Boost Phoenix V3

Thomas Heller
Hartmut Kaiser

Thomas Heller

Overview

- What's Phoenix?
  - Short History, Why a new version, Motivation
  - Phoenix as C++ in C++
  - Building a simple Asio echo server
- Boost Proto: The Phoenix' Workhorse
- Phoenix' Extension Mechanism
  - Building a parallel for construct
- Code as Data
- Compile Times

What's Boost Phoenix?

What's Phoenix? Short history

- Together with Joel de Guzman and Eric Niebler, Thomas implemented a new incarnation of the Boost.Phoenix library
- Started off as a Google Summer of Code project in 2010
- Passed Boost mini review early this year
- Will be part of Boost starting with V1.47, however it is in SVN already

What's Phoenix? Why a new Version?

- Phoenix V2 was developed as a supporting library for Spirit
  - Features similar to Boost.Bind and Boost.Lambda
  - Hand-rolled Expression Templates (ET)
- Boost review in 2008
  - Unification of functional libraries with minimal disruption for users
  - Use Boost.Proto for unified placeholders and cross library integration
  - Use of C++11 features (rvalues, variadics)
- Improve and document extension mechanism

What's Phoenix? Really?

- Enables functional programming techniques in C++
  - Higher order functions
  - Lambda (unnamed functions)
  - Currying (partial function application)
- C++ Embedded Domain Specific Language (EDSL) in C++
- Focus is more on usefulness and practicality than purity, elegance and strict adherence to FP principles

Motivation

- Functional Programming style in the C++ standard library algorithms use function objects as "callbacks"

```
int init = 0;
std::accumulate (
    container.begin(), container.end(), init
  , std::plus<int>()
);
```

- Phoenix is the next evolutionary step

Motivation

- Functional Programming style in the C++ standard library algorithms use function objects as “callbacks”

```
int init = 0;
std::accumulate (
    container.begin(), container.end(), init
  , arg1 + arg2
);
```

- Phoenix is the next evolutionary step

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
- Statements
- Partial function application
- Construct, New, Delete, Casts
- Adapt arbitrary functions
- …
- Everything is a function object!

Phoenix as C++ in C++

- Values and References

```
cout << val(42)();                          // 42

int i = 42;
cout << ref(i)();                           // 42
```

Phoenix Function Objects

- Pseudo code for `val()`:

```
template <typename T>
struct val_impl {
    T value;
    val_impl(T t) : value(t) {}
    T operator()(…) const { return value; }
};

template <typename T> val_impl<T> val(T t)
{
    return val_impl<T>(t);
}
```

Phoenix Function Objects

- Pseudo code for `ref()`:

```
template <typename T>
struct ref_impl {
    T& value;
    ref_impl(T& t) : value(t) {}
    T& operator()(…) const { return value; }
};

template <typename T> ref_impl<T> ref(T& t)
{
    return ref_impl<T>(t);
}
```

Phoenix as C++ in C++

- Values and References
- Arguments

```
cout << arg1(42);                          // 42

const char* s = "Hello World!";
cout << arg1(s);                           // Hello World!
```

  - Alternative placeholder names available

```
cout << _1(i);                             // 42
cout << _1(s);                             // Hello World!
```

Phoenix Function Objects

- Pseudo code for `arg1`:

```
struct arg1_impl {
    template <typename T1, …>
    T1 operator()(T1 t1, …) const { return t1; }
};

arg1_impl const arg1 = arg1_impl();
```

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators

```
cout << (arg1 * arg2)(2, 3);          // 6

int x = 3, z = 5;
(ref(x) = arg1 + ref(z))(4);
cout << x;                            // 9

(arg1 = arg2 + (3 * arg3))(ref(x), 3, 4)
cout << x;                            // 15
```

Phoenix Function Objects

- Pseudo code for +:

```
template <typename F1, typename F2> struct plus_impl {
    F1 f1; F2 f2;
    plus_impl(F1, f1, F2 f2) : f1(f1), f2(f2) {}
    template <typename T1, …>
    T1 operator()(T1 t1, …) const
    { return f1(t1, …) + f2(t1, …); }
};

template <typename F1, typename F2>
plus_impl<F1, F2> operator+(F1 f1, F2 f2)
{
    return plus_impl<F1, F2>(f1, f2);
}
```

