

Parasol

Smarter computing.
Texas A&M University

Mach7: The Design and Evolution of a Pattern Matching Library for C++

Parasol

Yuriy Solodkyy • Gabriel Dos Reis • Bjarne Stroustrup

Microsoft *Microsoft* *Morgan Stanley*

Developed at Texas A&M University

May 14, 2014

C++ Now 2014, Aspen, CO

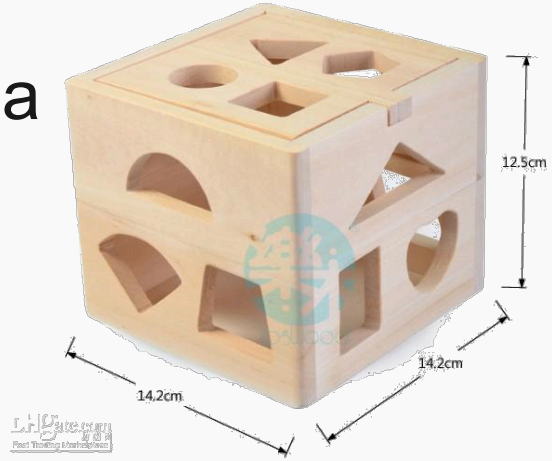
Partially supported by NSF grants:
CCF-0702765, CCF-1043084, CCF-1150055
Open-sourced under BSD License

http://parasol.tamu.edu/mach7/
https://github.com/solodon4/SELL

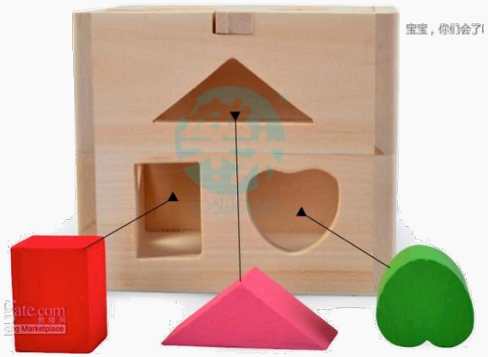
Introduction

Parasol

- What is pattern?
  - Pattern – some breve notation for a structure of interest
    - Implied argument – the subject
    - Composition
  - Subject – an entity with expected structure



- What is pattern matching?
  - Efficient decision procedure determining which of the given set of patterns match a given subject



Examples of Patterns / Subjects

Parasol

- File mask / file
  - `*.?pp`
- Regular expression / string
  - `[A-Za-z_][A-Za-z_0-9]+`
- Grammar / program
  - `exp : exp + exp | exp - exp`
- Mathematical notation / value
  - $a+bi$

Examples of Patterns / Subjects

Parasol

- Bitmask / value
  - (value & (A|B|C)) && !(value & (D|E|F))
- Function template / arguments
  - template <class T> bool foo(list<T,allocator<T>>)
- Exception handler / exception
  - try {…} catch(Car&) {} catch(Vehicle&) {} catch(…) {}
- XPath / XML elements
  - book[/bookstore/@specialty=@style]
- Schedules / time
  - Mon-Fri: 9am-8pm; Sat: 11am-7pm; Sun: Closed

Pattern Matching vs. Pattern Recognition

Parasol

Pattern Matching

The act of checking & decomposition of a perceived structure of a value.

PL-term

- Programing Languages

Analysis in PL

- Exact matching
- Pre-existing patterns
- Efficiency of pattern matching
- 1 subject, N patterns

  find first/all/best pattern that matches the subject

Pattern Recognition

The assignment of a label to a given input value

ML-term

- Machine Learning

Synthesis in ML

- Most-likely matching
- Learnt patterns
- Accuracy of pattern recognition
- N data points, 1 pattern

  find pattern that describes all/ most/some of the data

A pattern obtained via Machine Learning can be encoded in a program with pattern matching constructs to perform efficient pattern recognition

Pattern Matching Semantics

Parasol

- First fit
  - most functional languages, exceptions in C++
- Best fit
  - overload resolution in C++
- Exact fit
  - C-like switch statement
- All/Any fit
  - Languages based on guarded commands etc.

Patterns in Other Languages

Parasol

- Mathematica

```
fib[0|1]:=1
fib[n_] := fib[n-1] + fib[n-2]
```

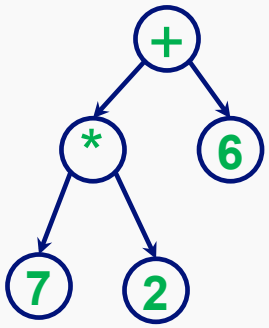
- Haskell

```
take 0     _      = []
take _     []     = []
take (n+1) (x:xs) = x : take n xs
```

- Erlang

```
f("prefix" ++ Str) -> ...
<<D:16,E,F>> = <<1,17,42:16>>
{_N,_N} = {1,2}
```

Example: Case Analysis in FP



Parasol

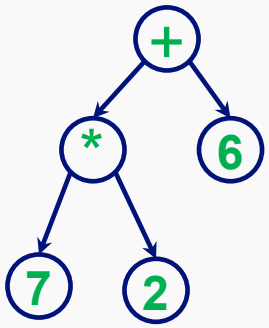
exp ::= *val* | *exp* + *exp* | *exp* - *exp* | *exp* * *exp* | *exp* / *exp*

```
type expr =
    Value  of int
  | Plus   of expr * expr
  | Minus  of expr * expr
  | ...
  ;;

let rec eval e =
 match e with
  Value  v  -> v
| Plus(a,b) -> (eval a)+(eval b)
| Minus(a,b)-> (eval a)-(eval b)
| ...
;;
```

- Pros
  - Elegant
  - Intuitive
  - Extensibility of functions
  - Checked
    - redundancy
    - exhaustiveness
  - Local reasoning
  - Relational
- Cons
  - No extensibility of data
  - Not suitable for classes:
    - Variants are closed & disjoint
    - Classes are extensible & hierarchical

Example: Nested Matching

Parasol

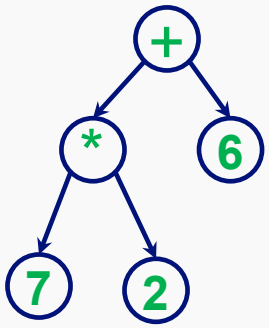


$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp / exp$$

$$xa + xb \rightarrow x(a+b)$$

```
let collect e =
 match e with
  | Plus(Times(x1,a), Times(x2,b)) when x1=x2
       -> Times(x1,Plus(a,b))
  | Plus(Times(a,x1), Times(b,x2)) when x1=x2
       -> Times(Plus(a,b),x1)
  |  e -> e
 ;;
```

Example: Relational Matching



Parasol

exp ::= *val* | *exp* + *exp* | *exp* - *exp* | *exp* * *exp* | *exp* / *exp*

```
let rec equal e1 e2 =
  match (e1,e2) with
    | (Value(v1)     , Value(v2)     ) -> v1 == v2
    | (Plus  (a1, b1), Plus  (a2, b2))
    | (Minus (a1, b1), Minus (a2, b2))
    | (Times (a1, b1), Times (a2, b2))
    | (Divide(a1, b1), Divide(a2, b2))
        -> (equal a1 a2) && (equal b1 b2)
    | _ -> false
   ;;
```

Object-Oriented Alternatives?

Parasol

- Case Analysis
  - Virtual functions
    - Intrusive, control inversion, non-local reasoning
  - Visitor Design Pattern
    - Intrusive, control inversion, hard to teach
- Nested Matching
  - Nested if-statements
    - Verbose
- Relational Matching
  - Double dispatch
    - Intrusive, control inversion, hard to teach

Why Pattern Matching?

Parasol

```
type color   = R | B
type 'a tree = E | T of color * 'a tree * 'a * 'a tree

(**color * 'a tree * 'a * 'a tree->'a tree*)
let balance = function
  | B, T(R, T(R,a,x,b), y, c), z, d
  | B, T(R, a, x, T(R,b,y,c)), z, d
  | B, a, x, T(R, T(R,b,y,c), z, d)
  | B, a, x, T(R, b, y, T(R,c,z,d))
                -> T(R, T(B,a,x,b), y, T(B,c,z,d))
  | col, a, x, b -> T(col, a, x, b)

(** val insert : 'a -> 'a tree -> 'a tree *)
let insert x s =
  let rec ins = function
    | E                 -> T(R,E,x,E)
    | T(col,a,y,b) as s ->
      if x < y then balance(col, ins a, y, b) else
      if x > y then balance(col, a, y, ins b) else s
  in let T(_,a,y,b) = ins s in T(B,a,y,b)
```

```
void rotate_left(rbtree t, node n) {
    node r = n->right;
    replace_node(t, n, r);
    n->right = r->left;
    if (r->left != NULL) {
        r->left->parent = n;
    }
    r->left = n;
    n->parent = r;
}

void rotate_right(rbtree t, node n) {
    node L = n->left;
    replace_node(t, n, L);
    n->left = L->right;
    if (L->right != NULL) {
        L->right->parent = n;
    }
    L->right = n;
    n->parent = L;
}
void replace_node(rbtree t, node oldn, node newn) {
    if (oldn->parent == NULL) {
        t->root = newn;
    } else {
        if (oldn == oldn->parent->left)
            oldn->parent->left = newn;
        else
            oldn->parent->right = newn;
    }
    if (newn != NULL) {
        newn->parent = oldn->parent;
    }
}
void rbtree_insert(rbtree t, void* key, void* value, compare_func compare) {
    node inserted_node = new_node(key, value, RED, NULL, NULL);
    if (t->root == NULL) {
        t->root = inserted_node;
    } else {
        node n = t->root;
        while (1) {
            int comp_result = compare(key, n->key);
            if (comp_result == 0) {
                n->value = value;
                /* inserted_node isn't going to be used, don't leak it */
                free (inserted_node);
                return;
            } else if (comp_result < 0) {
                if (n->left == NULL) {
                    n->left = inserted_node;
                    break;
                } else {
                    n = n->left;
                }
            } else {
                assert (comp_result > 0);
                if (n->right == NULL) {
                    n->right = inserted_node;
                    break;
                } else {
                    n = n->right;
                }
            }
        }
        inserted_node->parent = n;
    }
    insert_case1(t, inserted_node);
    verify_properties(t);
}
void insert_case1(rbtree t, node n) {
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(t, n);
}
void insert_case2(rbtree t, node n) {
    if (node_color(n->parent) == BLACK)
        return; /* Tree is still valid */
    else
        insert_case3(t, n);
}
void insert_case3(rbtree t, node n) {
    if (node_color(uncle(n)) == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(t, grandparent(n));
    } else {
        insert_case4(t, n);
    }
}
void insert_case4(rbtree t, node n) {
    if (n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(t, n->parent);
        n = n->left;
    } else if (n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(t, n->parent);
        n = n->right;
    }
    insert_case5(t, n);
}
void insert case5(rbtree t, node n) {
    }
}
```

Functional Solution to Red-Black Tree Insertion vs. *Imperative*

Design Ideals & Criteria for a Feature

Parasol

Design Ideals

- Simple/Intuitive
- Easy to teach
- Direct show of intent
- Checked
- Non-intrusive
- Open with respect to:
  - Patterns
  - Subjects

Design Criteria

- Work for C++ object model
  - Built-in & user-defined types
  - Multiple inheritance
  - Dynamic linking
- Comparable or faster than workarounds
  - Visitor Design Pattern
  - Closed switch
  - Patterns as Objects

Goals of the *Mach7* library

Parasol

- Turn around ideas quickly
  - Syntax, semantics, type checking, implementation, etc.
  - Feature implementation requires much more effort
  - Force ourselves to remain within an open setting
- Gain experience
  - What other language features would help?
  - What can be generalized?
  - Performance bottlenecks
  - Write/rewrite a couple of realistic applications
- Receive feedback from the users
  - What is intuitive and what is confusing
  - How users typically use the library
- Ignore the hacks
  - We don't like macros either, but they give us a good enough approximation
  - The syntax available in the library is not the proposed feature syntax!

