Domain-specific Language Integration with Compile-time Parser Generator Library

Zoltán Porkoláb, Ábel Sinkovics
ELTE, Budapest, Hungary

Outline

- DSL integration techniques
- Our concept
- Implementation details
- Practical example: typesafe printf
- Summary

DSLs

- Express problems in a particular domain
- Expressive
- Reflecting domain notations
- Embedding into a host language
- C++ examples
  - boost::xpressive
  - boost::proto
  - boost::spirit

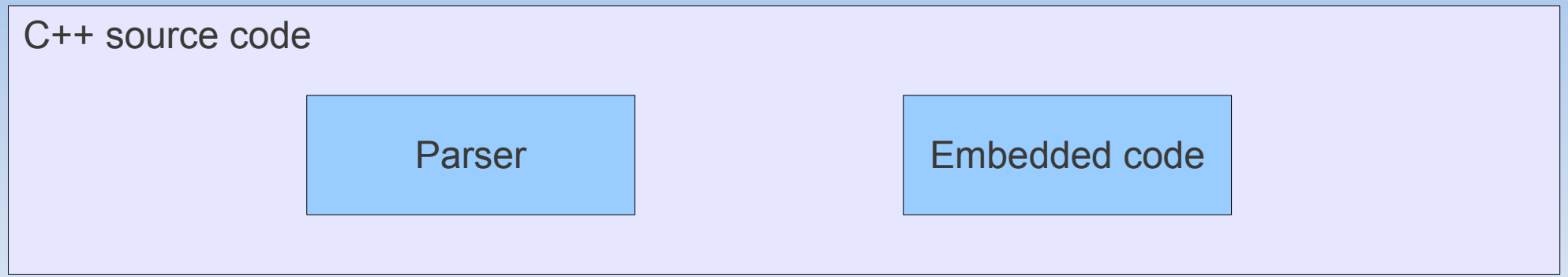
Integration techniques

- External frameworks
- Language extensions
- New languages designed for extension
- Generative approach
  - Using the C++ compiler itself
  - No need for external tools
  - DSL interacts with C++ code

