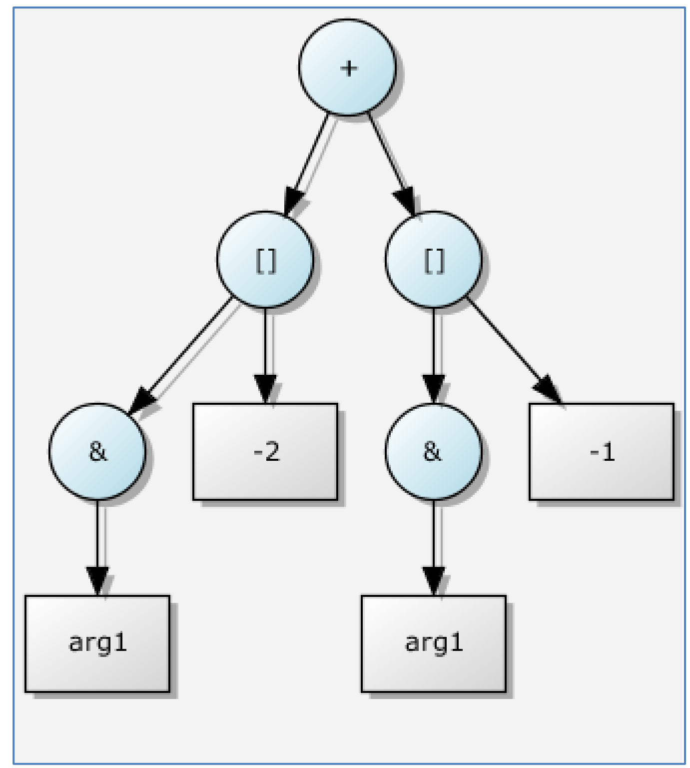
Trouble With Tuples

Compile Time Considerations of Variadic Templates

Why Care About Tuples?

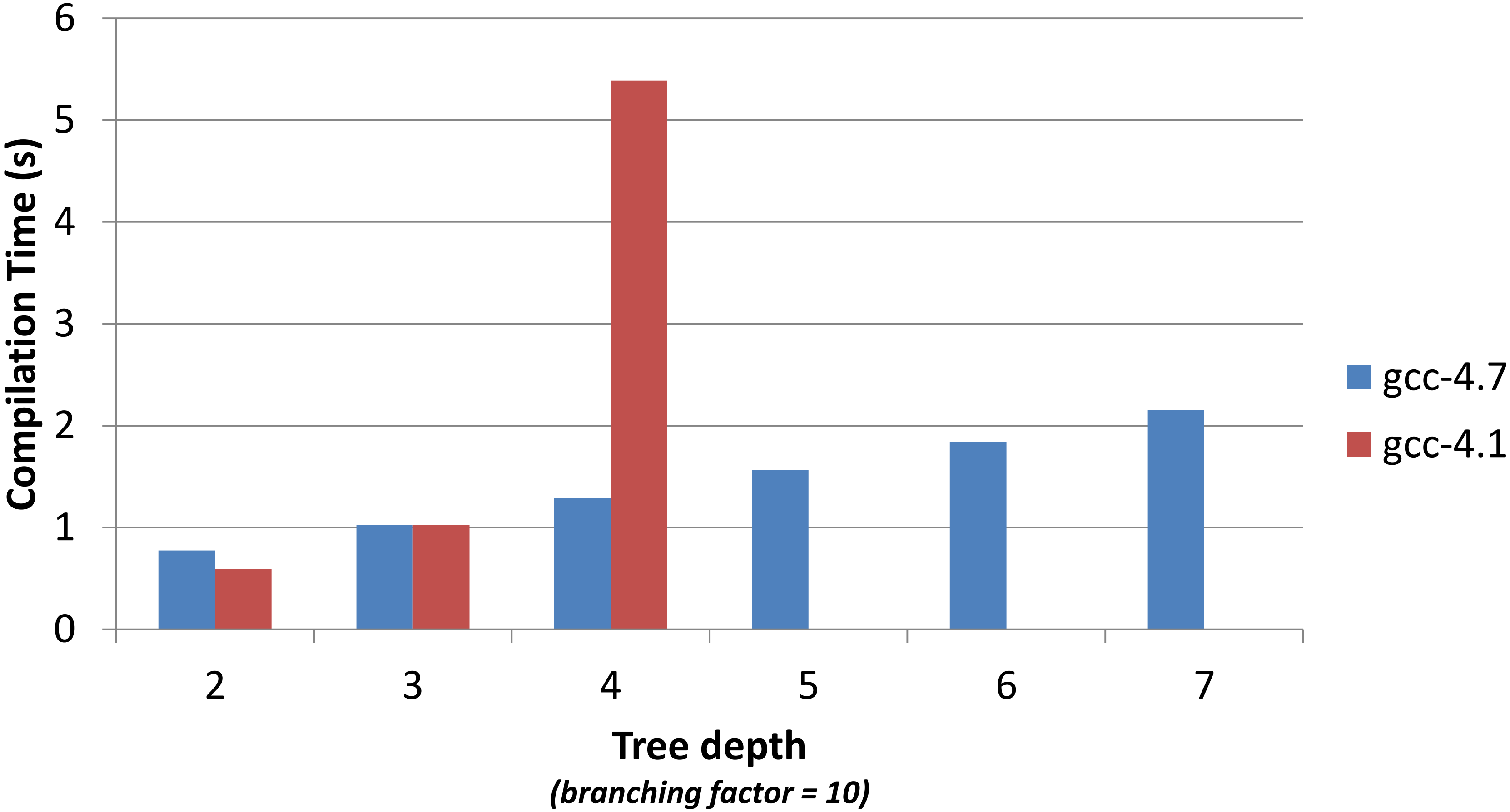
- Motivation: Proto expression trees

```
(&arg1)[-2] + (&arg2)[-1]
```