Mach7: Structural Decomposition

Parasol

Bindings

```
template <> class bindings<Value>
 { Members(Value::value); };

template <> class bindings<Plus>
 { Members(Plus::e1,Plus::e2); };

template <> class bindings<Minus>
 { Members(Minus::e1,Minus::e2); };

template <> class bindings<Times>
 { Members(Times::e1,Times::e2); };

template <> class bindings<Divide>
 { Members(Divide::e1,Divide::e2);};
```

C++

```
class Expr  { virtual ~Expr(); };
class Value : Expr { int value; };
class Plus  : Expr { Expr& e1; Expr& e2; };
class Minus : Expr { Expr& e1; Expr& e2; };
class Times : Expr { Expr& e1; Expr& e2; };
class Divide: Expr { Expr& e1; Expr& e2; };

int eval(const Expr* e) {
  var<int> n; var<const Expr*> a,b;
  Match(e)
   Case(C<Value> (n))   return n;
   Case(C<Plus>  (a,b)) return eval(a)+eval(b);
   Case(C<Minus> (a,b)) return eval(a)-eval(b);
   Case(C<Times> (a,b)) return eval(a)*eval(b);
   Case(C<Divide>(a,b)) return eval(a)/eval(b);
  EndMatch
}
```

Mach7: Nested Matching

Parasol

$$xa + xb \rightarrow x(a+b)$$

```
let collect e =

  match e with
    Plus(Times(x1,a), Times(x2,b))
      when x1=x2 -> Times(x1,Plus(a,b))
  | Plus(Times(a,x1), Times(b,x2))
      when x1=x2 -> Times(Plus(a,b),x1)
  |   e -> e
  ;;
```

- Language solution
- Predefined set of patterns
- Implicit variable introduction
- Linearity

```
const Expr* collect(const Expr* e) {
  var<const Expr*> x, a, b;
  Match(e)
   Case(C<Plus>(C<Times>(x,a),C<Times>(+x,b)))
       return new Times(x, new Plus(a,b));
   Case(C<Plus>(C<Times>(a,x),C<Times>(b,+x)))
       return new Times(new Plus(a,b), x);
  EndMatch
  return e;
}
```

- Library solution
- All patterns are user-defined
- Explicit variable introduction
- Equivalence combinator

Mach7: Relational Matching

Parasol

```
let rec equal e1 e2 =

  match (e1,e2) with
    (Value(v1), Value(v2)) -> v1 == v2
  | (Plus  (a1, b1), Plus  (a2, b2))
  | (Minus (a1, b1), Minus (a2, b2))
  | (Times (a1, b1), Times (a2, b2))
  | (Divide(a1, b1), Divide(a2, b2))
      -> (equal a1 a2) && (equal b1 b2)
  | _ -> false
  ;;
```

- Single scrutiny
- Closed world
  - Exhaustiveness check
  - Redundancy check
  - Compiler optimizations

```
bool operator==(const Expr& e1, const Expr& e2) {
  var<int> n; var<const Expr&> a, b;
  Match(e1              , e2               )
   Case(C<Value> (n)    , C<Value> (+n)    ) return true;
   Case(C<Plus>  (a, b), C<Plus>  (+a, +b)) return true;
   Case(C<Minus> (a, b), C<Minus> (+a, +b)) return true;
   Case(C<Times> (a, b), C<Times> (+a, +b)) return true;
   Case(C<Divide>(a, b), C<Divide>(+a, +b)) return true;
   Otherwise()                              return false;
  EndMatch
}
```

- Multiple scrutiny
- Open world
  - Open to new patterns & combinators
  - Open to new classes

Mach7: Pattern Combinators

Parasol

Operators for:

- creating new patterns
- modifying existing ones
- combining patterns and lazy expressions

Guard: P |= E
- x |= $x > 7$;
- x |= $x == y$

Equivalence: +E
- +x;
- +$(2*x)$

Logical: $P1$ && $P2$; $P1$ || $P2$; !P
- $2*x$ && $3*y$
- $+x$ || $+y$
- !$+x$

Location: &P ; *P
- x |= x != *nullptr* && $P(*x)$
- y |= $P(\&y)$

Quantifiers: all(P), exist(P)
- *all*(x |= x % 2)
- *exist*(!+x)

Mach7: Balancing Red-Black Tree

Parasol

```
class T{ enum color{black,red} col; T* left; K key; T* right; };

T* balance(T::color clr, T* l, const K& key, T* r)
{
  const T::color B = T::black, R = T::red;
  var<T*> a, b, c, d; var<K&> x, y, z; T::color col;
  Match(clr, l, key, r)
  {
   Case(B, C<T>(R, C<T>(R, a, x, b), y, c), z, d) return ...;
   Case(B, C<T>(R, a, x, C<T>(R, b, y, c)), z, d) return ...;
   Case(B, a, x, C<T>(R, C<T>(R, b, y, c), z, d)) return ...;
   Case(B, a, x, C<T>(R, b, y, C<T>(R, c, z, d))) return ...;
   Case(col, a, x, b) return new T{col, a, x, b};
  }
  EndMatch
}
```

Where `... ≡ new T{R, new T{B,a,x,b}, y, new T{B,c,z,d}};`

Mach7: Algebraic Decomposition

Parasol

- Constructor Patterns
  - structural induction

    Plus(Times(a,3),b)=r
- $n+k$ Patterns
  - mathematical induction

    $x+3=r$
- Application Patterns
  - equational semantics

    $f(x,y) = r$

- Notational Patterns
  - decomposition of mathematical objects
  - n/m – rational number
  - $3q+r$ – quotient and remainder
  - $a+bi$ – complex number
  - $2D$ line:

    $mX+c$ slope-intercept form

    $aX+bY=c$ linear equation form

    $(Y-y0)/(y1-y0) = (X-x0)/(x1-x0)$ two points form

Mach7: Algebraic Decomposition

Parasol

```
double power(double x, int n)
{
  var<int> m;
  Match(n)
  {
    Case(0)      return 1.0;
    Case(1)      return x;
    Case(2*m)    return   sqr(power(x,m));
    Case(2*m+1) return x*sqr(power(x,m));
  }
  EndMatch
}

size_t gcd(const size_t a, const size_t b)
{
  var<size_t> x;
  Match(a,b)
    Case(_,a)   return a;
    Case(_,a+x) return gcd(a,x);
    Case(b+x,_) return gcd(b,x);
  EndMatch
}
```

- Equational reasoning
  - Since n is of integral type:

$$x^n = \begin{cases} 1 & n = 0 \\ x & n = 1 \\ (x^m)^2 & n = 2m \\ x(x^m)^2 & n = 2m+1 \end{cases}$$

  - ***n=2*m*** $\longrightarrow$ ***n*** must be even and ***m=n/2***
  - ***n=2*m+1*** $\longrightarrow$ ***n*** must be odd and ***m=(n-1)/2***
- Type matters
  - ***b+x=a*** for unsigned ***x*** will match only when ***b > a***

Mach7: Other Patterns

Parasol

```
var<int> n,m,y,d,m;
auto month = m |= m > 0 && m < 13; // Save pattern to variable
auto day   = d |= d > 0 && d < 31; // Day pattern

Match(s)
{
  Case(rex("([0-9]+)-([0-9]+)-([0-9]+)", 979)) // Local phone
  Case(rex("([0-9]+)-([0-9]+)-([0-9]+)",
           any({800,888,877,866,855}), n, m)   // Toll-free phone
  Case(rex("([0-9]{4})-([0-9]{2})-([0-9]{2})",
           y, month, d |= d > 0 && d <= 31))   // Date
  Otherwise() // Something else
}
EndMatch
```

- All Patterns in the library are user-defined
- Patterns can be saved in variables and passed to functions

Patterns-subjects interaction

Parasol

```
match e with
  [0,1,2]          -> Check
| [x]              -> Introduce & bind variables
| [_,y,(0|1)]      -> Compose patterns
| x:y:tail         -> Structurally decompose
| x+1:2*y:tail     -> Algebraically decompose
| x:y:tail when x=y -> Restrict
| _ -> …                Perform actions
;;                      etc…
```

Always nice to have

Parasol

- Type checking/inference
- Openness to new patterns and combinators
- Openness to built-in and user-defined types
- Support of various ADT encodings
- Efficiency in matching
- Redundancy checking
- Exhaustiveness checking
- Extensible Notation
- Implicit Variable introduction

Idea: Patterns as Objects

Parasol

```
struct object { // root of the object class hierarchy
  virtual bool operator==(const object&) const = 0;
};

struct pattern { // pattern interface
  virtual bool match(const object&) = 0;
};
```

Run-time composition of patterns

Pros

- Open patterns
- First-class patterns
- Dynamic composition

Cons

- Intrusive
- Type errors at run-time
- Extremely slow

Patterns as Expression Templates

Parasol

```
template <typename... P>
struct some_pattern : std::tuple<P...>
{
    some_pattern(P&&...);
    template <typename S> bool operator()(const S& s) const;
};
```