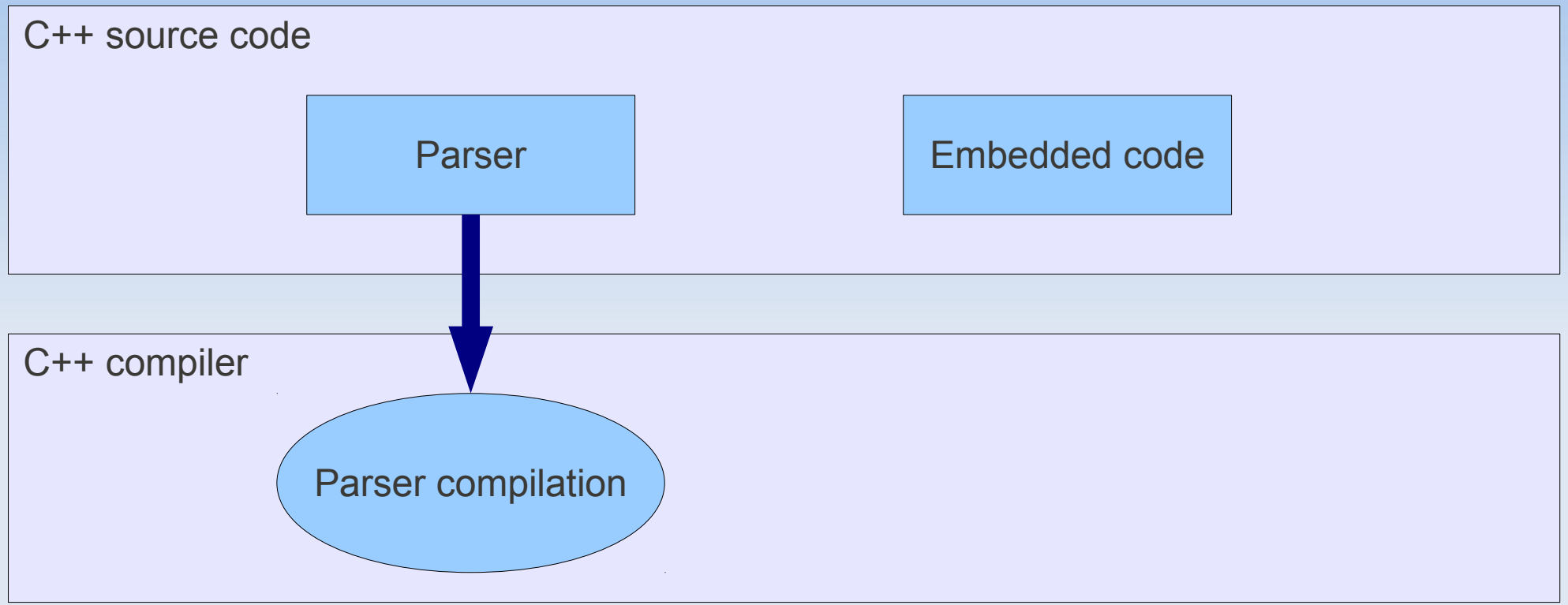
Our solution

- Write parsers in C++ template metaprograms
- Parsing happens at compile-time
- The parsed code becomes part of the compiled program
- Imagine spirit ”running” at compile-time

