Putting C/C++ on a Macro Diet

(as in eat more macros, not abstain from macros)

Matvey Soloviev (Cornell University)

Graduate Seminar

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.

Only one program

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.

Sliding Scale of Expressivity

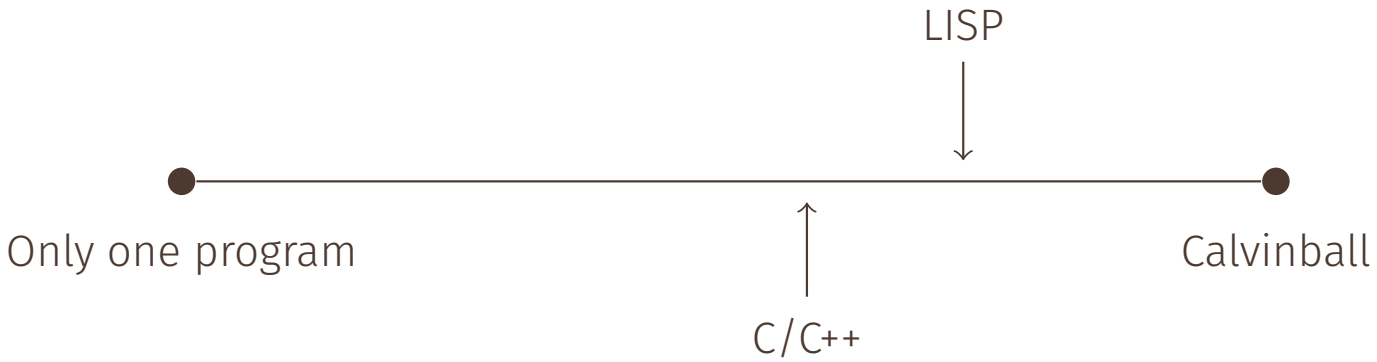
Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.

Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

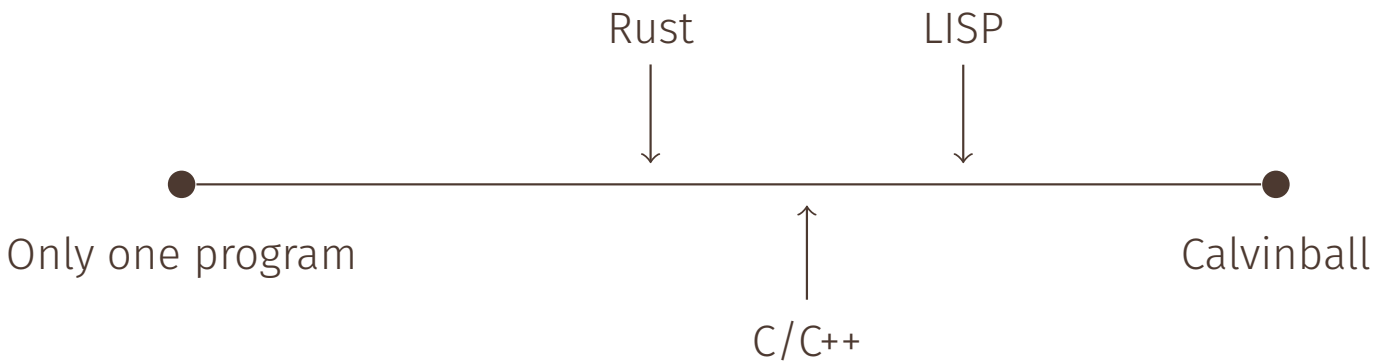
Two dimensions, but let's project down to one.



Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

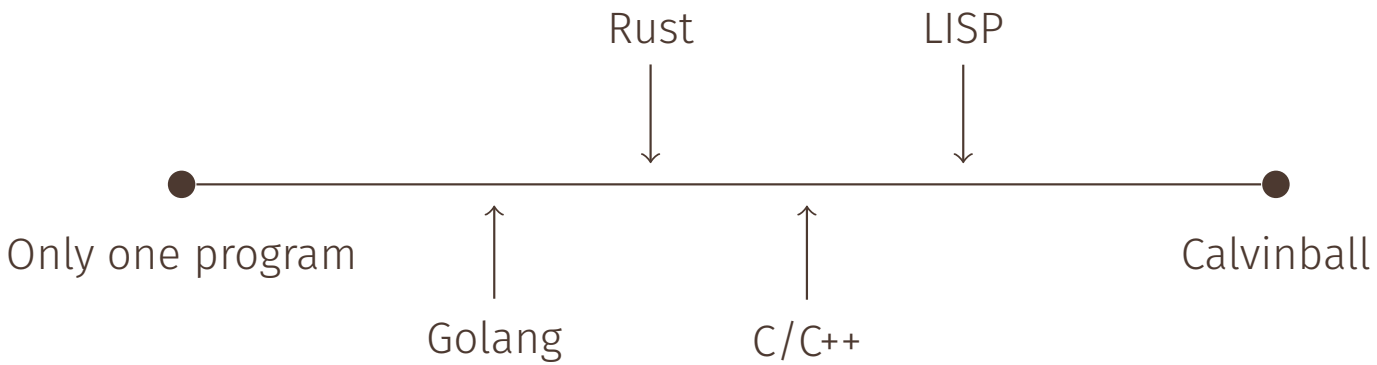
Two dimensions, but let's project down to one.



Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

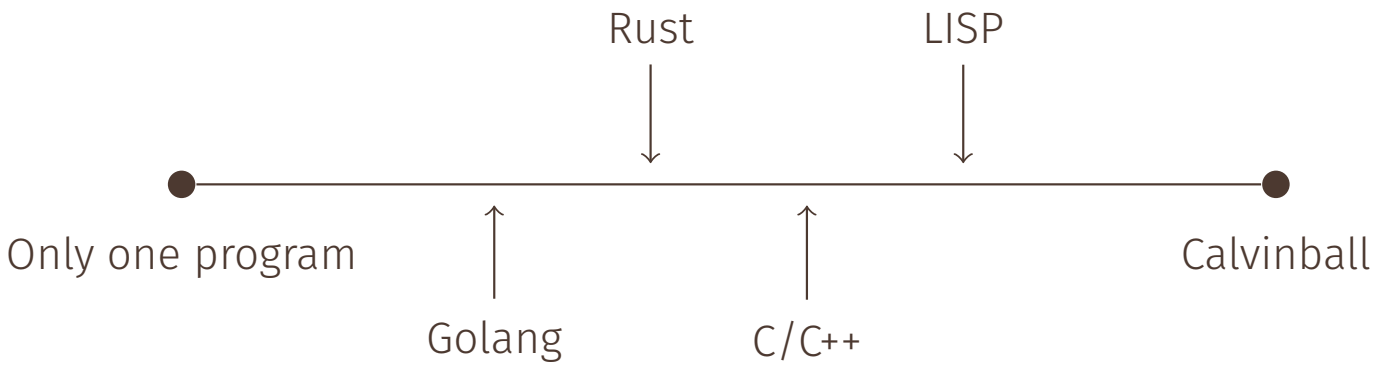
Two dimensions, but let's project down to one.



Sliding Scale of Expressivity

Expressivity: What programs can you express? How hard is it to express a given program?

Two dimensions, but let's project down to one.



General trend away from expressivity?

Types of expressivity

- Express more programs.
- Express given program in more ways. ← you are here

Types of expressivity

- Express more programs.
- Express given program in more ways. ← you are here

Why care about more ways to express a program if we already have one?

Types of expressivity

- Express more programs.
- Express given program in more ways. ← you are here

Why care about more ways to express a program if we already have one?

- Want shorter code.

Types of expressivity

- Express more programs.
- Express given program in more ways. ← you are here

Why care about more ways to express a program if we already have one?

- Want shorter code.
- Want easier to read code.

Types of expressivity

- Express more programs.
- Express given program in more ways. ← you are here

Why care about more ways to express a program if we already have one?

- Want shorter code.
- Want easier to read code.
- Want code that is closer to your mental model.

How are languages made expressive?

- Control flow: `goto`, coroutines, `break` $*n*...

How are languages made expressive?

- Control flow: `goto`, coroutines, `break` $*n*...
- Data representation: structs, inheritance, enums, ADTs...

How are languages made expressive?

- Control flow: `goto`, coroutines, `break` $*n*...
- Data representation: structs, inheritance, enums, ADTs...
- Overloading: operators, closures, `operator( )`...

How are languages made expressive?

- Control flow: `goto`, coroutines, `break` $*n*...
- Data representation: structs, inheritance, enums, ADTs...
- Overloading: operators, closures, `operator()`...
- Metaprogramming: templates, macros...