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
  - Unary prefix: `~, !, -, +, ++, --,` & (reference), * (dereference)
  - Unary postfix: `++, --`
  - Binary: `=, [], +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= +, -, *, /, %, &, |, ^, <<, >> ==, !=, <, >, <=, >= &&, ||, ->*`
  - Ternary: `if_else(c, a, b)`

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
- Statements

```
cout << (if_(arg1 > 5)[std::cout << arg1])(6);  // 6

vector<int> v = { 4, 5, 6, 7 };
std::for_each(v.begin(), v.end(),
    if_(arg1 > 5)[
        cout << arg1 << ", "
    ]);                                          // 6, 7
```

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
- Statements
- Partial function application

```
void foo (int x, int y)
{
    std::cout << x+y << std::endl;
}
int i = 4;
bind(&foo, arg1, 3)(i);            // 7
```

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
- Statements
- Partial function application
- Construct, New, Delete, Casts

```
construct<std::string>(arg1, arg2)
new_<std::string>(arg1, arg2)
delete_(arg1)
static_cast_<int*>(arg1)
```

Phoenix as C++ in C++

- Values and References
- Arguments
- Operators
- Statements
- Partial function application
- Construct, New, Delete, Casts
- Adapt arbitrary functions
  - Factorial

```
cout << fact(4)();                                    // 24

vector<double> v = { 1, 2, 3, 4 };
for_each(v.begin(), v.end(), cout << fact(_1)); // 1 2 6 24
```

Phoenix Function Function Objects

- Factorial code:

```
struct fact_impl {
    template <typename Sig>
    struct result;

    template <typename This, typename Arg>
    struct result<This(Arg const &)>
    { typedef Arg type; }

    template <typename Arg>
    Arg operator()(Arg const & n) const
    { return (n <= 1) ? 1 : n * (*this)(n-1); }
};
function<fact_impl> const fact = fact_impl();
```

Phoenix vs. C++11 Lambdas

C++11 Lambdas

- ✓ Built-in language feature
- ✓ No significant compile time hit

- Constructs monomorphic function objects

- ❖ Expressions need to be wrapped into lambda syntax

Phoenix

- ✓ Library
- ✓ Expression are placed directly into code
- ✓ Constructs polymorphic function objects
- ✓ Constructs variadic function objects

- ❖ Significant compile time hit

More Examples

More Examples

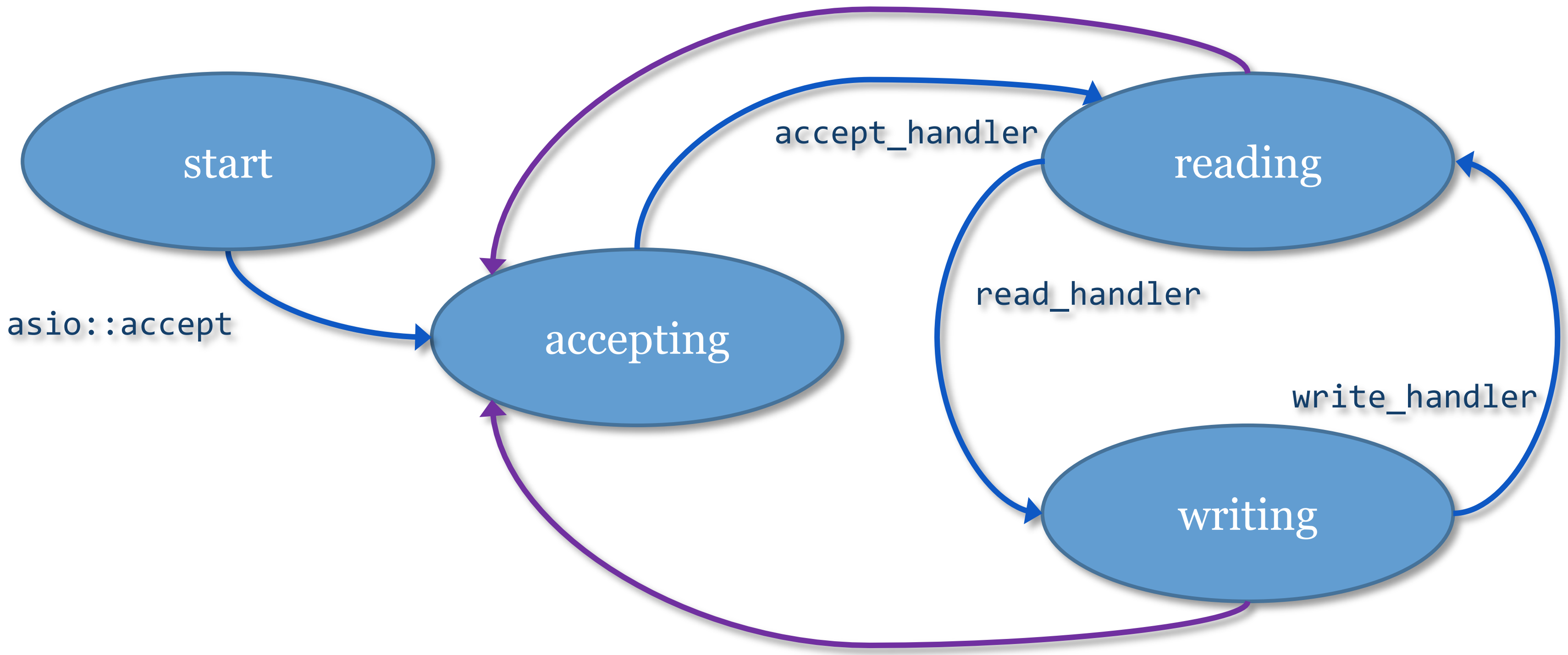
```
// simple function object invocation
template <typename F>
void print(F f) { std::cout << f() << std::endl; }

print(val(3));                              // 3
print(val("Hello World"));                  // Hello World


// find the first odd number in a vector
std::vector<int> c = { 2, 10, 4, 5, 1, 6, 8, 3, 9, 7 };
auto it = std::find_if(c.begin(), c.end(), arg1 % 2 == 1);
if (it != c.end()) cout << *it << std::endl;
```

