Turing Completeness and Template Metaprogramming

Tom Peters
thomas.d.peters@gmail.com
tdp2110 on slack, github
Feb 21, 2019

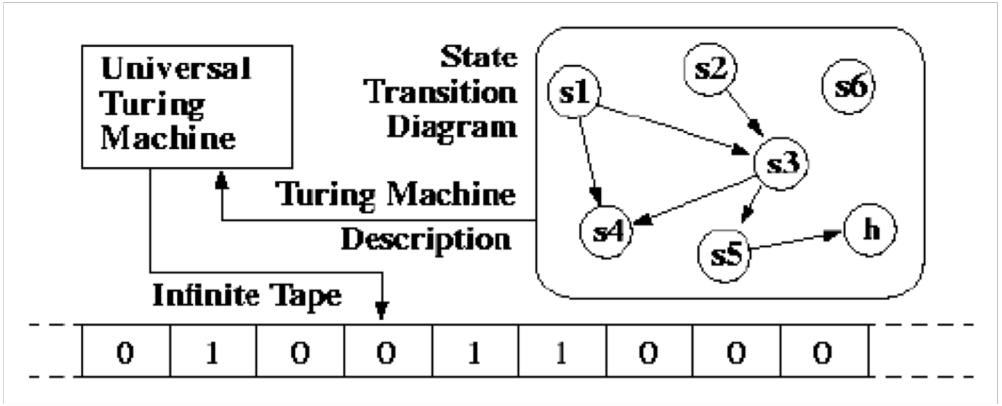
Outline

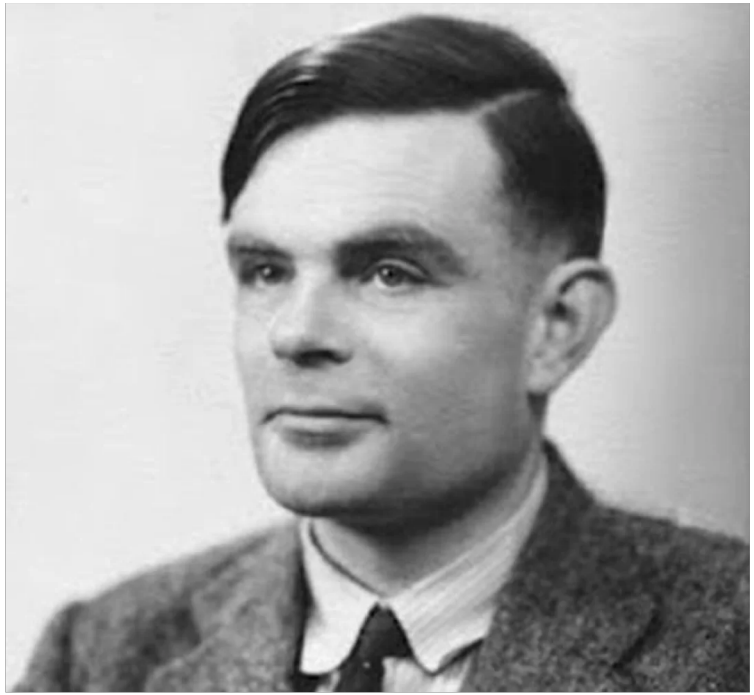
- Turing Completeness
- C++ Metaprogramming crash course
- Proving Turing completeness of Template Metaprogramming