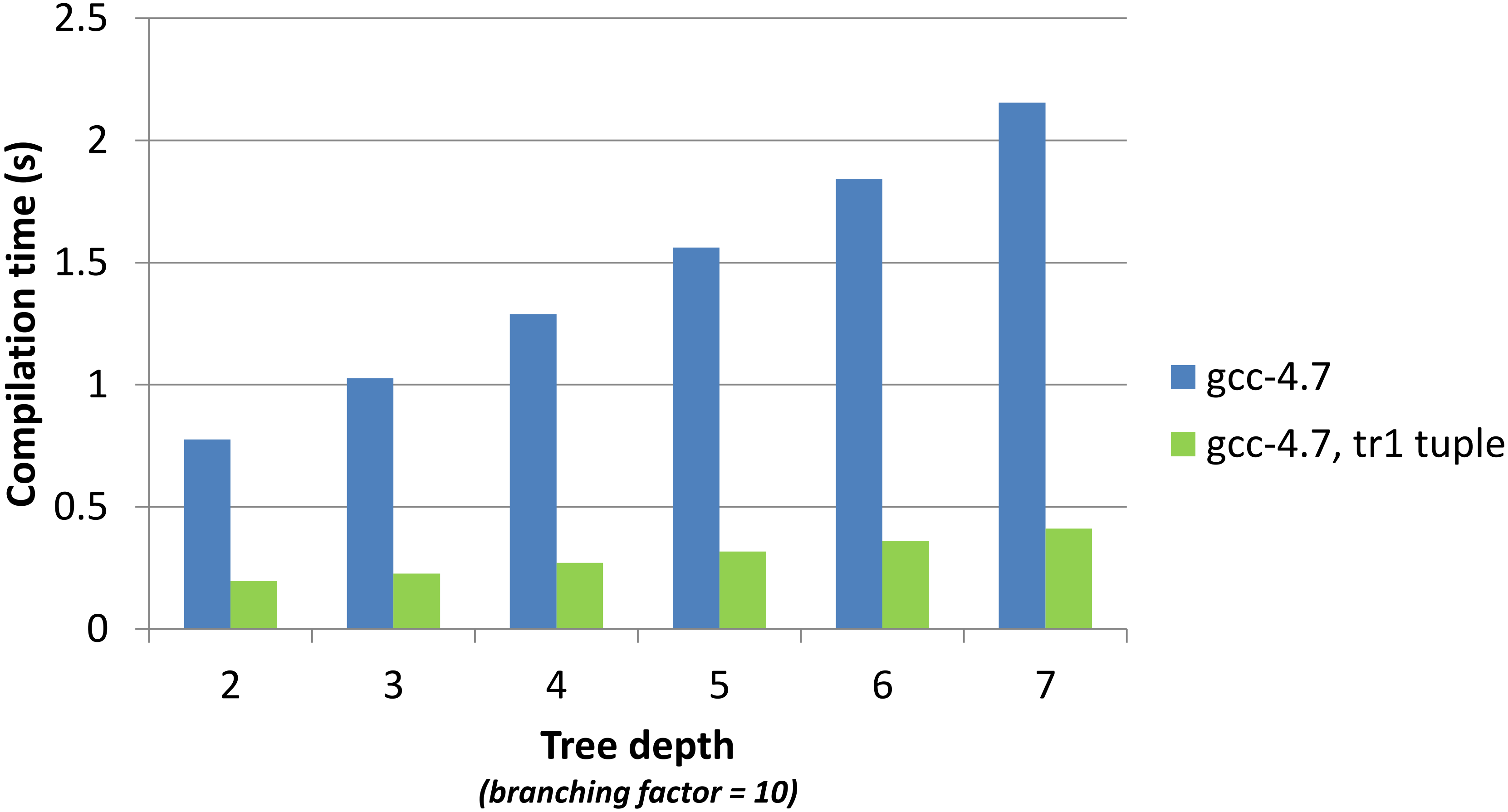



```
tuple<tuple<…>, tuple<…>>
```

Death Match! gcc 4.7 vs. gcc 4.1



Death Match Redux



TR1 Tuple

```
// TR1 tuple can only hold a fixed number of elements.
// It is implemented with lots of specializations.
template<class T0 = void, class T1 = void, class T2 = void, … class T9 = void>
struct tuple;

template<>
struct tuple<> {};

template<class T0>
struct tuple<T0>
{
    T0 t0;
    tuple() : t0() {}
    explicit tuple(T0 const &a0) : t0(a0) {}
    /* … */
};

template<class T0, class T1>
struct tuple<T0, T1>
{
    T0 t0;
    T1 t1;
    tuple() : t0(), t1() {}
    tuple(T0 const &a0, T1 const &a1) : t0(a0), t1(a1) {}
    /* … */
};

/* … ad infinitum, ad nauseum … */
```

Specialization for 0-element tuples

Specialization for 1-element tuples

Specialization for 2-element tuples

… and so on.

C++11 Tuple

```cpp
// In C++11, tuple is implemented by using pack
// expansion to inherit from a bunch of tuple
// element wrappers.

template<int I, class T>
struct tuple_elem
{
     T value;

     template<class U>
     explicit tuple_elem(U && u)
       : value(std::forward<U>(u)) {}
     /* ... */
 };

template<int... I>
struct ints;
```

```cpp
// get<1>(tup) gets the 1st elem from tup
template<int I, class T>
T & get(tuple_elem<I, T> & elem) noexcept
{
     return elem.value;
}
```

```cpp
template<class Ints, class ...T>
struct tuple_impl;

template<int... Ints, class ...T>
struct tuple_impl<ints<Ints...>, T...>
  : tuple_elem<Ints, T>...
{
    template<class ...U>
    explicit tuple_impl(U &&... u)
      : tuple_elem<Ints, T>(std::forward<U>(u))...
    {}
};

template<class ...T>
struct tuple
  : tuple_impl<indices<sizeof...(T)>, T...>
{
    template<class ...U>
    explicit tuple(U &&...u)
      : tuple_impl<indices<sizeof...(T)>, T...>(
            std::forward<U>(u)...)
    {}
};
```

In C++11, a tuple of N elements requires O(N) template instantiations.