Full Example

Minimal Asio Echo Server

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
  - Writing a simple asynchronous echo server



Full Example: Asio Echo Server

- Adapt existing code, for example ASIO

read:

```
BOOST_PHOENIX_ADAPT_FUNCTION(void, read, asio::async_read, 3)
```

write:

```
BOOST_PHOENIX_ADAPT_FUNCTION(void, write, asio::async_write, 3)
```

buffer:

```
BOOST_PHOENIX_ADAPT_FUNCTION(
    asio::mutable_buffers_1, buffer, asio::buffer, 2)
```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO

accept:

```
template <typename Acceptor, typename Socket, typename Handler>
void accept_impl(Acceptor& acceptor, Socket& socket,
    Handler const& handler)
{
    acceptor.async_accept(socket, handler);
}

BOOST_PHOENIX_ADAPT_FUNCTION(void, accept, accept_impl, 3)
```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
- Using these functions we construct handlers:
  - read_handler

    ```
    boost::function<void(system::error_code&, size_t)>
    ```
  - write_handler

    ```
    boost::function<void(system::error_code&, size_t)>
    ```
  - accept_handler

    ```
    boost::function<void(system::error_code&)>
    ```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
  - accept_handler, will be called by ASIO when connection was accepted (or refused)

```
boost::function<void(system::error_code&)>
   accept_handler =
      if_(!_1)[
          read(ref(socket), buffer(ref(buf), max_length),
               ref(read_handler))
      ]
      .else [
          bind(&asio::ip::tcp::socket::close, ref(socket)),
          accept(ref(acceptor), ref(socket), ref(accept_handler))
      ];
```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
  - read_handler: will be called by ASIO after read_async

```
boost::function<void(system::error_code&, size_t)>
   read_handler =
      if_(!_1)[
         write(ref(socket), buffer(ref(buf), _2),
               ref(write_handler)
      ]
      .else [
         bind(&asio::ip::tcp::socket::close, ref(socket)),
         accept(ref(acceptor), ref(socket), ref(accept_handler))
      ];
```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
  - write_handler: will be called by ASIO after write_async

```
boost::function<void(system::error_code&, size_t)>
   write_handler =
      if_(!_1)[
         read(ref(socket), buffer(ref(buf), max_length),
              ref(read_handler)
      ]
      .else [
         bind(&asio::ip::tcp::socket::close, ref(socket)),
         accept(ref(acceptor), ref(socket), ref(accept_handler))
      ];
```