Macros

Fundamentally, macros are code (executed at compile time) that computes code.

Macros

Fundamentally, macros are code (executed at compile time) that computes code.

(Is a compiler a macro? Is `moc` a macro? No. Should also say macros must exist at the same level as the target code.)

Macros

Fundamentally, macros are code (executed at compile time) that computes code.

(Is a compiler a macro? Is `moc` a macro? No. Should also say macros must exist at the same level as the target code.)

Long history: LISP had Fexprs since the ’60s.

Why Macros? (1)

So, why macros?

Why Macros? (1)

So, why macros?

Many arguments against: make code unaccessible, hide complexity, break abstraction boundaries...

Why Macros? (1)

So, why macros?

Many arguments against: make code unaccessible, hide complexity, break abstraction boundaries...

A lot of modern languages forswear metaprogramming altogether: Golang, Python...

Why Macros? (2)

However, there are plenty of advantages:

Why Macros? (2)

However, there are plenty of advantages:

- Simplify repetitive code.
  Ever have to draw ∼16 triangles with correct normals and texture coordinates in immediate mode OpenGL?
  Now try doing 16 variants of the above in a tight inner loop (e.g. marching cubes).

Why Macros? (2)

However, there are plenty of advantages:

- Simplify repetitive code.
  Ever have to draw ∼16 triangles with correct normals and texture coordinates in immediate mode OpenGL?
  Now try doing 16 variants of the above in a tight inner loop (e.g. marching cubes).
- Zero-overhead debugging.
  Unfortunately, `-O0` still mostly means nothing is inlined, and `-O1` means the value that causes your bug has probably been optimised out.

Why Macros? (3)

- Domain-specific languages.
  Common example for C: packet (de)serialisation in network code.
  With sufficiently powerful macro system, can write shaders, packet filters etc. in-line.

Why Macros? (3)

- Domain-specific languages.
  Common example for C: packet (de)serialisation in network code.
  With sufficiently powerful macro system, can write shaders, packet filters etc. in-line.
- Configuration.
  Maybe build system fashions have moved on, but the entire Linux ecosystem was still built on autoconf.
  Macros provide a clean, programmer-controlled interface for outside tooling to reshape code and adapt it to circumstances.

Why Macros? (4)

Build your own language features.

C/C++ keep adding functionality. As it stands, all of it has to be supported by the compiler.

Why Macros? (4)

Build your own language features.

C/C++ keep adding functionality. As it stands, all of it has to be supported by the compiler.

Google “gcc compiler bug”, lots of scary examples...

Why Macros? (4)

Build your own language features.

C/C++ keep adding functionality. As it stands, all of it has to be supported by the compiler.

Google “gcc compiler bug”, lots of scary examples...

But most of the ++ part of C++ could easily be implemented by macros outputting C, given sufficient power!

Why Macros? (4)

Build your own language features.

C/C++ keep adding functionality. As it stands, all of it has to be supported by the compiler.

Google “gcc compiler bug”, lots of scary examples...

But most of the ++ part of C++ could easily be implemented by macros outputting C, given sufficient power!

(Get closer to the ideal of research languages? Lean, verified core; fancy features get converted to it at first compilation pass)

Why Macros? (4)

Build your own language features.

C/C++ keep adding functionality. As it stands, all of it has to be supported by the compiler.

Google “gcc compiler bug”, lots of scary examples...

But most of the ++ part of C++ could easily be implemented by macros outputting C, given sufficient power!

(Get closer to the ideal of research languages? Lean, verified core; fancy features get converted to it at first compilation pass)

The C preprocessor

C actually has a macro system. Unfortunately, it's rather limited:

```
1 #define TEST(1) 1+TEST(1) //this won't loop forever :(
```

There are some workarounds, but none will give you true recursion.

Also, can only create "fake variables" and "fake function calls".

Other macro systems

LISP: Arbitrary LISP code operating on LISP code.

This works because LISP code is approximately `thing::=(thing ...)|name|value`! Writing down a type of C/C++ expressions is an MEng thesis (and it probably won't be quite right).

Other macro systems

LISP: Arbitrary LISP code operating on LISP code.

This works because LISP code is approximately `thing::=(thing ...)|name|value`! Writing down a type of C/C++ expressions is an MEng thesis (and it probably won't be quite right).