The Ugly Truth About Variadics

- No random access into a parameter pack
  - *It’s like a Forward Range*
- Can’t store a parameter pack as a data member
  - *‘cause it ain’t a first class thingy*

Tuple: A Hybrid Approach

- Use preprocessor to handle up to N elements
- Recurse to handle next N elements
- Still O(N), but hopefully better

Unrolled Tuple

```
// C++11 tuple uses loop unrolling to bring
// down TMP overhead.
template<class ...T>
struct tuple;

template<>
struct tuple<> {};

template<class T0>
struct tuple<T0>
{
    T0 t0;
    template<class U0>
    explicit tuple(U0 && a0) : t0(std::forward<U0>(a0)) {}
    /* … */
};

/* … */

template<class T0, class T1, class T2, class ...Rest>
struct tuple<T0, T1, T2, Rest...>
{
    T0 t0; T1 t1; T2 t2;
    tuple< Rest... > tail; // Recursion here!!!
    template<class U0, class U1, class U2, class ...V>
    tuple(U0 && a0, U1 && a1, U2 && a2, V &&... v)
      : t0(std::forward<U0>(a0)), t1(std::forward<U1>(a1)), t2(std::forward<U2>(a2))
      , tail(std::forward<V>(v)...) {}
    /* … */
};
```

Specialization for 0-element tuples

Specialization for 1-element tuples

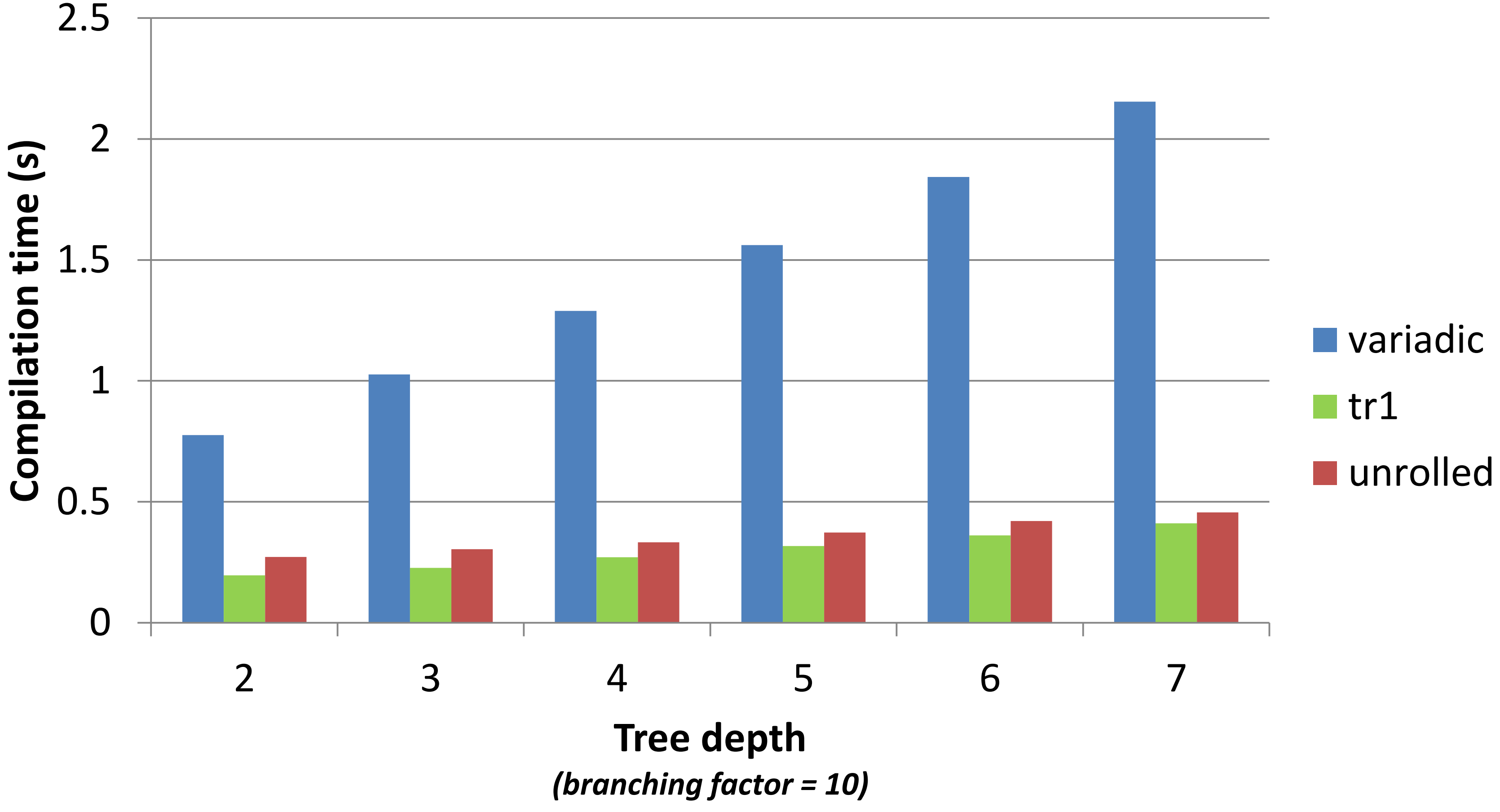
Specializations for 2-and 3-element tuples (not shown)