Part 1: Turing Completeness

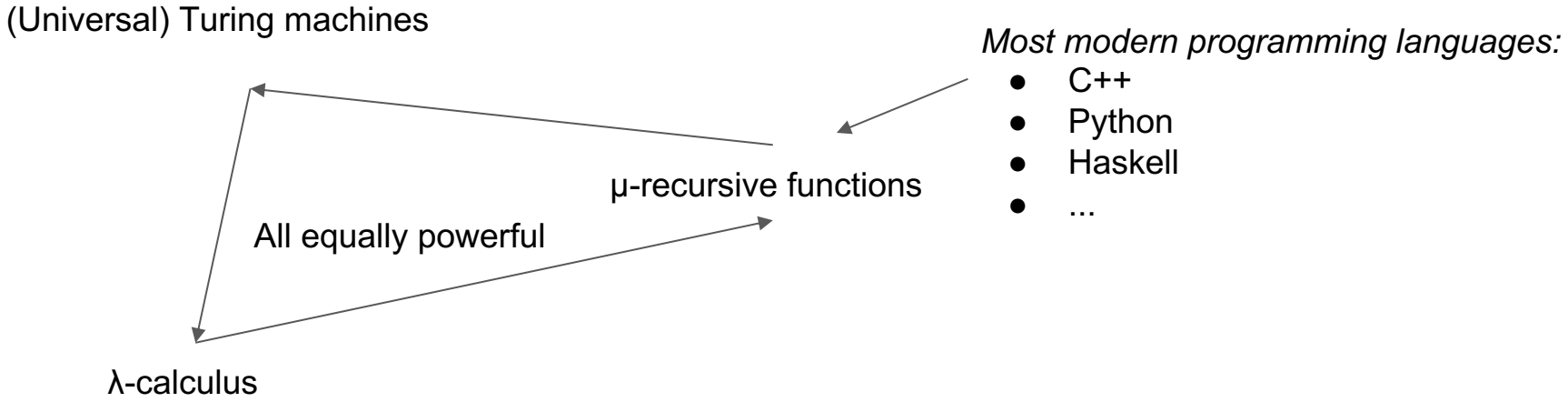
Turing machines

Alan Turing, 1936





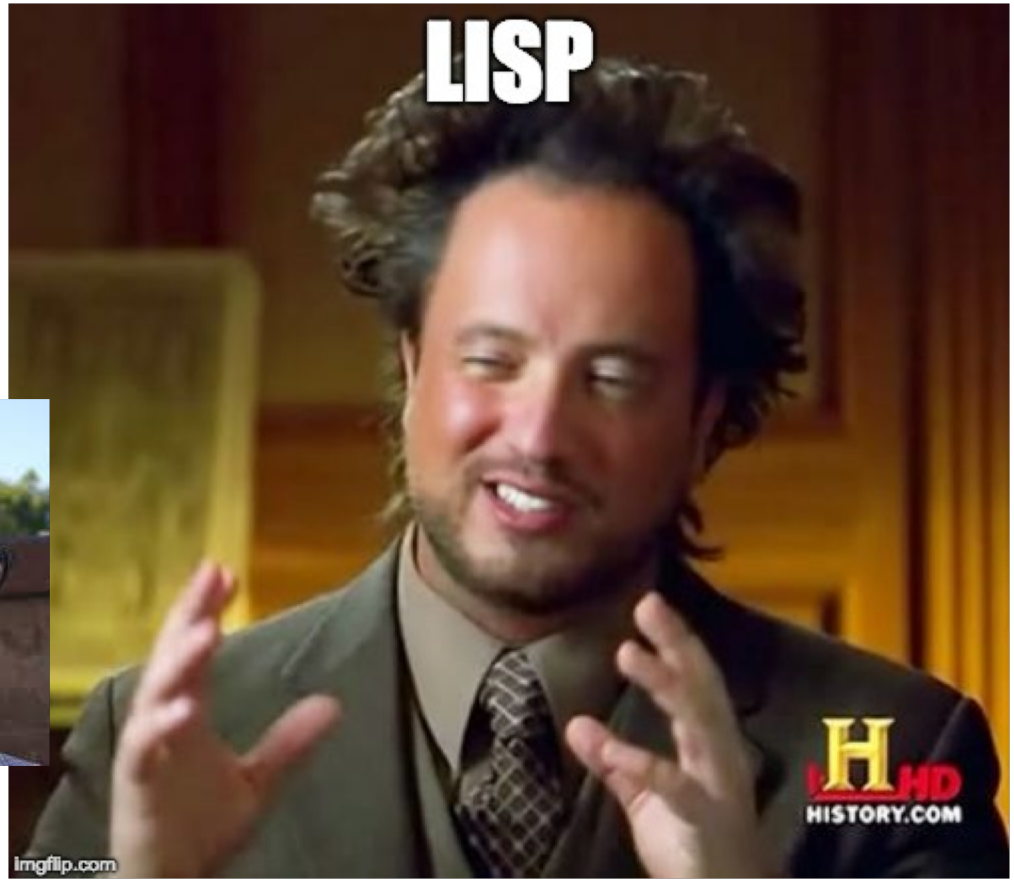
Turing Completeness

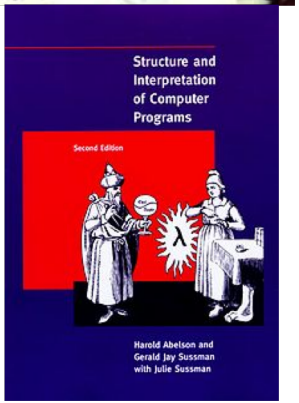


Church Turing Thesis: hypothesis that *anything* mechanically computable is computable by a Turing machine.

Compile time C++ is Turing Complete

- 1994 Erwin Unruh: enumerating primes at compile time
- Boost Spirit, Qi: compile time parser generators
- Boost Proto: compile time embedded DSL generation
- Expression templates (eg, Eigen linear algebra library)
- 2017: Compile time ray tracing
- 2018: Compile time ARM emulator
- 2018: Compile time regular expressions


LISP
HISTORY.COM
imgflip.com


Structure and
Interpretation
of Computer
Programs
Second Edition
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

Lisp

- 1958 (one year younger than fortran!)
- Innovative as heck:
  - Garbage collector
  - Recursion
  - Conditionals
  - REPL
  - Continuations
  - Multiple dispatch
  - ...

- Homoiconicity
- Metacircular evaluation

Part 2: C++ Template Metaprogramming

C++ Templates: parameterized types

```
template <class T>
class Vector { … };
```

```
Vector<int> vector_of_int;
Vector<float> vector_of_float;
```

```
template <int i>
class IntegralConstant { … };
```

```
IntegralConstant<1> one;
IntegralConstant<2> two;
```

Template specialization

```
template <class T>
class Vector { … };
```

```
// "primary template"
```

```
template <>
class Vector<bool> { … };
```

```
// specialize for vector of bool
Vector<bool> vector_of_bool;
```

```
template <class T1, class T2>
class Vector<Pair<T1, T2>> { … };
```

```
// can specialize on patterns
// of types
```

Member typedefs

```
template <class T>
class Vector {
public:
  using element_type = T;
};
```

```
Vector<int>::element_type i;