Compile-time composition of patterns

Pros

- Open patterns
- First-class patterns
- Non-intrusive
- Type errors at compile-time

Cons

- No run-time composition

Mach7 Concepts

Parasol

Pattern

```
template <typename P>
constexpr bool Pattern() {
  return Copyable<P>
      && is_pattern<P>::value
      && requires (typename S, P p, S s)
         {
           bool = { p(s) };
           AcceptedType<P,S>;
         };
}
```

LazyExpression

```
template <typename E>
constexpr bool LazyExpression() {
  return Copyable<E>
      && is_expression<E>::value
      && requires (E e)
         {
           ResultType<E>;
           ResultType<E> == { eval(e) };
           ResultType<E> { e };
         };
}
```

Primitive Patterns

Parasol

```
/// Wildcard pattern
struct wildcard
{
  template <typename T>
  bool operator()(const T&) const noexcept
  {
    return true;
  }
};

/// Value pattern
template <class T>
struct value
{
  T m_value; // Value to compare to

  explicit value(const T&  t) : m_value(t) {}

  bool operator()(const T& t) const {
    return m_value == t;
  }
};
```

```
/// Variable pattern
template <class T>
struct var
{
  mutable T m_value; // Bound value

  var() : m_value() {}

  /// bind the value and accept match
  bool operator()(const T& t) const {
    m_value = t;
    return true;
  }

  /// What if type of subject is different?
  template <typename U>
  bool operator()(const U& u) const
  {
    m_value = u;
    return m_value == u;
  }
};
```

Structural Decomposition: Bindings

Parasol

User defines bindings by specializing

```
template <typename T> struct bindings;
```

For example, for classes:

```
template <> struct bindings<Value> { Members(Value::value); };
template <> struct bindings<Plus>  { Members(Plus::e1, Plus::e2); };
```

After *Members*-macro substitution

```
template <> struct bindings<Plus> {
    static auto member1() noexcept { return &Plus::e1; }
    static auto member2() noexcept { return &Plus::e2; }
};
```

Allows us to access members by position in bindings:

```
T* t = ...;
auto s2 = apply_member(t, bindings<T>::member2());
// apply_member is overloaded by different kinds of member pointers
```

Constructor Pattern

Parasol

```
template <typename T, typename P1, typename P2>
struct constr2 /// Constructor/Type pattern of 2 arguments
{
    P1 m_p1; ///< Pattern representing 1st operand
    P2 m_p2; ///< Pattern representing 2nd operand

    constr2(const P1& p1, const P2& p2) : m_p1(p1), m_p2(p2) {}

    bool operator()(const T& t) const {
        return m_p1(apply_member(t, bindings<T>::member1())) ///< Apply 1st sub-pattern to 1st argument
            && m_p2(apply_member(t, bindings<T>::member2()));///< Apply 2nd sub-pattern to 2nd argument
    }

    template <typename U>
    bool operator()(const U& u) const {      ///< When subject type is different
        auto t = dynamic_cast<const T*>(&u); ///< Enabled only for polymorphic T and U
        return t && operator()(*t);
    }
};

template <typename T, typename P1, typename P2> ///< Syntactic helper
auto C(P1&& p1, P2&& p2) { return constr2<T, P1, P2>(forward<P1>(p1), forward<P2>(p2)); }
```

Algebraic Decomposition

Parasol

```
/// Expression pattern for binary operation
template <typename F, typename E1, typename E2>
struct expr
{
    static_assert(is_expression<E1>::value, "Argument E1 must be a lazy expression");
    static_assert(is_expression<E2>::value, "Argument E2 must be a lazy expression");

    typedef ...F(E1,E2)... result_type; ///< Type of result when used in expression.

    /// Type function returning type accepted by the pattern for a given subject type S.
    template <typename S> struct accepted_type_for { typedef result_type type; };

    /// Eagerly evaluate in the right-hand side
    operator result_type() const { return eval(*this); }

    template <typename U>
    bool operator()(const U& u) const { return solve(*this,u); }

    E1 m_e1; ///< Expression template with the 1st operand
    E2 m_e2; ///< Expression template with the 2nd operand
};
```

Algebraic Decomposition

Parasol

```
template <LazyExpression E, typename S>
inline bool solve(const E&, const S&) noexcept
{
    static_assert(std::is_same<E,S>::value,"There is no solver for matching E(S)”);
    return false;
}


template <LazyExpression E, typename T>
    requires Field<typename E::result_type>()
bool solve(const mult<E,value<T>>& e, const E::result_type& r)
    { return solve(e.m_e1,r/eval(e.m_e2)); } // e.m_e2 is c


template <LazyExpression E, typename T>
    requires Integral<typename E::result_type>()
bool solve(const mult<E,value<T>>& e, const E::result_type& r) {
    T c = eval(e.m_e2); // e.m_e2 is c
    return r%c == 0 && solve(e.m_e1,r/c);
}
```

What can we do so far?

Parasol

Explicitly declare primitive patterns

```
var<const Expr&> a, b, c, d;
wildcard _;
```

Decompose objects by composing patterns

```
Expr& expr = ...;
if (C<Times>(a, _)(expr)) { ... }
```

Nest patterns:

```
if (C<Plus>(C<Times>(a, _), C<Times>(c, val(0)))(expr)) {..
```

The matching syntax is somewhat lame

```
if (C<Times>(a, b)(expr)) { ... } else
if (C<Divide>(_, val(0))(expr)) { ... } else
if (C<Plus>(C<Times>(a, _), C<Times>(c, val(0)))(expr)) {..
```

With macros and meta-programming

Parasol

Explicitly declare primitive patterns

```
var<const Expr&> a, b, c, d;
wildcard _;
```

Decompose objects by composing patterns

```
Expr& expr = ...;
if (C<Times>(a, _)(expr)) { ... }
```

Nest patterns:

```
if (C<Plus>(C<Times>(a, _), C<Times>(c, 0))(expr)) {...}
```

The matching syntax is somewhat better

```
Match(expr)
  Case(C<Times>(a, b)) ...
  Case(C<Divide>(_, 0)) ...
  Case(C<Plus>(C<Times>(a, _), C<Times>(c, 0))) ...
EndMatch
```

Cool Syntax Bro!

Parasol

- Must be slow
  - Sequential execution
  - Definitely slow when dynamic_cast is involved
  - Instantiates tons of temporaries
  - Re-matches same sub-patterns on same subjects
- Does not check for redundancy
- What would exhaustiveness mean here?
- Does my pattern composition make sense?
- What if I don't use polymorphic classes?

Memoization Device: Check Once!

Parasol

- Hypothetical Statement

  Execute the 1st statement s_i whose predicate P_i is true

```
switch (x) {
    case P1(x): s1;
     ...
    case Pn(x): sn;
}
```

- Memoize

  Clause of the 1st successful predicate

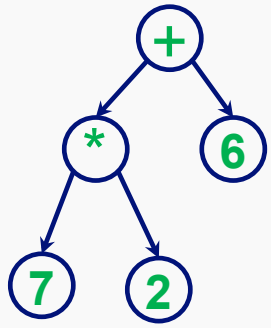
- Assume

  Functional behavior

```
NOTE: The code is generated!

typedef decltype(x) T;
static hash_map<T,size_t> labels;

switch (size_t& l = labels[x])
{
default: // we have not seen x yet
  if (P1(x)) { l = 1; case 1: s1; }
  else
   ...
  if (Pn(x)) { l = n; case n: sn; }
  else         l = n+1;
case n+1: // none is true on x
}
```

Expression Problem

Parasol



$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp / exp$$

Functional Languages

```
type expr =
            Value  of int
          | Plus   of expr * expr
          | Minus  of expr * expr
          | Times  of expr * expr
          | Divide of expr * expr ;;

let rec eval e =
 match e with
  Value  v    -> v
| Plus  (a,b) -> (eval a) + (eval b)
| Minus (a,b) -> (eval a) - (eval b)
| Times (a,b) -> (eval a) * (eval b)
| Divide(a,b) -> (eval a) / (eval
b) ;;
```

Easy to add new functions
Adding new variants is intrusive

Object-Oriented Languages

```
class Expr  { };
class Value : Expr { int value; };
class Plus  : Expr { Expr &e1, &e2; };
class Minus : Expr { Expr &e1, &e2; };
class Times : Expr { Expr &e1, &e2; };
class Divide: Expr { Expr &e1, &e2; };

int Value::eval() { return value; }
int Plus ::eval() { return e1.eval()+e2.eval(); }
int Minus::eval() { return e1.eval()-e2.eval(); }
int Times::eval() { return e1.eval()*e2.eval(); }
int Divide::eval(){ return e1.eval()/e2.eval(); }
```

Easy to add new variants
Adding new functions is intrusive

Functional Style Case Analysis

Parasol

$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp / exp$$

```
type expr =
    Value  of int
  | Plus   of expr * expr
  | Minus  of expr * expr
  | ...
  ;;

let rec eval e =
 match e with
  Value  v  -> v
| Plus(a,b) -> (eval a) + (eval b)
| Minus(a,b)-> (eval a) - (eval b)
| ...
;;
```

- Pros
  - Elegant
  - Intuitive
  - Extensibility of functions
  - Checked
    - redundancy
    - exhaustiveness
  - Local reasoning
  - Relational
- Cons
  - No extensibility of data
  - Not suitable for classes:
    - Variants are closed & disjoint
    - Classes are extensible & hierarchical

Object-Oriented Decomposition

Parasol

exp ::= *val* | *exp* + *exp* | *exp* - *exp* | *exp* * *exp* | *exp* / *exp*

```
class Expr {
  virtual int eval();
};

class Value : Expr {
  int eval() { return value; }
  int value;
};

class Plus : Expr {
  int eval() {
    return e1.eval()+e2.eval();
  }
  Expr& e1;
  Expr& e2;
};
```

- Pros
  - Modularity
  - Encapsulation
  - Extensibility of data
  - Works in the presence of
    - multiple inheritance
    - dynamic linking
- Cons
  - No extensibility of functions
  - No local reasoning
  - Non-relational