Recursive specialization for >3-element tuples

Unrolled Tuple get()

```
// C++11 eliminates the need for macros! Oh, darn...
#define RETURN(...) -> decltype(__VA_ARGS__) { return __VA_ARGS__; }
template<int I> using int_ = std::integral_constant<int, I>;

namespace detail {
    template<class Tuple>
    auto get_elem(Tuple && tup, int_<0>)
    RETURN((std::forward<Tuple>(tup).t0)) // extra parens are significant!

    /* … get_elem for 1st and 2nd elements defined similarly … */

    template<class Tuple, int I>
    auto get_elem(Tuple && tup, int_<I>)
    RETURN(get_elem(std::forward<Tuple>(tup).tail, int_<I-3>()))
}

template<int I, class ...T>
auto get(tuple<T...> & tup) RETURN(detail::get_elem(tup, int_<I>()))

template<int I, class ...T>
auto get(tuple<T...>  const & tup) RETURN(detail::get_elem(tup, int_<I>()))

template<int I, class ...T>
auto get(tuple<T...> && tup) RETURN(detail::get_elem(std::forward<tuple<T...>>(tup), int_<I>()))
```

Death Match Re-redux



A Possible Solution for C++1x

- When not part of a pack expansion expression, a parameter pack *is* a tuple; albeit of a built-in type.

```
template<class ...T>
struct tuple : T // Look ma! No pack expansion
{
    template<class ...U>
    explicit tuple(U &&...u) : T(u) {}
};
```

- Overloads of std::get could operate on built-in tuples, too.

A Possible Solution for C++1x

- Parameter packs are 1st class objects:
  - Stored in variables
  - Returned from functions
- Built-in tuples can still be expanded like packs.
- Add an overloadable pack expansion operator…

```
template<class ...T>
void foo(T &&... t) { /* ... */ }

template<class ...T>
void bar(T &&... t)
{
    auto tup = t; // It's a built-in tuple.
    foo(tup...);  // But it can still be expanded.
}
```

```
template<class ...T>
struct tuple // A tuple that can be expanded!
{
    T elems;

    template<class ...U>
    explicit tuple(U &&...u) : elems(u) {}

    // Explicit pack expansion operator.
    operator T...() { return elems...; }

    // Implicit conversion to built-in tuple.
    operator T &() { return elems; }
};
```

‘Nuther Possible Solution

- Courtesy of Richard Smith, Clang dev
- Allow expanded pack expressions to be members.
- Add an infix N…M pack expression

```
template<typename ...T>
struct tuple
{
    T ...values;

    T &...get_impl(mpl::size_t< 0 ... sizeof...(T)-1 >)
    {
        return values;
    }
};

template<size_t N, typename...T>
auto get(tuple<T...> &t)
    -> decltype(t.get_impl(mpl::size_t<N>()))
{
    return t.get_impl(mpl::size_t<N>());
}
```

‘Nuther Possible Solution, cont.

- Add an overloadable prefix operator ... for custom pack expansion

This:

```
template<typename...T>
struct tuple
{
   T ...values;

   T operator...() const { return values; }
};



f(Wrap(...my_tuple)...);
```

expands to:

```
template<>
struct tuple<int, char>
{
   int values$0;
   char values$1;
   int operator...$0() const { return values$0; }
   char operator...$1() const { return values$1; }
};

f(Wrap(my_tuple.operator...$0()),
  Wrap(my_tuple.operator...$1()));
```

Conclusions

- Variadics rule, but …
- There are artificial limitations that inflate compile times.
- These can be partly worked around with preprocessor repetition.
- Some simple(?) language extensions could improve the situation.