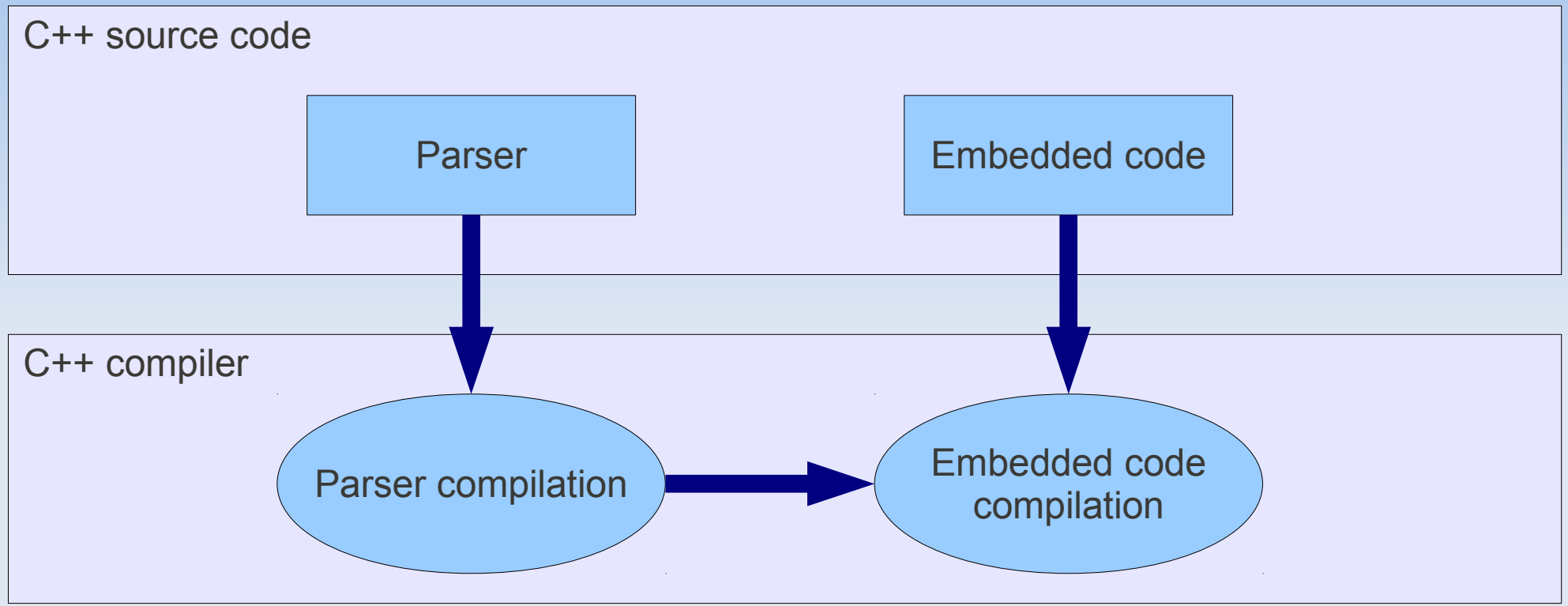
Our solution



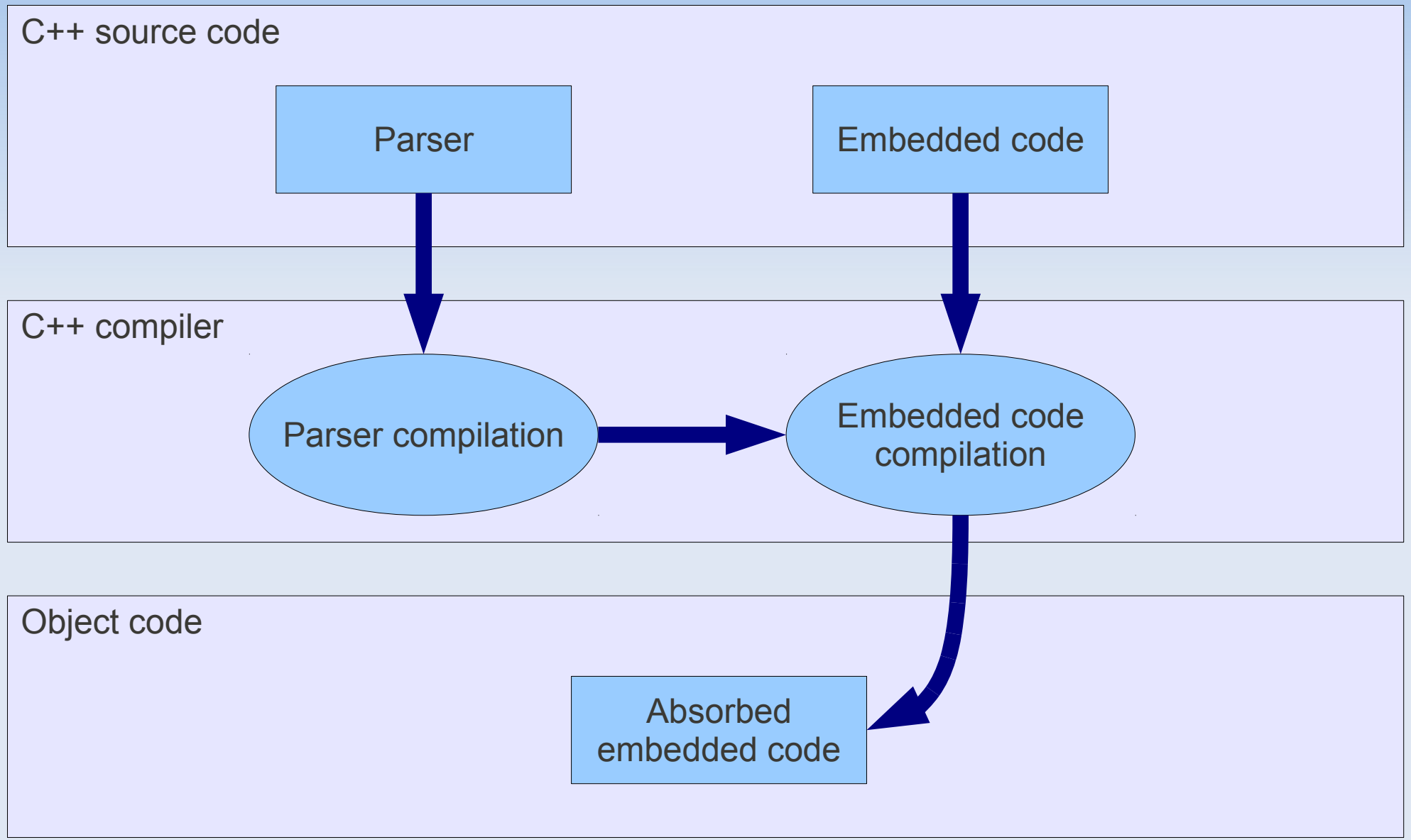
Our solution



Our solution



Our solution



C++ Template metaprogramming

- Templates are used in a special way to make the compiler execute a desired algorithm at compile-time
- The metaprogram affects the compilation process and the compiled code
- A Turing-complete sublanguage of C++