ADT in C++: Polymorphic Encoding

Parasol

```
class Expr     { virtual ~Expr() {} };
class Value : Expr { int value; };
class Plus  : Expr { Expr& e1, &e2; };
class Minus : Expr { Expr& e1, &e2; };
class Times : Expr { Expr& e1, &e2; };
class Divide: Expr { Expr& e1, &e2; };

int eval(Expr& e)
{
  if (Value*  p = dynamic_cast<Value* >(&e))
     return p->value;
  if (Plus*   p = dynamic_cast<Plus*  >(&e))
     return eval(p->e1) + eval(p->e2);
  if (Minus*  p = dynamic_cast<Minus* >(&e))
     return eval(p->e1) - eval(p->e2);
  if (Times*  p = dynamic_cast<Times* >(&e))
     return eval(p->e1) * eval(p->e2);
  if (Divide* p = dynamic_cast<Divide*>(&e))
     return eval(p->e1) / eval(p->e2);
  return 0; // assert(0) or throw(something)
}
```

- Pros
  - somewhat intuitive
  - type safe
  - local reasoning
  - independent extensibility of data
  - substitutability
  - non-intrusive
- Cons
  - no checking
  - slow
  - verbose

ADT in C++: Tag Encoding

Parasol

```
class Expr { int m_tag; Expr(int t):m_tag(t){} };
class Value : Expr { enum{tag=1}; int value; };
class Plus  : Expr { enum{tag=2}; Expr*e1,*e2; };
class Minus : Expr { enum{tag=3}; Expr*e1,*e2; };
class Times : Expr { enum{tag=4}; Expr*e1,*e2; };
class Divide: Expr { enum{tag=5}; Expr*e1,*e2; };

int eval(const Expr* e)
{
  switch (e->m_tag)
  {
  case Value::tag:  { Value*  p = static_cast<Value*> (e);
                      return p->value;                    }
  case Plus::tag:   { Plus*   p = static_cast<Plus*>  (e);
                      return eval(p->e1) + eval(p->e2); }
  case Minus::tag:  { Minus*  p = static_cast<Minus*> (e);
                      return eval(p->e1) - eval(p->e2); }
  case Times::tag:  { Times*  p = static_cast<Times*> (e);
                      return eval(p->e1) * eval(p->e2); }
  case Divide::tag: { Divide* p = static_cast<Divide*>(e);
                      return eval(p->e1) / eval(p->e2); }
  }
}
```

- Pros
  - somewhat intuitive
  - fast
  - local reasoning
  - controlled extensibility of data
- Cons
  - not type safe
  - no checking
  - no substitutability
  - verbose
  - intrusive

ADT in C++: Discriminated Union Enc.

Parasol

```
struct Expr
{
  enum {Value,Plus,Minus,Times,Divide} m_tag;
  union {
    struct { int value; };
    struct { Expr* e1; Expr* e2; };
  };
};

int eval(const Expr* e)
{
  switch (e->m_tag)
  {
  case Value: return e->value;
  case Plus:  return eval(e->e1)+eval(e->e2);
  case Minus: return eval(e->e1)-eval(e->e2);
  case Times: return eval(e->e1)*eval(e->e2);
  case Divide:return eval(e->e1)/eval(e->e2);
  }
}
```

- Pros
  - intuitive
  - fast
  - local reasoning
- Cons
  - not type safe
  - no checking
  - not hierarchical
  - no extensibility of data

Visitor Design Pattern

Parasol

```
// Forward declare cases
struct Value; struct Plus; struct Minus; struct Times; struct Divide;
// Cases interface
struct Visitor {
    virtual void caseValue (const Value& ) {}
    virtual void casePlus  (const Plus&  ) {}
    virtual void caseMinus (const Minus& ) {}
    virtual void caseTimes (const Times& ) {}
    virtual void caseDivide(const Divide&) {}
};
// Implementation of variants (omitting data members etc.)
struct Expr           { virtual void accept(Visitor& v) const = 0; };
struct Value  : Expr { void accept(Visitor& v) const { v.caseValue (*this); } … };
struct Plus   : Expr { void accept(Visitor& v) const { v.casePlus  (*this); } … };
struct Minus  : Expr { void accept(Visitor& v) const { v.caseMinus (*this); } … };
struct Times  : Expr { void accept(Visitor& v) const { v.caseTimes (*this); } … };
struct Divide : Expr { void accept(Visitor& v) const { v.caseDivide(*this); } … };
```

Visitor Design Pattern

Parasol

```
int eval(const Expr& e)
{
  struct EvalVisitor : Visitor // Visitor implementing case analysis for evaluation
  {
    int result; // Variable where result of evaluation will be stored
    void caseValue (const Value&  e) { result = e.value; }
    void casePlus  (const Plus&   e) { result = eval(e.e1) + eval(e.e2); }
    void caseMinus (const Minus&  e) { result = eval(e.e1) - eval(e.e2); }
    void caseTimes (const Times&  e) { result = eval(e.e1) * eval(e.e2); }
    void caseDivide(const Divide& e) { result = eval(e.e1) / eval(e.e2); }
  };

  EvalVisitor v;   // Instantiate evaluation visitor
  e.accept(v);     // Pass it to accept method on e to find out case of e
  return v.result; // Return the result of case analysis
}
```

Visitor Design Pattern

Parasol

```
// Forward declaration of known variants
class Value; class Plus; class Minus; class Times; class Divide;

// Visitation interface
class Visitor
{
  virtual void visit(const Value&)  = 0;
  virtual void visit(const Plus&)   = 0;
  virtual void visit(const Minus&)  = 0;
  virtual void visit(const Times&)  = 0;
  virtual void visit(const Divide&) = 0;
};

// Abstract base and known derived classes
class Expr { virtual void accept(Visitor&); };
class Value : Expr { void accept(Visitor& v) { v.visit(*this); } int   value;   };
class Plus  : Expr { void accept(Visitor& v) { v.visit(*this); } Expr& e1, &e2; };
class Minus : Expr { void accept(Visitor& v) { v.visit(*this); } Expr& e1, &e2; };
class Times : Expr { void accept(Visitor& v) { v.visit(*this); } Expr& e1, &e2; };
class Divide: Expr { void accept(Visitor& v) { v.visit(*this); } Expr& e1, &e2; };

// Actual implementation of eval with visitor design pattern
int eval(const Expr& e)
{
    struct EvalVisitor : Visitor
    {
        int result;
        void visit(const Value&  e) { result = e.value; }
        void visit(const Plus&   e) { result = eval(e.e1) + eval(e.e2); }
        void visit(const Minus&  e) { result = eval(e.e1) - eval(e.e2); }
        void visit(const Times&  e) { result = eval(e.e1) * eval(e.e2); }
        void visit(const Divide& e) { result = eval(e.e1) / eval(e.e2); }
    } v;
    e.accept(v);
    return v.result;
}
```

- Pros
  - Extensibility of functions
  - Speed
  - Library solution
- Cons
  - Hard to teach
  - Intrusive
  - Specific to hierarchy
  - Lots of boilerplate code
  - Control inversion
  - Hinders extensibility of classes

Polymorphic Exception Idiom

Parasol

```
class Expr { virtual void raise() = 0; };
class Value : Expr { void raise() { throw(*this); } };
class Plus  : Expr { void raise() { throw(*this); } };
class Minus : Expr { void raise() { throw(*this); } };
class Times : Expr { void raise() { throw(*this); } };
class Divide: Expr { void raise() { throw(*this); } };

int eval(Expr& e)
{
  try { e.raise(); }
  catch(Value & e) { return e.value; }
  catch(Plus  & e) { return eval(e.e1) + eval(e.e2); }
  catch(Minus & e) { return eval(e.e1) - eval(e.e2); }
  catch(Times & e) { return eval(e.e1) * eval(e.e2); }
  catch(Divide& e) { return eval(e.e1) / eval(e.e2); }
}
```

- Pros
  - checked: redundancy
  - type safe
  - local reasoning
  - independent extensibility of data
  - substitutability
- Cons
  - slow
  - intrusive

Closer to common syntax, but …

Parasol

- ADT encodings
  - Polymorphic
  - Tagged
  - Discriminated Union
- Ways of handling them
  - Cascading-if
  - Visitor Design Pattern
  - Polymorphic Exception Idiom
  - Hierarchy Linearization
  - Switch
- Syntactic Form

```
Match(expr)
  Case(P1) ...
  Case(Pn) ...
  Otherwise() ...
EndMatch
```

- Can generate most
  - first-fit or all-fit
  - relies on forwarding
  - unique top-level
  - no multiple inheritance
  - no substitutability

Variations within Syntactic Form

Parasol

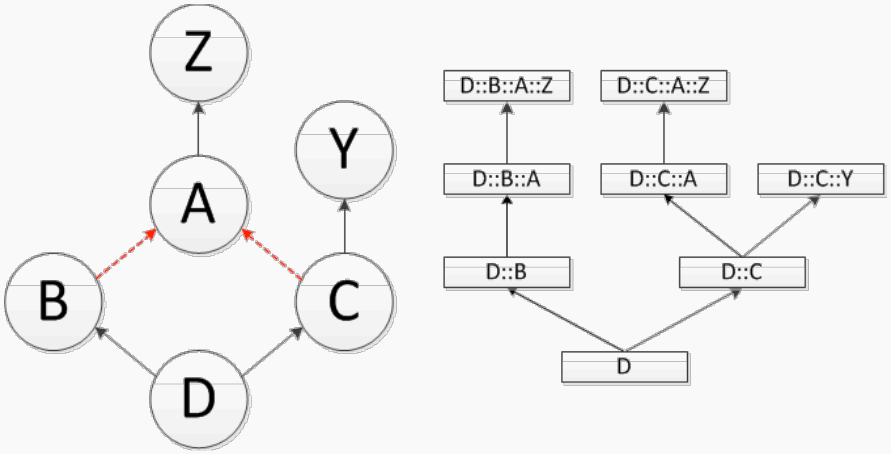
Encoding+Handling	Substitutability	Relational	Repeat Case	Multiple Inheritance	Speed
Poly+if	+	+	+	+	577
Poly+memoization device	+	+	+	+	62
Tag+switch	–	–	–	–	42
Tag+linearization	*	–	–	–	45
Union+switch	–	–	–	n/a	39
Poly+visitors	*	–	–	+	55
Poly+exceptions	+	–	–	+	17K

