C++ Templates are Turing Complete

Todd L. Veldhuizen
tveldhui@acm.org
Indiana University Computer Science

Abstract

We sketch a proof of a well-known folk theorem that C++ templates are Turing complete. The absence of a formal semantics for C++ template instantiation makes a rigorous proof unlikely.

1 Introduction

It has been known for some time that C++ templates permit complicated computations to be performed at compile time. The first example was due to Erwin Unruh [3] who circulated a small C++ program that computed prime numbers at compile time, and listed them encoded as compiler error messages. In this short note we sketch a proof that C++ templates are Turing complete. We assume familiarity with both C++ templates and basic theory of computation; for background on Turing machines readers are referred to e.g. [2]. The proof is straightforward: we show how any Turing machine may be embedded in the C++ template instantiation mechanism, from which the result is immediate.

2 Encoding Turing machines in C++ Templates

A Turing machine is a quadruple (K, Σ, δ, s), where K is a finite set of states, Σ is an alphabet, $s \in K$ is the start state, and δ is the transition function $K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{\Leftarrow, \Rightarrow\})$. The special state h is the *halt state*, $\Leftarrow$ and $\Rightarrow$ are special symbols indicating *left* and *right*, and $\# \in \Sigma$ is the blank symbol.

To illustrate how a Turing machine may be encoded as a C++ template metaprogram, we use as an example this simple machine which replaces a string of a's with $\#$'s and then halts:

$$K = \{q_0, q_1, h\}$$
$$\Sigma = \{a, \#\}$$
$$s = q_0$$

q	σ	$\delta(q,\sigma)$
q_0	a	$(q_1, \#)$
q_0	$\#$	$(h, \#)$
q_1	a	(q_0, a)
q_1	$\#$	$(q_0, \Rightarrow)$

We encode the states K and alphabet $\Sigma \cup \{\Leftarrow, \Rightarrow\}$ as empty C++ types:

```
/* States */
struct Halt {};
struct Q0 {};
struct Q1 {};
```

```
/* Alphabet */
struct Left {};
struct Right {};
struct A {};
struct Blank {};
```

To encode the tape, we use a standard functional-style list:

```
/* Tape representation */
struct Nil { };
template<class Head, class Tail>
struct Pair {
  typedef Head head;
  typedef Tail tail;
};
```

Using these classes, the tape $a\#a$ is encoded as the C++ type `Pair<A,Pair<Blank,Pair<A,Nil> > >`. To represent the position of the Turing machine at a particular place on the tape, we split the tape into three parts: to the left, the contents of the current tape cell, and to the right. So the tape $abc\underline{d}e$, in which the Turing machine is positioned at d, would be represented as the triple (abc, d, e). To provide easy access to the tape cell directly to the left of the read head, the left tape contents are stored in reverse order. So the tape $abc\underline{d}e$ would be encoded as these three types:

abc	`Pair<C,Pair<B,Pair<A,Nil> > >`
d	`D`
e	`Pair<E,Nil>`

The transition function $\delta(q, \sigma)$ maps from the current state q and contents of the tape cell σ to the succeeding state and action (character to be written, $\Leftarrow$ or $\Rightarrow$). To realize δ in templates, we provide specializations of a template class `TransitionFunction<State,Character>`. Inside each instance are `typedef`s for `next_state` and `action`, which encode (respectively) the next state and action:

```
/* Transition Function */
template<typename State, typename Character>
struct TransitionFunction { };
/* q0 a -> (q1,#) */
template<> struct TransitionFunction<Q0,A> {
 typedef Q1    next_state;
 typedef Blank action;
};
/* q0 # -> (h,#)  */
template<> struct TransitionFunction<Q0,Blank> {
 typedef Halt  next_state;
 typedef Blank action;
};
/* q1 a -> (q0,a) */
template<> struct TransitionFunction<Q1,A> {
 typedef Q0    next_state;
 typedef A     action;
};
/* q1 # -> (q0,->) */
template<> struct TransitionFunction<Q1,Blank> {
 typedef Q0    next_state;
 typedef Right action;
};
```

A *configuration* is a member of $K \times \Sigma^* \times \Sigma \times \Sigma^*$ and represents the state of the machine and tape at a single point in the computation. We encode a configuration as an instance of the template class `Configuration<>`, which takes these template parameters:

Template parameter	Meaning
`State`	Current state of the machine
`Tape_Left`	Contents of the tape to the left of the read head (in reverse order)
`Tape_Current`	Content of the tape cell under the read head
`Tape_Right`	Contents of the tape to the right of the read head
`Delta`	Transition function