Full Example: Asio Echo Server

- Adapt existing code, for example ASIO
  - Start server

```
asio::io_service io_service;
asio::ip::tcp::acceptor acceptor(io_service,
     asio::ip::tcp::endpoint("localhost:1234"));
asio::ip::tcp::socket socket(io_service);

acceptor.async_accept(socket, accept_handler);
```

Boost Proto

The workhorse behind Phoenix V3

Boost Proto: The Phoenix' Workhorse

- The facilities of proto are used to form the backend of phoenix
  - Creation of expression template (ET) classes and composition
  - Formulation of transformations on the created ET tree

Boost Proto: The Phoenix' Workhorse

- Provides facilities to generate your abstract syntax tree (AST)
  - Representation of your expression in terms of proto::expr<>, a hierarchical type holding terms of the expression by reference
- Describe your EDSL in terms of grammar rules
  - Not all expressions are valid
  - These grammars describe valid expressions
- Based on these rules, perform actions on the AST
  - Every rule is associated with an action, which is executed for the node the rule matched

Boost Proto: Phoenix' Workhorse

- The type generated from a Proto expression is an instantiation of the Proto expression class

```
_1 + _2

template <typename Lhs, typename Rhs>
proto::expr<proto::tag::plus, Lhs, Rhs>
operator+ (Lhs const & lhs, Rhs const & rhs)
{
    return proto::make_expr<proto::tag::plus>(lhs, rhs);
}
```

SIMPLIFIED!

Boost Proto: Phoenix' Workhorse

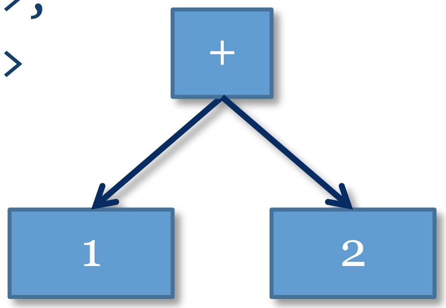
- This type can be interpreted as a tree. This tree is what we call the AST for our language.

```
1 + 2 → proto::make_expr<proto::tag::plus>(1, 2)
```

Boost Proto: Phoenix' Workhorse

- This type can be interpreted as a tree. This tree is what we call the AST for our language.

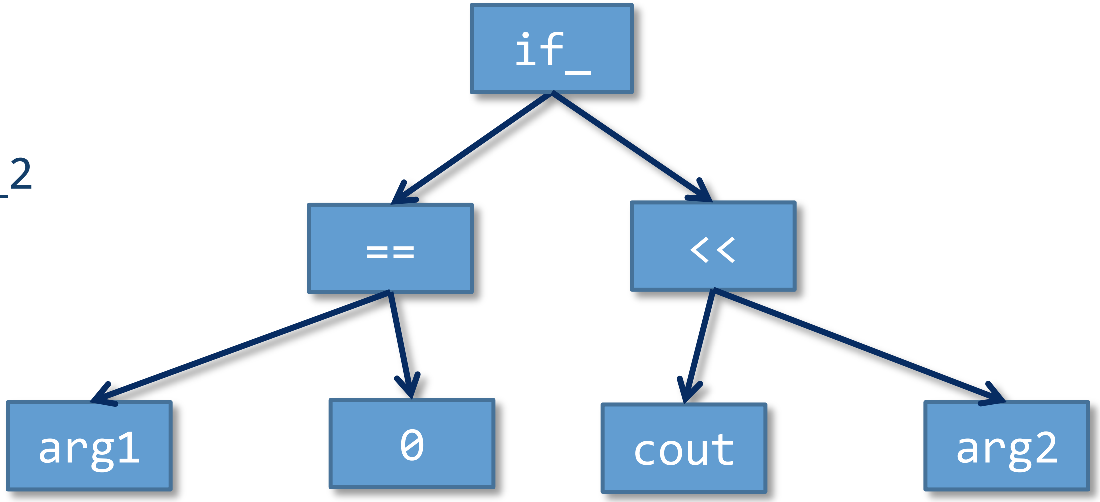
```
val(1) + 2 → proto::make_expr<proto::tag::plus>(1, 2)

proto::expr<
    proto::tag::plus, proto::list2<
        proto::expr<proto::tag::terminal, int>,
        proto::expr<proto::tag::terminal, int>
    > >

plus(terminal(1), terminal(2))
```



