closure type declaration

Type qualifiers

- General problem with annotation-driven analysis
  - Need to have an equivalent host-language tree
- Previous work[3] has extended C with generic type qualifiers
  - Changes meaning of host again, not an extension in our sense
- New qualifiers as language extensions (See our GPCE paper this year!)

```
typedef  datatype Expr  Expr;
datatype Expr {
  Add (Expr * nonnull, Expr * nonnull);
  Mul (Expr * nonnull, Expr * nonnull);
  Const (int);
};
```

Enabling more extensions

Restrictions on extensions are host-language-relative

GCC extensions escaping the confines of expressions
Operator overloading additional options for “hooking into” extensions
Lifting declarations escaping the confines of local scope
Type qualifiers some kinds of annotation-driven analysis

Host language type system is important

Summary

- Extended notion of expression problem
- Reliable & automatic composition of language extensions
- Breadth of extensions possible
- Language extension impacts language design
    - Clear line between extension and modification
    - We can get experience with extensions before standardization

Thanks!

Get in touch:

- Eric Van Wyk `<evw@cs.umn.edu>`
- Ted Kaminski `<tedinski@cs.umn.edu>`

Check things out:

- melt.cs.umn.edu
- github.com/melt-umn