Inside the class `Configuration<>`, the `next_state` and `action` are computed by evaluating $\delta(q, \sigma)$, and a helper class `ApplyAction` is instantiated to compute the next configuration:

```
/* Representation of a Configuration */
template<typename State,
  typename Tape_Left,
  typename Tape_Current,
  typename Tape_Right,
  template<typename Q, typename Sigma> class Delta>
struct Configuration {
  typedef typename Delta<State,Tape_Current>::next_state
      next_state;
  typedef typename Delta<State,Tape_Current>::action
      action;
  typedef typename ApplyAction<next_state, action,
    Tape_Left, Tape_Current, Tape_Right,
    Delta>::halted_configuration
        halted_configuration;
};
```

The class `ApplyAction` has five versions, to handle:

- Writing a character to the current tape cell;
- Transitioning to the *halt* state;
- Moving left;
- Moving right;
- Moving right when at the rightmost non-blank cell on the tape.

Each of these instantiates the next `Configuration<>`, and recursively defines the `halted_configuration`.

```
/* Default action: write to current tape cell */
template<typename NextState, typename Action,
  typename Tape_Left, typename Tape_Current,
  typename Tape_Right,
  template<typename Q, typename Sigma> class Delta>
struct ApplyAction {
  typedef Configuration<NextState, Tape_Left,
    Action, Tape_Right, Delta>::halted_configuration
      halted_configuration;
};

/* Move read head left */
template<typename NextState,
  typename Tape_Left, typename Tape_Current,
  typename Tape_Right,
  template<typename Q, typename Sigma> class Delta>
struct ApplyAction<NextState, Left, Tape_Left,
  Tape_Current, Tape_Right, Delta>
{
  typedef Configuration<NextState,
    typename Tape_Left::tail,
    typename Tape_Left::head,
    Pair<Tape_Current,Tape_Right>,
    Delta>::halted_configuration
      halted_configuration;
};

/* Move read head right */
template<typename NextState, typename Tape_Left,
  typename Tape_Current, typename Tape_Right,
  template<typename Q, typename Sigma> class Delta>
struct ApplyAction<NextState, Right, Tape_Left,
  Tape_Current, Tape_Right, Delta>
{
  typedef Configuration<NextState,
    Pair<Tape_Current,Tape_Left>,
    typename Tape_Right::head,
    typename Tape_Right::tail,
    Delta>::halted_configuration
      halted_configuration;
};
/*
 * Move read head right when there are no nonblank characters
 * to the right -- generate a new Blank symbol.
 */
template<typename NextState, typename Tape_Left,
  typename Tape_Current,
  template<typename Q, typename Sigma> class Delta>
struct ApplyAction<NextState, Right, Tape_Left,
  Tape_Current, Nil, Delta>
{
  typedef Configuration<NextState,
    Pair<Tape_Current,Tape_Left>,
    Blank, Nil, Delta>::halted_configuration
      halted_configuration;
};

template<typename Action, typename Tape_Left,
  typename Tape_Current, typename Tape_Right,
  template<typename Q, typename Sigma> class Delta>
struct ApplyAction<Halt, Action, Tape_Left,
  Tape_Current, Tape_Right, Delta>
{
  /*
   * We halt by not declaring a halted_configuration.
   * This causes the compiler to display an error message
   * showing the halting configuration.
   */
};
```

To "run" the Turing machine, we instantiate `Configuration<>` on an appropriate starting configuration. For example, to apply the machine to the string aaa, we use the starting configuration $(q_0, \underline{a}aa)$:

```
/*
 * An example "run": on the tape aaa starting in state q0
 */
typedef Configuration<Q0, Nil, A, Pair<A,Pair<A,Nil> >,
  TransitionFunction>::halted_configuration Foo;
```

When compiled with g++, this generates the error messages shown in Figure 1; the errors show a trace of the machine from its starting configuration $(q_0, \underline{a}aa)$ to its halting configuration $(h, \#\#\#\underline{\#})$.

```
turing.cpp: In instantiation of ‘Configuration<Q0,Pair<Blank,Pair<Blank,Pair<Blank,Nil> > >,Blank,Nil,
  TransitionFunction>’:
turing.cpp:82:   instantiated from ‘Configuration<Q1,Pair<Blank,Pair<Blank,Nil> >,Blank,Nil,TransitionFunction>’
turing.cpp:82:   instantiated from ‘Configuration<Q0,Pair<Blank,Pair<Blank,Nil> >,A,Nil,TransitionFunction>’
turing.cpp:82:   instantiated from ‘Configuration<Q1,Pair<Blank,Nil>,Blank,Pair<A,Nil>,TransitionFunction>’
turing.cpp:82:   instantiated from ‘Configuration<Q0,Pair<Blank,Nil>,A,Pair<A,Nil>,TransitionFunction>’
turing.cpp:82:   instantiated from ‘Configuration<Q1,Nil,Blank,Pair<A,Pair<A,Nil> >,TransitionFunction>’
turing.cpp:82:   instantiated from ‘Configuration<Q0,Nil,A,Pair<A,Pair<A,Nil>>,TransitionFunction>’
turing.cpp:163:   instantiated from here
turing.cpp:91: no type named ‘halted_configuration’ in ‘struct ApplyAction<Halt,Blank,Pair<Blank,
  Pair<Blank,Pair<Blank,Nil> > >,Blank,Nil,TransitionFunction>’
```