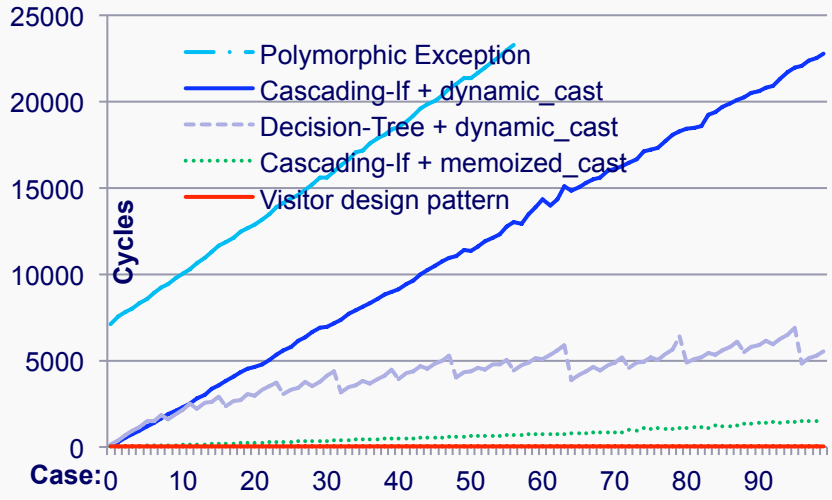
```
Match(expr)
  Case(P1) ...
  Case(Pn) ...
  Otherwise() ...
EndMatch
```

Problem of Open Type Switching

Parasol

- Classes are:
  - Extensible
    - Important: Separate compilation
    - Important: Dynamic linking
  - Hierarchical
    - Multiple Inheritance
    - Up-, down- and cross-casts
    - Cast is not a no-op
    - Ambiguities
- Existing approaches
  - Closed world: jump tables
    - Unrealistic for modern C++ use
  - Open world: constant-time subtype tests + decision trees
    - Most are not suitable for repeated multiple inheritance
    - Most require computations or run-time code generation at load time
    - Time increases with case number

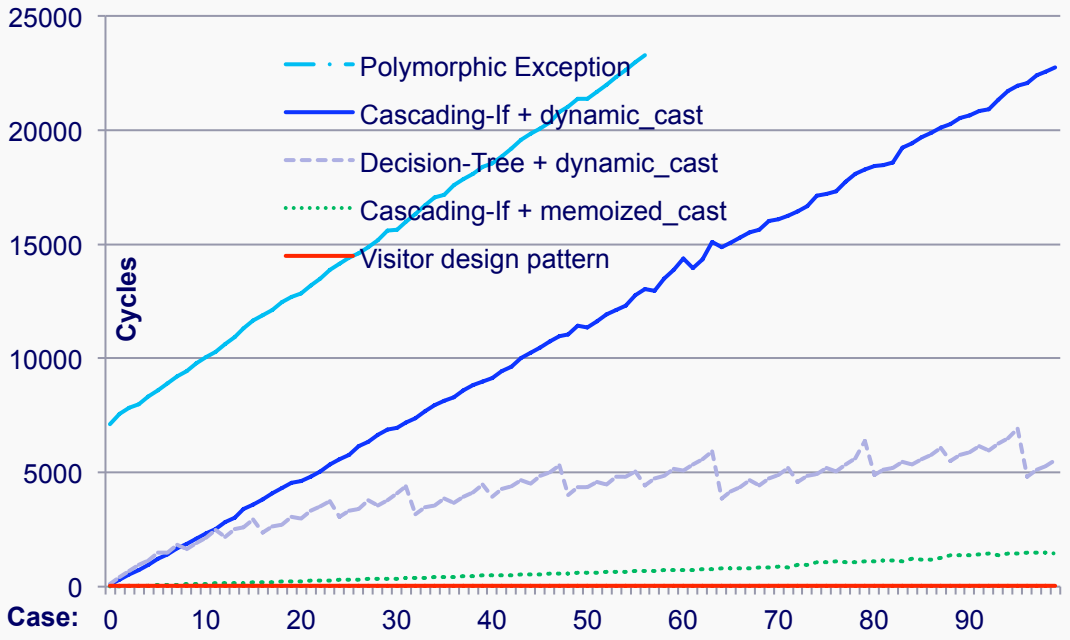




Open but Inefficient Solution

Parasol

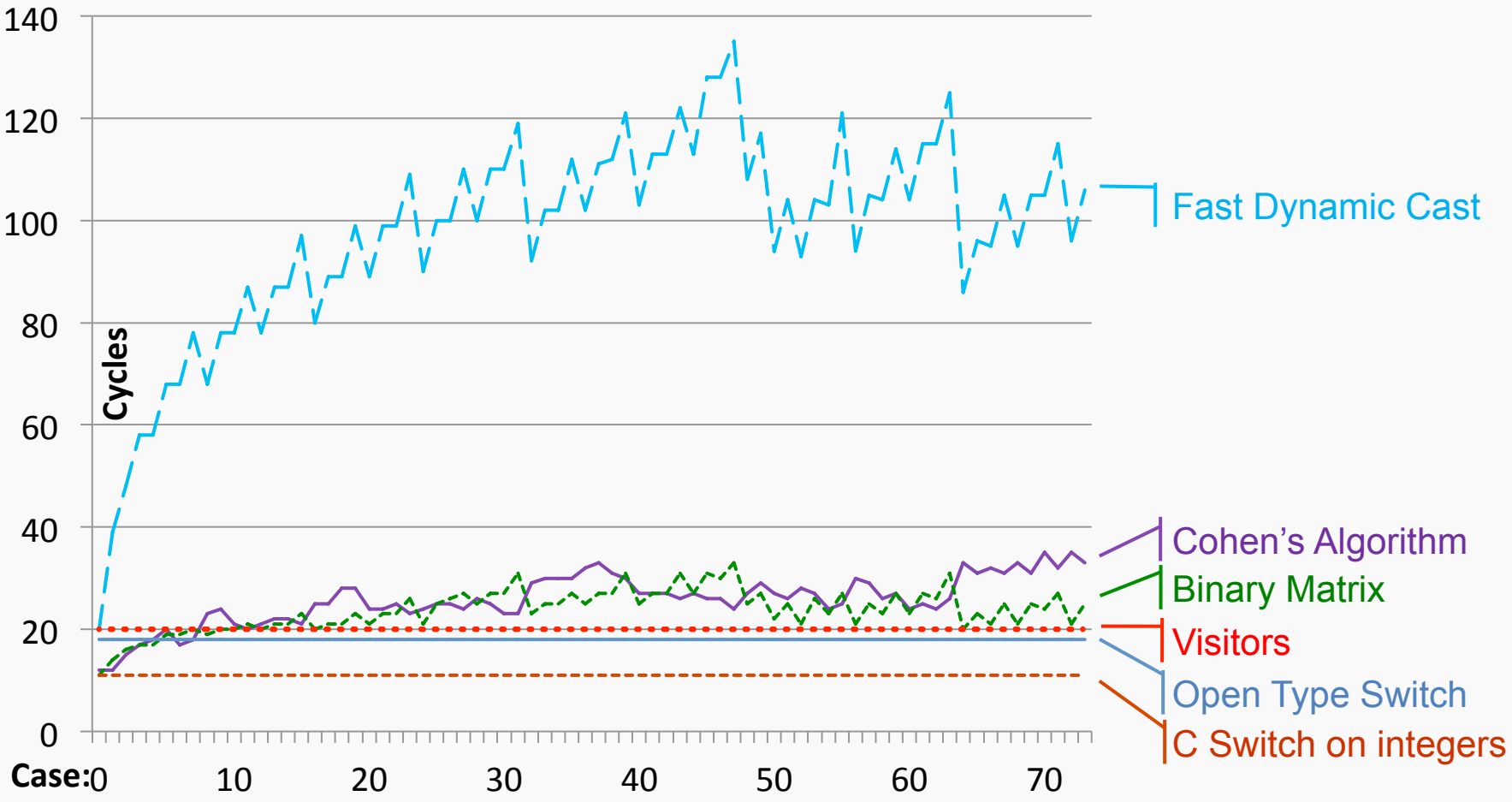
```
switch (object)
{
case Type_1: action_1;
case Type_2: action_2;
  ...
case Type_n: action_n;
}
```




```
if (Type_1* match=dynamic_cast<T_1*>(object)) { action_1; } else
if (Type_2* match=dynamic_cast<T_2*>(object)) { action_2; } else
  ...
if (Type_n* match=dynamic_cast<T_n*>(object)) { action_n; }
```