The input of the parsers

- Input of the parser is the embedded source code

```
boost::mpl::string<'Hell','o Wo','rld'>
```

The input of the parsers

- Input of the parser is the embedded source code

```
boost::mpl::string<'Hell','o Wo','rld'>
```

```
_S("Hello World")
```

Building the parser

- Parser generator library
- Strong connection between C++ template metaprogramming and the functional paradigm
- Port a parser generator library written in Haskell
- Based on parser combinators

The translation process

```
type Parser a = String -> Maybe (a, String)
```

The translation process

```
type Parser a = String -> Maybe (a, String)
```

- Template metafunction class
- Takes input string as argument
- Returns a pair of parsed object and remaining string or a special class, nothing

Simple parsers

- return_, fail
- one_char

Simple parsers

- return_, fail
- one_char

```
struct one_char
{
  template <class s>
  struct apply
  {
    typedef
      mpl::pair<
        typename mpl::front<s>::type,
        typename mpl::pop_front<s>::type
      >
      type;
  };
};
```

Parser combinators

- transform
- accept_when
- any, any1
- sequence

Parser combinators

- transform
- accept_when
- any, any1
- sequence

```
typedef accept_when<one_char, is_digit> accept_digit;

typedef any1<digit> accept_digit_sequence;
```

Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```

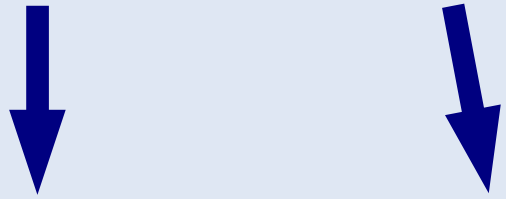
Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```



```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Grammar

```
S ::= CHARS (PARAM CHARS)*
```

```
struct S :
  second_of<
    NormalChars,
    any<
      first_of<
        Parameter,
        NormalChars
      >
    >
  >
  {};
```

Grammar

```
S ::= CHARS (PARAM CHARS)*
```

```
struct S :
  second_of<
    NormalChars,
    any<
      first_of<
        Parameter,
        NormalChars
      >
    >
  >
  {};
```

Example usage

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Example usage

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Compile-time parser

```
mpl::list<int, int, int>
```

Example usage

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Compile-time parser

```
mpl::list<int, int, int>
```

=

```
mpl::list<int, int, int>
```

Implementation

```
template <                                        >
int safe_printf(                        )
{




}
```

Implementation

```
template <class S                                    >
int safe_printf(                    )
{
}
```

Implementation

```
template <class S                                   >
int safe_printf(   a1,    a2,    a3)
{