Figure 1: Compiler errors from g++ 2.95.2. Reading the error messages backwards, one sees the configuration trace $(q_0, \underline{a}aa) \vdash_M (q_1, \underline{\#}aa) \vdash_M (q_0, \#\underline{a}a) \vdash_M (q_1, \#\underline{\#}a) \vdash_M (q_0, \#\#\underline{a}) \vdash_M (q_1, \#\#\underline{\#}) \vdash_M (q_0, \#\#\#\underline{\#}) \vdash_M (h, \#\#\#\underline{\#})$.

3 C++ Templates are Turing Complete

In the previous section, we gave an encoding of a simple Turing machine in C++ templates. It is straightforward to encode *any* Turing machine in such a manner, by defining appropriate alphabet and state types, and defining the relevant specializations of a transition function.

Let M be a Turing machine and α a starting configuration. Suppose $\alpha \vdash_M s_1 \vdash_M s_2 \vdash_M \ldots$ is a trace of the Turing machine. The following lemma states that if you compile a C++ program encoding M with an initial configuration corresponding to α, then the C++ compiler will produce instantiations of the `Configuration` template corresponding to $s_1, s_2, \ldots$. An important qualification is that we assume a C++ compiler without limits on the number of template instantiations it will produce.

Lemma 1. *Let M be a Turing machine, and α a starting configuration. Let p be a C++ program encoding the machine M and configuration α as outlined in the previous section. Let $\phi : (K \times \Sigma^* \times \Sigma \times \Sigma^*) \rightarrow$* `type` *be a map that encodes configurations of M as instances of the template type* `Configuration`*. If $\alpha \vdash_M^* \beta$, then a C++ compiler without instantiation limits will, in compiling p, instantiate $\phi(\beta)$.*

A formal proof of Lemma 1 presents problems, since one would have to define formally the semantics of C++ template instantiation, something that to our knowledge has never been attempted. I believe its truth would be apparent to anyone familiar with C++ template instantiation willing to comb through the encoding of the previous section.

Theorem 1. *In the absence of instantiation bounds, C++ templates are Turing-complete.*

Proof. Immediate from the construction of the previous section and Lemma 1. □

A universal Turing machine is a special case of a Turing machine; thus UTMs can be implemented by C++ templates. The usual diagonalization argument for undecidability applies. Therefore:

Corollary 1. *In the absence of instantiation limits, whether a C++ compiler will halt when compiling a given program is undecidable.*

In recognition of this difficulty, the C++ standards committee allows conforming compilers to limit the depth of "recursively nested template instantiations," with a recommended minimum limit of 17 [1]. Compilers have adopted this limit, many with an option to increase it.

```
template<int Depth, int A, typename B>
struct K17 {
  static const int x =
    K17<Depth+1, 0, K17<Depth,A,B> >::x
  + K17<Depth+1, 1, K17<Depth,A,B> >::x
  + K17<Depth+1, 2, K17<Depth,A,B> >::x
  + K17<Depth+1, 3, K17<Depth,A,B> >::x
  + K17<Depth+1, 4, K17<Depth,A,B> >::x;
};
template<int A, typename B>
struct K17<16,A,B> {
  static const int x = 1;
};
static const int z = K17<0,0,int>::x;
```

Figure 2: A standard-conforming C++ program which does not exceed the limit of 17 recursively nested template instantiations, but nevertheless instantiates 5^{17}=762,939,453,125 templates.

This limit does not translate into any reliable time or space bound on compiles, though; it is straightforward to construct C++ programs which instantiate k^{17} templates (i.e. within the recommended limit) for any arbitrarily large k; see Figure 2 for an example with $k = 5$.

References

[1] ANSI/ISO. *Working Paper for Draft Proposed International Standard for Information Systems– Programming Language C++.* Washington DC, April 1995. Doc. No. ANSI X3J16/95-0087 ISO WG21/N0687.

[2] Lewis, H. R., and Papadimitriou, C. H. *Elements of the theory of computation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[3] Unruh, E. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.