The Phoenix AST in Proto's world

- Every Phoenix construct can be seen as an AST node of our Phoenix EDSL
- By composing these we create a bigger AST

```
if_(_1 == 0)
[
      std::cout << _2
]
```



The Phoenix AST in Proto's world

- By using Proto we are able to introspect and transform this AST in any way we like
- The (lazy) evaluation inside Phoenix can be seen as a transformation of this AST
- Default (predefined) evaluation in Phoenix corresponds to 'normal' operator semantics
- By defining your own nodes you can extend Phoenix in any way you wish
- By customizing the evaluation of certain (predefined) nodes you can change the overall scheme
- By transforming the tree before evaluation you can do additional tricks

Phoenix’ Extension Mechanism

Add New Constructs: Define your own Node

Phoenix' Extension Mechanism

Add new constructs

- Let's start simple. Assume you have the following code:

```
#pragma omp parallel for
for (int i = 0; i < NUM; ++i)
   c[i] = a[i] + b[i];
```

- And you want to express exactly this with Phoenix. How would you do that?

Phoenix' Extension Mechanism

Add new constructs

- Anticipated syntax:

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```

- Phoenix expression takes 4 arguments
  - Each is a Phoenix expression on its own

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```


Phoenix’ Extension Mechanism

Add new constructs

- Step 1: A function object:

```
namespace omp
{
    struct for_eval
    {
        typedef void result_type;
        template <typename I, typename C, typename S, typename D,
                  typname Ctx>
        void operator()(I init, C cond, S step, D do_, Ctx ctx) const
        {
            #pragma omp parallel for
            for(eval(init, ctx); eval(cond, ctx); eval(step, ctx))
                eval(do_, ctx);
        }
    };
}
```

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```

Phoenix’ Extension Mechanism

Add new constructs

- Step 2: Create the Phoenix expression type:

```
BOOST_PHOENIX_DEFINE_EXPRESSION(
    (omp)(for_)                          // define omp::for_
  , (boost::phoenix::meta_grammar)       // Cond
    (boost::phoenix::meta_grammar)       // Init
    (boost::phoenix::meta_grammar)       // Step
    (boost::phoenix::meta_grammar)       // Do
)
```

Four Arguments

- Defines:

```
omp::make_for_, omp::result_of::make_for_, omp::expression::for_
omp::rule::for_, omp::tag::for_, omp::functional::make_for_
```

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```


Phoenix’ Extension Mechanism

Add new constructs

- Step 3: Create expression generator: `omp::for_`

```
namespace omp {
    template <typename Init, typename Cond, typename Step>
    struct for_gen {…};

    template <typename Init, typename Cond, typename Step>
    inline for_gen<Init, Cond, Step> const
    for_(Init const& init, Cond const& cond, Step const& step)
    {
        return for_gen<Init, Cond, Step>(init, cond, step);
    }
}
```

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```

Phoenix’ Extension Mechanism

Add new constructs

- Step 3: Create expression generator

```
template <typename Init, typename Cond, typename Step>
struct for_gen {
    for_gen(Init const& init, Cond const& cond, Step const& step)
      : init(init), cond(cond), step(step) {}

    template <typename Do>
    typename omp::result_of::make_for_<Init, Cond, Step, Do>::type const
    operator[](Do const& do_) const
    {
        return omp::make_for_(init, cond, step, do_);
    }

    Init init; Cond cond; Step step;
};
```

Phoenix “Magic”

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```


Phoenix’ Extension Mechanism

Add new constructs

- Step 4: Define how to evaluate the new expression

```
namespace boost { namespace phoenix
{
    template <>
    struct default_actions::when<omp::rule::for_>
      : phoenix::call<omp::for_eval>
    {};
}}
```

Use `...::when` to associate a grammar rule with an action

```
omp::for_(<init>, <cond>, <reinit>)
[
    <parallel work>
]
```

Phoenix’ Extension Mechanism

Add new constructs

- Step 5: Use it

```
std::vector<int> a(NUM, 1), b(NUM, 2), c(NUM, 0);
(
    let(_a = begin(_1), _b = begin(_2), _c = begin(_3))
    [
        omp::for_(nothing, _a != end(_1), (++_a, ++_b, ++_c))
        [
            *_c = *_a + *_b            // executed in parallel!
        ]
      , std::cout << accumulate(_3, 0) << "\n"
    ]
)
(a, b, c);
```