}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{





}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{

      typename mpl::apply<printf_grammar, S>::type



}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{


      typename mpl::apply<printf_grammar, S>::type
      mpl::list<T1, T2, T3>



}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{

    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >


}
```

Implementation

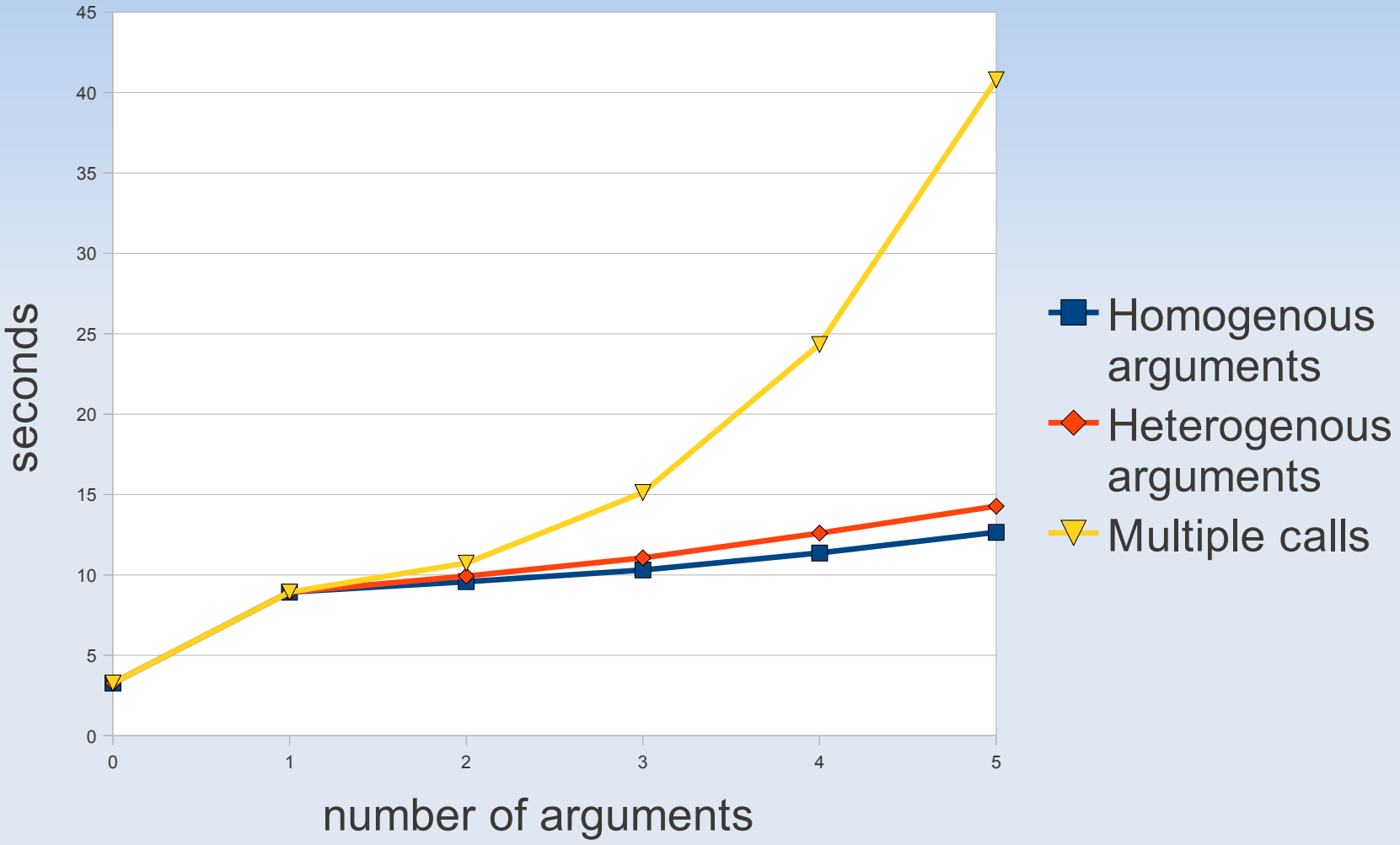
```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{
  BOOST_STATIC_ASSERT(
    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >
  );

}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{
  BOOST_STATIC_ASSERT(
    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >
  );
  return printf(mpl::c_str<S>::type::value, a1, a2, a3);
}
```

Performance



Conclusion

- Embedded languages are often used
- Integration of an embedded language is difficult
- The C++ compiler can be utilised to parse the embedded language
- This has a heavy cost at compile time
- It has no cost at runtime

Long term plans

- C++ template metaprograms are not maintenable
- They follow the functional paradigm
- They should be written in a high-level functional language
- C++ source code should remain assembly of metaprograms
- We are planning to compile Haskell code to template metaprograms using this tool

Q & A

http://abel.web.elte.hu/mpllibs/

Zoltán Porkoláb <gsd@elte.hu>
Ábel Sinkovics <abel@elte.hu>