Rust: DSL for matching, capturing and emitting token streams.

Good start, but the Rust team fell for the "don't surprise the reader" meme.

Towards a better macro system for C/C++

Let’s build something like that for C/C++!

Towards a better macro system for C/C++

Let's build something like that for C/C++!

In fact, I did: https://github.com/blackhole89/macros

Simple example

```
struct { int value; LinkedList *next; } LinkedList;

// define recursive macro to create a linked list
@define MakeList {
    ( {@^[,]$head, @^$tail} ) => (
        new LinkedList( {$head, MakeList {$tail}} )
    )
    ( {@^[,]$singleton} ) => (
        new LinkedList( {$singleton, NULL} )
    )
    ( {} ) => ( NULL )
}

// create a linked list with 5 elements
LinkedList *l = MakeList {1,2,3,4,5};
```

Basic summary

- `@define name {...}` creates a new macro `name`;
- `{...}` contains a series of pattern-outcome pairs.
- Whenever `name` is encountered, the parser tries to match the tokens following it to patterns in order.
- If a match succeeds, the outcome is processed and then emitted.
- Finally, processing proceeds with the first token that was not consumed by the match.

A successful pattern match may capture tokens or streams of tokens into variables, which are available for the processing of the outcome.

More features

The pattern language contains facilities for matching various common grammars such as separated lists or everything until a particular token is encountered. More complex grammars should be implemented by capturing everything and rerunning the matcher on it using `match`.

More features

The pattern language contains facilities for matching various common grammars such as separated lists or everything until a particular token is encountered. More complex grammars should be implemented by capturing everything and rerunning the matcher on it using `match`.

Also support S-expression-valued variables and iteration over them, basic arithmetic, basic string processing.

Algebraic datatypes

Can add algebraic datatypes to C++ (∼208 lines of macros):

```
datatype List<T> = Nil | Cons(T,List<T>) ;

/* flatten a list of lists */
template<class T> List<T> unions(List<List<T>> ls)
{
    match(ls) {
    case Cons(Cons(&x,&xs),&ys):
        return Cons(x, unions(Cons(xs,ys)));
    case Cons(Nil,&ys):
        return unions(ys);
    case Nil:
        return Nil<T>;
    }
}
```

Painless reflection

By re`@define`ing the keyword `class`, we can get reflection (and serialisation, and Java-style annotations...):

```
class TestClass {
    int test; // (rest omitted for space)
};

int main(int argc, char* argv[])
{
    printf("Members of class TestClass:\n");
    for( auto a : Reflect<TestClass>::members ) {
        printf("  %s\n",a.c_str());
    }
    return 0;
}
```

Thoughts

- Macros are fun. Can tweak the language to be more like you want it to be without embarking on a project to write your own language.

Thoughts

- Macros are fun. Can tweak the language to be more like you want it to be without embarking on a project to write your own language.
- For stuff like reflection, it's easy to beat ISO (does their compromise solution make anyone happy?)

Thoughts

- Macros are fun. Can tweak the language to be more like you want it to be without embarking on a project to write your own language.
- For stuff like reflection, it's easy to beat ISO (does their compromise solution make anyone happy?)
- Turns out that this approach to macros basically gives you TeX, with all its subtleties (`\expandafter`, anyone?). Many nontrivial design choices surrounding variable scope and substitution behaviour.

Future work

- Make the preprocessor more aware of C/C++. Maybe one day it will be able to distinguish less-than from template brackets.

Future work

- Make the preprocessor more aware of C/C++. Maybe one day it will be able to distinguish less-than from template brackets.
- Support more flexible grammar rules? Custom operators with fixity? (Yes, I really want C++ ∪ ML...)

Future work

- Make the preprocessor more aware of C/C++. Maybe one day it will be able to distinguish less-than from template brackets.
- Support more flexible grammar rules? Custom operators with fixity? (Yes, I really want C++ ∪ ML...)
- Once we can write Haskell as a C++ DSL, we're officially self-hosting :)

Future work

- Make the preprocessor more aware of C/C++. Maybe one day it will be able to distinguish less-than from template brackets.
- Support more flexible grammar rules? Custom operators with fixity? (Yes, I really want C++ ∪ ML...)
- Once we can write Haskell as a C++ DSL, we're officially self-hosting :)

Thanks for listening!