Phoenix' Extension Mechanism

Customizing Existing Constructs: Changing the Evaluation Scheme

Phoenix' Extension Mechanism

Reusing for_ - Changing the evaluation scheme

- Anticipated syntax:

```
omp::parallel(
    <parallel work>
    for_(<init>, <cond>, <reinit>) [ <body> ]
)
```

- Any Phoenix expression inside omp::parallel() will be (optionally) parallelized
  - We show how to parallelize for_

```
omp::parallel(
    <parallel work>
    for_(<init>, <cond>, <reinit>) [ <body> ]
)
```

Phoenix’ Extension Mechanism

Reusing for_ - Changing the evaluation scheme

- Step 1: Define how to evaluate the new expression

```
// Define new a new action
namespace omp {
    struct parallel_actions
    {
        template <typename Rule>
        struct when : phoenix::default_actions::when<Rule>
        {};
    };

    // only change what we are interested in:
    template <>
    struct parallel_actions::when<phoenix::rule::for_>
      : phoenix::call<omp::for_eval>
    {};
}
```

```
omp::parallel(
    <parallel work>
    for_(<init>, <cond>, <reinit>) [ <body> ]
)
```


Phoenix’ Extension Mechanism

Reusing for_ - Changing the evaluation scheme

- Step 2: Change the evaluation scheme on the fly

```
// omp::make_parallel, omp::rule::parallel, etc.
BOOST_PHOENIX_DEFINE_EXPRESSION(
    (omp)(parallel), (phoenix::meta_grammar))

namespace boost { namespace phoenix
{
    template <>
    struct default_actions::when<omp::rule::parallel>
      : proto::call<phoenix::evaluator(
                proto::_child0,
                phoenix::functional::context(
                    phoenix::_env, omp::parallel_actions()),
                phoenix::unused)>
    {};
}}
```

One Argument

```
omp::parallel(
    <parallel work>
    for_(<init>, <cond>, <reinit>) [ <body> ]
)
```

Phoenix’ Extension Mechanism

Reusing for_ - Changing the evaluation scheme

- Step 3: Create expression generator

```
namespace omp
{
    template <typename Expr>
    typename omp::result_of::make_parallel<Expr>::type const
    parallel(Expr const& expr)
    {
        return omp::make_parallel(expr);
    }
}
```

One Argument

```
parallel(
    <parallel work>
    for_(<init>, <cond>, <reinit>) [ <body> ]
)
```


Phoenix’ Extension Mechanism

Reusing for_ - Changing the evaluation scheme

- Step 4: Use it

```
std::vector<int> a(NUM, 1), b(NUM, 2), c(NUM, 0);
(
    let(_a = begin(_1), _b = begin(_2), _c = begin(_3))
    [
        omp::parallel(    // now being executed in parallel!
            for_(nothing, _a != end(_1), (++_a, ++_b, ++_c))
            [
                *_c = *_a + *_b
            ]
        )
      , std::cout << accumulate(_3, 0) << "\n"
    ]
)(a, b, c);
```

Phoenix’ Extension Mechanism

Code as Data

Code as Data

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

Greenspun's Tenth Rule of Programming

Code as Data

- We now have both,
  - the possibility of implementing custom expressions and
  - the ability to transform these expressions,
- We gained a powerful implementation of scheme like AST macro capabilities at compile time

Code as Data

Example – Using proto to transform the AST

- Inverting arithmetic expressions

```
struct invert_actions
{
    // By default, just return the expression itself
    template <typename Rule>
    struct when : proto::_
    {};
};
```

Code as Data

Example – Using proto to transform the AST

- Inverting arithmetic expressions

```
// When there is a plus, make a minus out of it
template <>
struct invert_actions::when<phoenix::rule::plus>
  : proto::call<
        phoenix::functional::make_minus(
            phoenix::evaluator(
                proto::_left, phoenix::_context, phoenix::unused),
            phoenix::evaluator(
                proto::_right, phoenix::_context, phoenix::unused)
        >
{};
```