Problem of Efficient Type Switching

Parasol



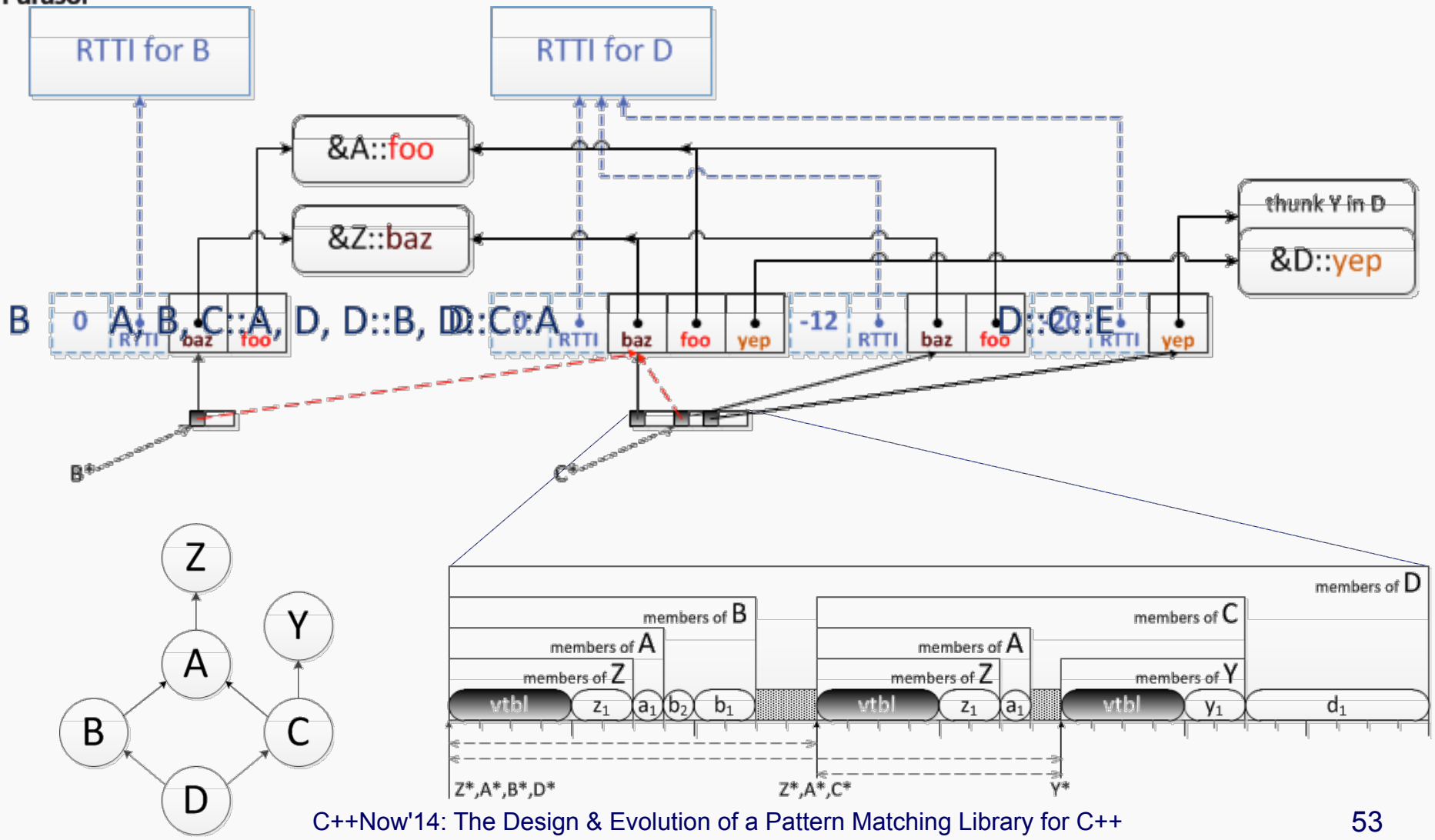
Key Features of Our Solution

Parasol

- Memoization device
  - Maps types to execution paths
  - Uses dynamic types of objects
- Hash of dynamic type
  - A pointer to a virtual function table (v-table) identifies
    - Type of object
    - Sub-object offset
  - High-quality hashing needed for performance
    - Experiments to achieve perfect and/or compact hashing
    - We use the structure of v-table pointers

Uniqueness of V-Table Pointers

Parasol



Intermediate Solution

Parasol

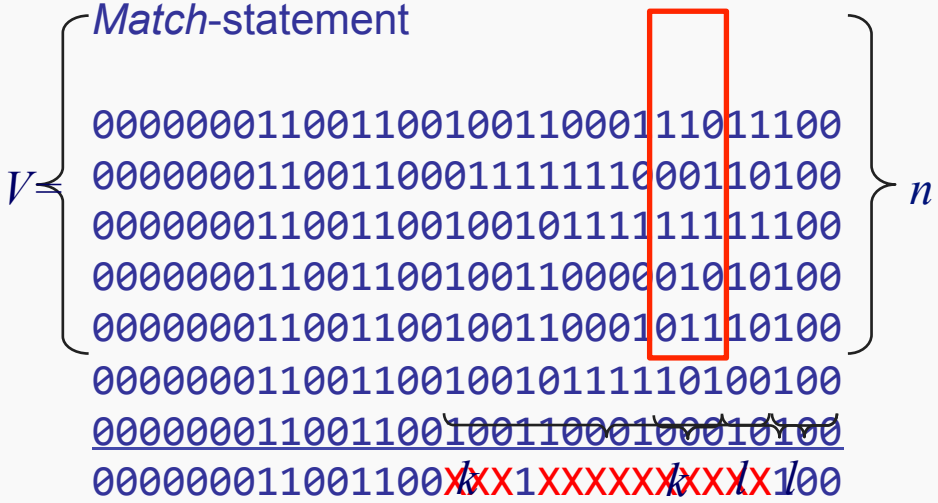
- We use:
  - source sub-object as a hash map key
  - P_i(x) ≡ dynamic_cast<T_i^*>(x)!=0
- We map it to:
  - offset to target sub-object
  - jump target
- On first entry
  - memoize this-pointer offset
  - memoize jump target
- On subsequent entries
  - jump to target
  - adjust this-pointer

```
struct info
{
    ptrdiff_t offset;
    size_t    target;
};

// Case(Ti* t) ----- Ti ≡ Typei -----

if (auto t = dynamic_cast<Ti*>(x))
{
    n.offset = int(t)-int(x);
    n.target = i;
case i:
    auto match =
       adjust_ptr<Ti>(x,n.offset);
    si;
}
```

Structure of V-Table Pointers

Parasol

$V = \{v\downarrow 1, \ldots, v\downarrow n\}$ is a set of v-table pointers passed through a given *Match*-statement



- Probability of conflict:
  - $p\downarrow kl\,(V) = m/n$
  - $m = n - |H\downarrow kl\,(V)|$

For each such set V we:

- Use cache of $2\uparrow k$ entries addressed by:

  $Hkl(v) = v/2\uparrow l \bmod 2\uparrow k$

- Vary parameters:
  - $k \in [K, K+1] : 2\uparrow K - 1 < n \leq 2\uparrow K$
  - $l \in [2, 32-k]$
- Find optimal $H\downarrow kl\uparrow V$ as:
  - $(k,l) = \arg\max_{k,l} |H\downarrow kl\,(V)|$
  - $|H\downarrow kl\,(V)|$ - *size of image on* V

Handling Multiple Arguments

Parasol

- Morton Order (Z-order)
  - a function that maps *N*-dimensional data to one dimension while preserving locality of the data points
  - obtained by interleaving the binary representations of all coordinates
- Parameters: $k, l_1, \ldots, l_N$
  - Minimize probability of conflict
  - But not every reconfiguration!

Visitors Comparison

Parasol

		Open					
		G++		MS Visual C++			
		Lnx	Win	PGO		w/o PGO	
		x86-32	x86-32	x86-32	x86-64	x86-32	x86-64
	REP	16%	14%	1%	18%	2%	37%
	SEQ	56%	12%	48%	22%	2%	46%
	RND	56%	0%	9%	19%	5%	46%
Forwarding	REP	33%	22%	8%	17%	24%	36%
Forwarding	SEQ	55%	233%	135%	135%	193%	32%
Forwarding	RND	78%	25%	3%	4%	13%	23%

Lnx: Dell Dimension® desktop with

- Intel® Pentium® D (Dual Core) CPU at 2.80 GHz; 1GB of RAM; Fedora Core 13
- G++ 4.4.5 executed with -O2; x86 binaries

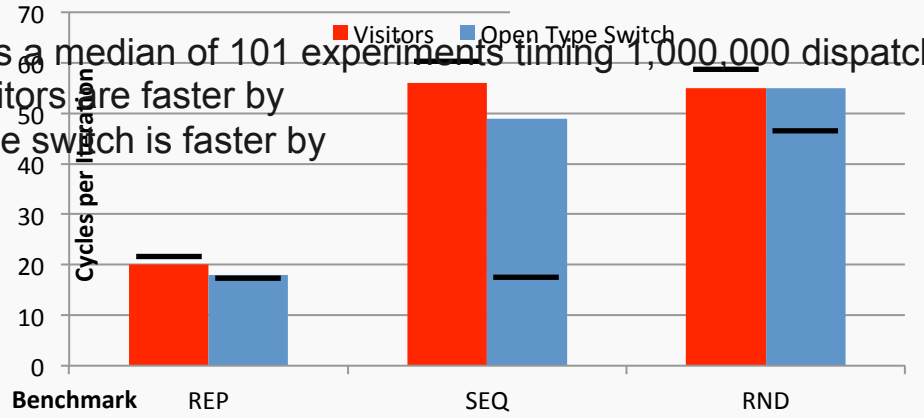
Win: Sony VAIO® laptop with

- Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Pro.
- G++ 4.6.1 / MinGW executed with -O2; x86 binaries
- MS Visual C++ 2010 Professional x86/x64 binaries with and without Profile-Guided Optimizations

Each number represents a median of 101 experiments timing 1,000,000 dispatches each