// i has type int
```

```
template <class T>
class Foo {
using bar_type =
Vector<T>::element_type;
};
```

```
// compiler error: expected typename
before Vector<t>
```

```
using bar_type = typename Vector<T>::element_type;
```

Metafunctions (via alias templates)

```
template <class T>
class Vector {
public:
  using element_type = T;
};
```

```
template <class T>
using vector_element_type = typename Vector<T>::element_type;
```

```
vector_element_type<int>  i;       // i has type int
vector_element_type<bool> b;       // b has type bool
```

More interesting metafunction:

```
template <bool, class IfTrue, class IfFalse>
struct If;

template <class IfTrue, class _>
struct If<true, IfTrue, _> { using type = IfTrue; };

template <class _, class IfFalse>
struct If<false, _, IfFalse> { using type = IfFalse; };

If<false, int, float>::type f;
If<true,  int, float>::type i;
```

Typelists

```
template  <class … Ts>
struct TypeList {};
```

```
template <class T>
struct HeadImpl;

template<class T, class … Ts>
struct HeadImpl<TypeList<T, Ts …>>
{ using type = T; };

template <class T>
using Head = typename HeadImpl<T>::type;
```

```
using Types1 = TypeList<int, float>;
Head<Types1> h1;
```

```
using Types2 = TypeList<>;
Head<Types2> h2;
```

Typelists: searching

```
template  <class … Ts>
struct TypeList {};
```

```
template <class T, class List>
struct ContainsImpl { static constexpr bool result = false; }

template <class T, class … Ts>
struct ContainsImpl<T, TypeList<T, Ts …>>
{ static constexpr bool result = true; };

template <class T, class H, class … Ts>
struct ContainsImpl<T, TypeList<H, Ts …>>
{ static constexpr bool result = ContainsImpl<T, TypeList<Ts…>::result; };

template <class T, class List>
static constexpr bool Contains<T, List> = ContainsImpl<T, List>::result;
```