Code as Data

Example – Using proto to transform the AST

- Inverting arithmetic expressions

```
// When there is a minus, make a plus out of it
template <>
struct invert_actions::when<phoenix::rule::minus>
  : proto::call<
        phoenix::functional::make_plus(
            phoenix::evaluator(
                proto::_left, phoenix::_context, phoenix::unused),
            phoenix::evaluator(
                proto::_right, phoenix::_context, phoenix::unused)
        )
    >
{};
```

Code as Data

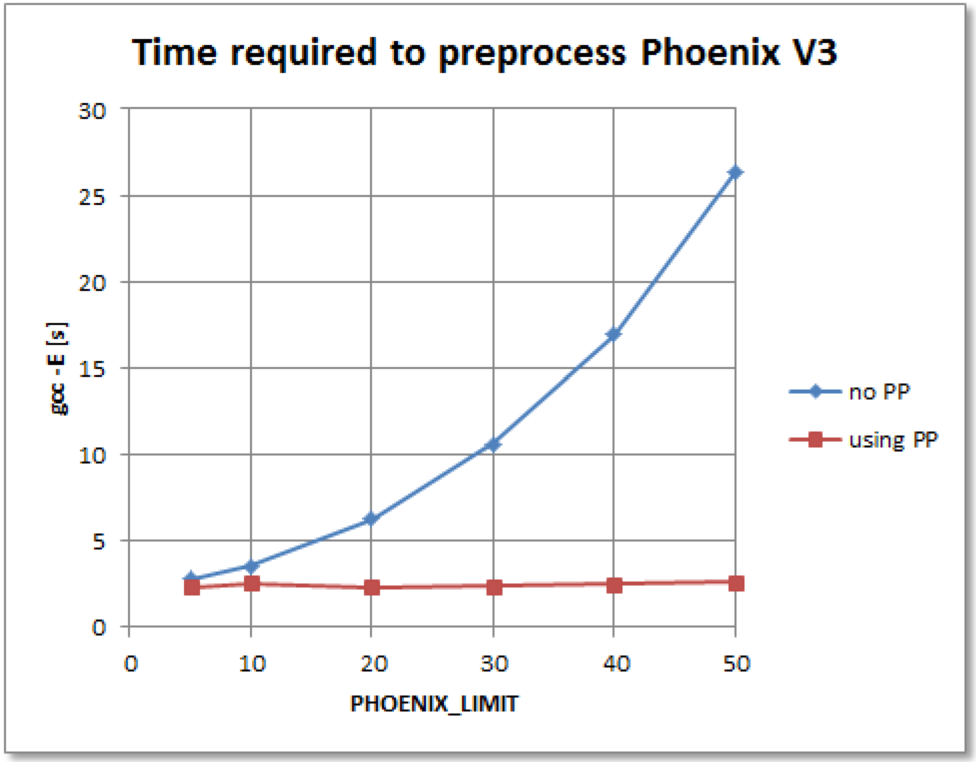
Example – Using proto to transform the AST

- Creating the generator

```
template <typename Expr>
typename phoenix::result_of::eval<
    Expr const&,
    phoenix::result_of::make_context<
        phoenix::make_env<>, invert_actions>::type
>::type
invert(Expr const& expr)
{
    return phoenix::eval(
        expr,
        phoenix::make_context(phoenix::make_env(), invert_actions())
    );
}

(_1 * invert(_2 - _3))(2, 3, 4);                    // 14
```

Phoenix Compile Times

Phoenix Compile Times



- Effect of partial preprocessing of Phoenix Headers on compile times
  - No PP: without partially preprocessed headers
  - Using PP: when using partially preprocessed headers

Phoenix Compile Times

- T1: bind_member_function_tests.cpp

PHOENIX_LIMIT = 10	**T1 (gcc)**	**T1 (VC2010)**
Phoenix V2	2.9 [s]	2.58 [s]
Phoenix V3, no PP	3.4 [s]	4.05 [s]
Phoenix V3, no PP, Proto PP	2.6 [s]	3.43 [s]
Phoenix V3, full PP	2.3 [s]	2.96 [s]

Conclusions

Imagine the unimaginable

- We modeled C++ inside C++
- With the help of Proto, we created a powerful compiler toolkit
- Enabling the creation of new technologies:
  - Multi stage programming, completely done in C++
  - Optimize code based on the high level information of the AST
  - Change the evaluation of a Phoenix expression to whatever you like