18% - percentage visitors are faster by

14% - percentage type switch is faster by

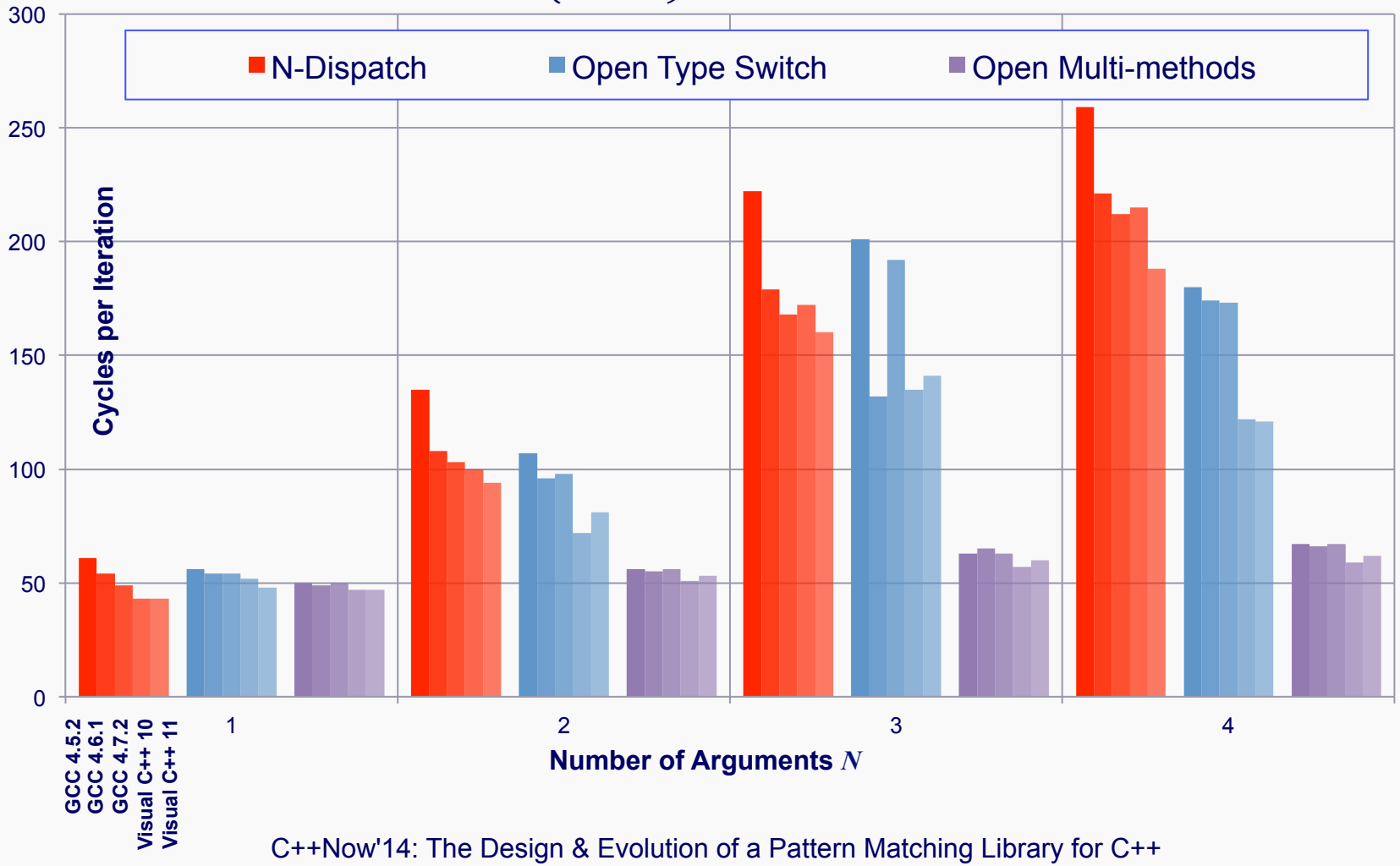


Multi-Argument Comparison

Parasol

n is the number of subobjects in a class hierarchy of arguments

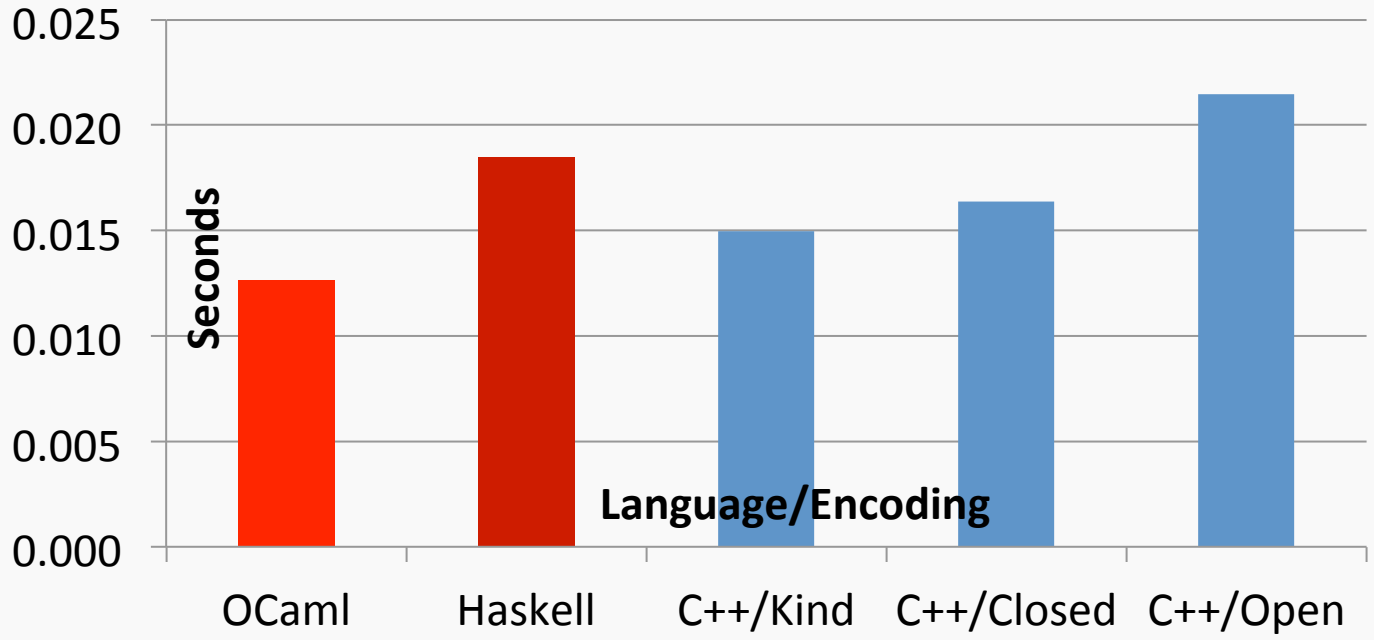
Memory: $n\uparrow N + n\uparrow N-1 + \ldots + n\uparrow 2 + n + 1$ $(2^{N+3})n\uparrow N + N + 7$ $n\uparrow N + Nn + N$



Other Languages Comparison

Parasol

Our library code is approximately as fast as natively compiled functional code. We will squeeze even more with a language solution.



Note: This is not a detailed language comparison, but we compare well with “the gold standard” for pattern matching

Real World Class Hierarchies

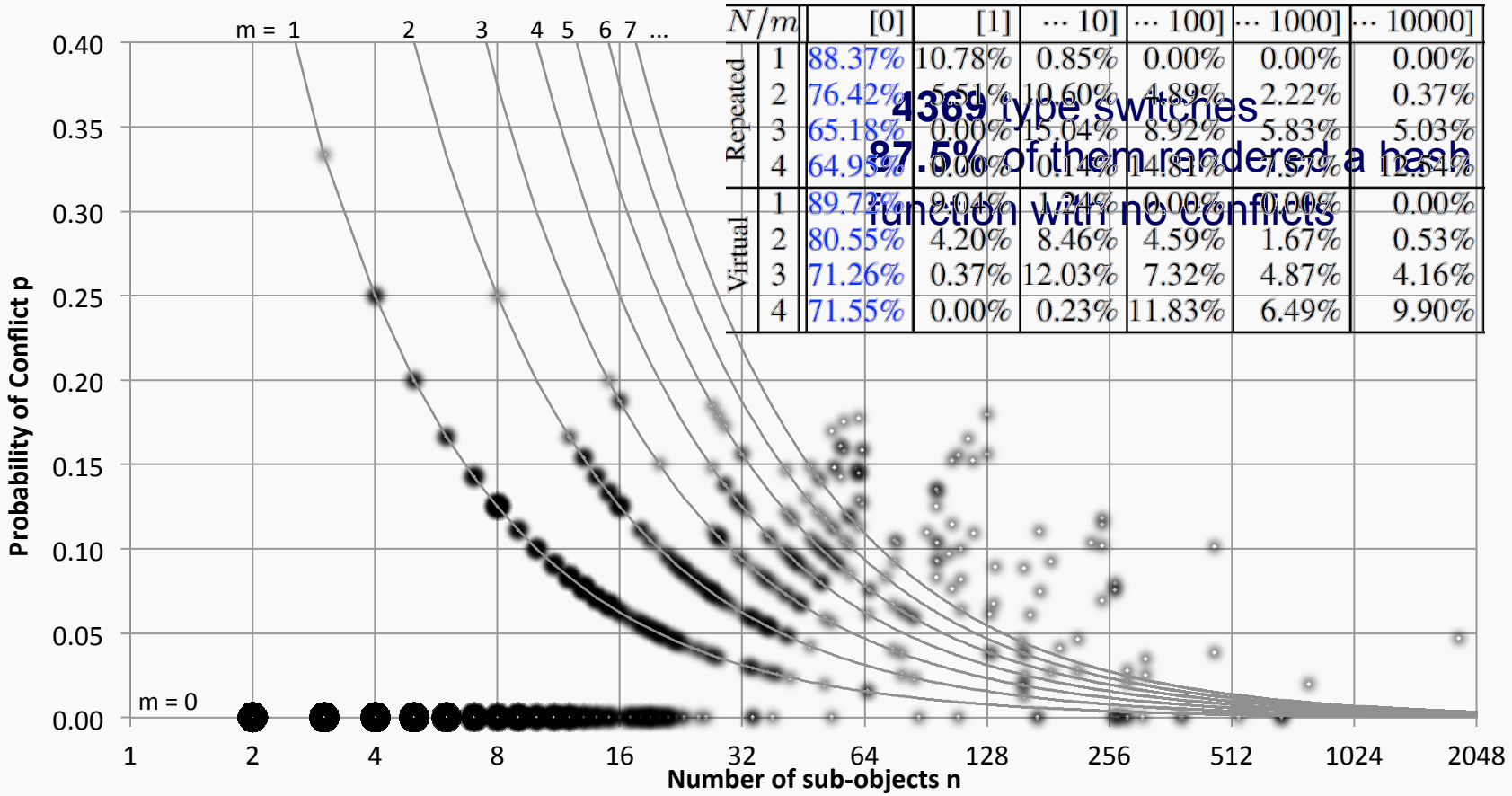
Parasol

Lib	Language	Classes	Paths	Height	Roots	Leafs	Both	Parents		Children	
								avg	max	avg	max
DG2	Smalltalk	534	534	11	2	381	1	1	1	3.48	59
DG3	Smalltalk	1356	1356	13	2	923	1	1	1	3.13	142
ET+	C++	370	370	8	87	289	79	1	1	3.49	51
GEO	Eiffel	1318	13798	14	1	732	0	1.89	16	4.75	323
JAV	Java	604	792	10	1	445	0	1.08	3	4.64	210
LOV	Eiffel	436	1846	10	1	218	0	1.72	10	3.55	78
NXT	Objective-C	310	310	7	2	246	1	1	1	4.81	142
SLF	Self	1801	36420	17	51	1134	0	1.05	9	2.76	232
UNI	C++	613	633	9	147	481	117	1.02	2	3.61	39
VA2	Smalltalk	3241	3241	14	1	2582	0	1	1	4.92	249
VA2	Smalltalk	2320	2320	13	1	1868	0	1	1	5.13	240
VW1	Smalltalk	387	387	9	1	246	0	1	1	2.74	87
VW2	Smalltalk	1956	1956	15	1	1332	0	1	1	3.13	181
	Overalls	**15246**	**63963**	**17**	**298**	**10877**	**199**	**1.11**	**16**	**3.89**	**323**

% sub-hierarchies	1%	3%	5%	10%	20%	25%	50%	64%	100%
with more than … classes	700	110	50	20	10	7	3	2	1

Efficiency of Hashing

Parasol



	N/m	[0]	[1]	··· 10]	··· 100]	··· 1000]	··· 10000]
Repeated	1	88.37%	10.78%	0.85%	0.00%	0.00%	0.00%
	2	76.42%	5.51%	10.60%	4.89%	2.22%	0.37%
	3	65.18%	0.00%	15.04%	8.92%	5.83%	5.03%
	4	64.93%	0.00%	0.14%	14.81%	7.57%	12.54%
Virtual	1	89.72%	9.04%	1.24%	0.00%	0.00%	0.00%
	2	80.55%	4.20%	8.46%	4.59%	1.67%	0.53%
	3	71.26%	0.37%	12.03%	7.32%	4.87%	4.16%
	4	71.55%	0.00%	0.23%	11.83%	6.49%	9.90%