static_assert

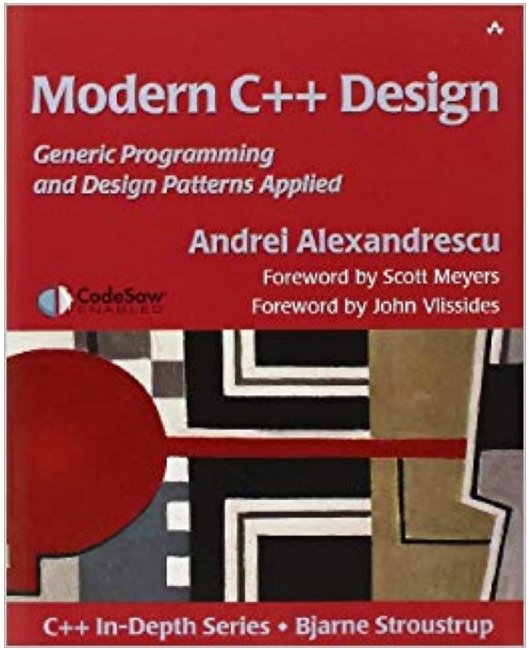
Introduces compile-time error if a (compile time) condition fails to hold:

```
#include <type_traits>      // for std::is_same_v

static_assert(std::is_same_v<int,
                             If<true, int, float>>);   // passes
static_assert(std::is_same_v<float,
                             If<true, int, float>>);   // compile-time error
```

If you're interested in practical metaprogramming ...

From 2001!



Libraries for metaprogramming

- Boost MPL: STL-modeled. "Old school"
- Boost Hana: modern take

Louis Dionne

Part 3: Turing Completeness of TMP

Booleans

```
template <bool>
struct Bool {};

using True = Bool<true>;
using False = Bool<false>;
```

Integers

```
template <int>
struct Int {};
```

Lists

```
template <class Car, class Cdr>
struct Cons {};

struct EmptyList {};
```

Strings

S-Expressions:

```
template <class Operator, class... Operands>
struct SExp {};

// (some) built-in operators:
enum class OpCode { Add,    Sub, Mul,
                     Eq,    Neq, Leq,
                   Neg,     Or, And,
                   Not,   Cons, Car,
                   Cdr, IsNull };

template <OpCode op>
struct Op {};
```

Variables

```
template <int>
struct Var {};
```

Lambdas (and closures)

```
template <class Body, class... Params>
struct Lambda {};

template <class Body, class Environment, class... Params>
struct Closure {};
```

To the CLI!

(using https://github.com/tdp2110/TmpLisp)

An example: (+ 1 2)

```
Eval<SExp<Op<OpCode::Add>, Int<1>, Int<2>>>;

typename Eval_<SExp<Op<OpCode::Add>, Int<1>, Int<2>>>::type;

Apply<Eval<Op<OpCode::Add>>, Eval<Int<1>>, Eval<Int<2>>>;

Apply<Op<OpCode::Add>, Int<1>, Int<2>>;

Int<1 + 2>;

Int<3>;
```

Example: (let ((x 2)) (+ 1 x))

```
Eval<Let<Env<Binding<Var_x, Int<2>>>, SExp<Op<OpCode::Add>, Int<1>, Var_x>>,
EmptyEnv>;
Eval<SExp<Closure<SExp<Op<OpCode::Add>, Int<1>, Var_x>, Env<Binding<Var_x, Int<2>>>>
Apply<Closure<SExp<Op<OpCode::Add>, Int<1>, Var_x>, Env<Binding<Var_x, Int<2>>>>
Eval<SExp<Op<OpCode::Add>, Int<1>, Var_x>, Env<Binding<Var_x, Int<2>>>>
Eval<SExp<Op<OpCode::Add>, Int<1>, Eval<Var_x, Env<Binding<Var_x, Int<2>>>>
Eval<SExp<Op<OpCode::Add>, Int<1>, Eval<Var_x, Env<Binding<Var_x, Int<2>>>>
Eval<SExp<Op<OpCode::Add>, Int<1>, Int<2>>>
Int<3> // from previous example
```

Greenspun's tenth rule of programming:

"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

credit??

Future directions

- Stateful operations
- Compile time parsing? :)

Thanks! Questions?