4369 type switches
87.5% of them rendered a hash function with no conflicts

Conflicts	0	1	2	3	4	5	6	>6
Type Switches	87.50%	5.58%	2.63%	0.87%	0.69%	0.69%	0.30%	1.76%

Interaction of Patterns & Type Switch

Parasol

Mach7 considers

- Is Case-clause a valid pattern?
- Which subjects are polymorphic?
- What is expected type of the subject?

```
template <typename... P>
struct some_pattern : std::tuple<P...> {
 static_assert(conjunction<is_pattern<P>...>::value,“Not a pattern");
 some_pattern(P&&... p);
 template <typename S> struct accepted_type_for { typedef S type; };
 template <typename S> bool operator()(const S& s) const;
};
```

Pattern Matching Overhead

Parasol

15% - percentage hand crafted code is faster by 15% - percentage pattern matching code is faster by

		Patterns as Expr. Templ.					Patterns as Objects				
		G++			Visual C++		G++			Visual C++	
Test	Patterns	4.5.2	4.6.1	4.7.2	10	11	4.5.2	4.6.1	4.7.2	10	11
factorial$_0$	1,v,_	15%	13%	17%	85%	35%	347%	408%	419%	2121%	1788%
factorial$_1$	1,v	0%	6%	0%	83%	21%	410%	519%	504%	2380%	1812%
factorial$_2$	1,n+k	7%	9%	6%	78%	18%	797%	911%	803%	3554%	3057%
fibonacci	1,n+k,_	17%	2%	2%	62%	15%	340%	431%	395%	2730%	2597%
gcd$_1$	v,n+k,+	21%	25%	25%	309%	179%	1503%	1333%	1208%	8876%	7810%
gcd$_2$	1,n+k,	5%	13%	19%	373%	303%	962%	1080%	779%	5332%	4674%
gcd$_3$	1,v	1%	0%	1%	38%	15%	119%	102%	108%	1575%	1319%
lambdas	&,v,C,+	58%	54%	56%	29%	34%	837%	780%	875%	259%	289%
power	1,n+k	10%	8%	13%	50%	6%	291%	337%	388%	1950%	1648%

Our Solution

Previous Approaches

Compilation Time Overhead

Parasol

15% - handcrafted compilation is faster by 15% - pattern matching compilation is faster by

		Patterns as Expr. Templ.			Patterns as Objects		
		G++	Visual C++		G++	Visual C++	
Test	Patterns	4.7.2	10	11	4.7.2	10	11
factorial$_0$	1,v,_	1.65%	1.65%	2.95%	7.10%	10.00%	10.68%
factorial$_1$	1,v	2.46%	1.60%	10.92%	7.14%	0.00%	1.37%
factorial$_2$	1,n+k	2.87%	3.15%	3.01%	8.93%	4.05%	3.83%
fibonacci	1,n+k,_	3.66%	1.60%	2.95%	11.31%	4.03%	1.37%
gcd$_1$	v,n+k,+	4.07%	4.68%	0.91%	9.94%	2.05%	8.05%
gcd$_2$	1,n+k,	1.21%	1.53%	0.92%	8.19%	2.05%	2.58%
gcd$_3$	1,v	2.03%	3.15%	7.86%	5.29%	2.05%	0.08%
lambdas	&,v,C,+	18.91%	7.25%	4.27%	4.57%	3.82%	0.00%
power	1,n+k	2.00%	6.40%	3.92%	8.14%	0.13%	4.02%

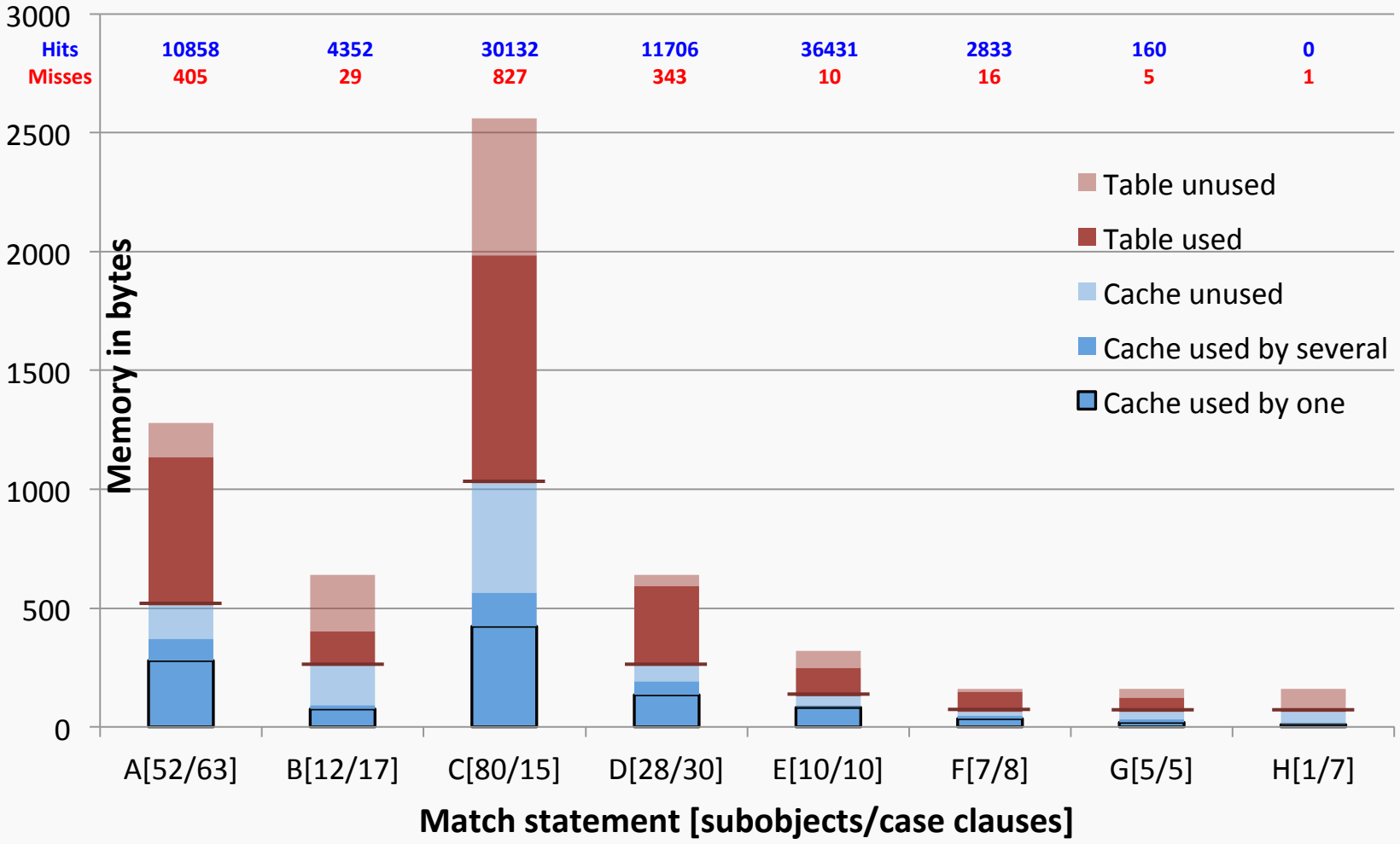
Pivot's C++ Pretty-printer: Speed

Parasol

Header	LOC	Faster Time	Performance	Header	LOC	Faster Time	Performance
climits	44	765	15.26%	cmath	338	5909	3.4%
cassert	47	781	16.09%	cwchar	430	6956	3.86%
cerrno	50	894	11.79%	functional	588	6352	2.39%
cstdarg	51	301	21.64%	iosfwd	981	14699	2.17%
csignal	53	1061	8.17%	utility	1095	15848	2.3%
cstddef	55	943	12.29%	limits	1521	20512	1.53%
cfloat	61	1188	10.44%	valarray	1551	30552	1.1%
csetjmp	65	1172	8.33%	iterator	2356	29158	2.65%
cctype	98	1667	3.01%	numeric	2481	30545	3.7%
cwctype	98	2137	10.88%	memory	3171	38761	4.33%
ctime	104	1714	8.74%	stdexcept	3907	50049	3.49%
exception	107	1803	10.02%	algorithm	5467	63553	3.37%
cstring	144	2560	2.03%	deque	5477	68619	2.75%
typeinfo	145	2267	8.11%	stack	5563	68893	3.22%
new	156	2255	7.94%	list	5981	72341	3.63%
cstdio	195	3482	1.58%	vector	5985	73268	3.94%
cstdlib	195	3211	0.86%	queue	9851	115871	3.6%

Pivot's C++ Pretty-printer: Memory

Parasol



Mach7: Problems w/ STL Containers

Parasol

- STL containers are not recursive data structures
  - They are not expressed in terms of themselves
  - Ranges also are not
  - But they can represent subparts of themselves
- We need patterns describing sets, sequences, collections of values

Mach7: Problems w/ STL Containers

Parasol

```
struct List : Term { Term* head; List* tail; };
struct Structure : Term {
    std::string name; std::vector<Term*> terms;
};

bool operator==(const Term& l, const Term& r) {
    var<std::string> s; var<int> n; var<const Term&> h,t; // rng<????> p;

    Match(l,r)
      Case(C<List>(&h,&t),      C<List>(&+h,&+t)              ) return true;
      Case(C<List>(&h,nullptr), C<List>(&+h,nullptr)          ) return true;
    //Case(C<Structure>(s,n,p), C<Structure>(+s,+n,all(+**p++))) return true;
      ...
    EndMatch
}
```

Conclusions

Parasol

- Open Pattern Matching for C++:
  - Open (w.r.t. Patterns & Objects analyzed)
  - Faster than alternatives (visitors, patterns as objects)
  - Non-intrusive
- *Mach7*:
  - A Library Solution
  - Open Sourced under BSD license: http://parasol.tamu.edu/mach7/
    will eventually live here: https://github.com/solodon4/SELL
  - Currently works for G++ 4.4+ and Visual C++ 2010+
- A Roadmap for Future Language Solution
  - A baseline for features
  - A baseline for performance
  - Transition for implementations
- A hope that future Object-Oriented languages will consider including support for pattern matching a-priori instead of a-posteriori

Ongoing and Future Work

Parasol

Ongoing

- Fix Clang support
- Lock-free vtbl-map
- boost::variant etc.
- Sequence patterns
- Improved type-checking
- Improved diagnostics
- Older compilers

Future

- Generic redundancy checking
- Generic implication
- Two-way matching
- Language feature

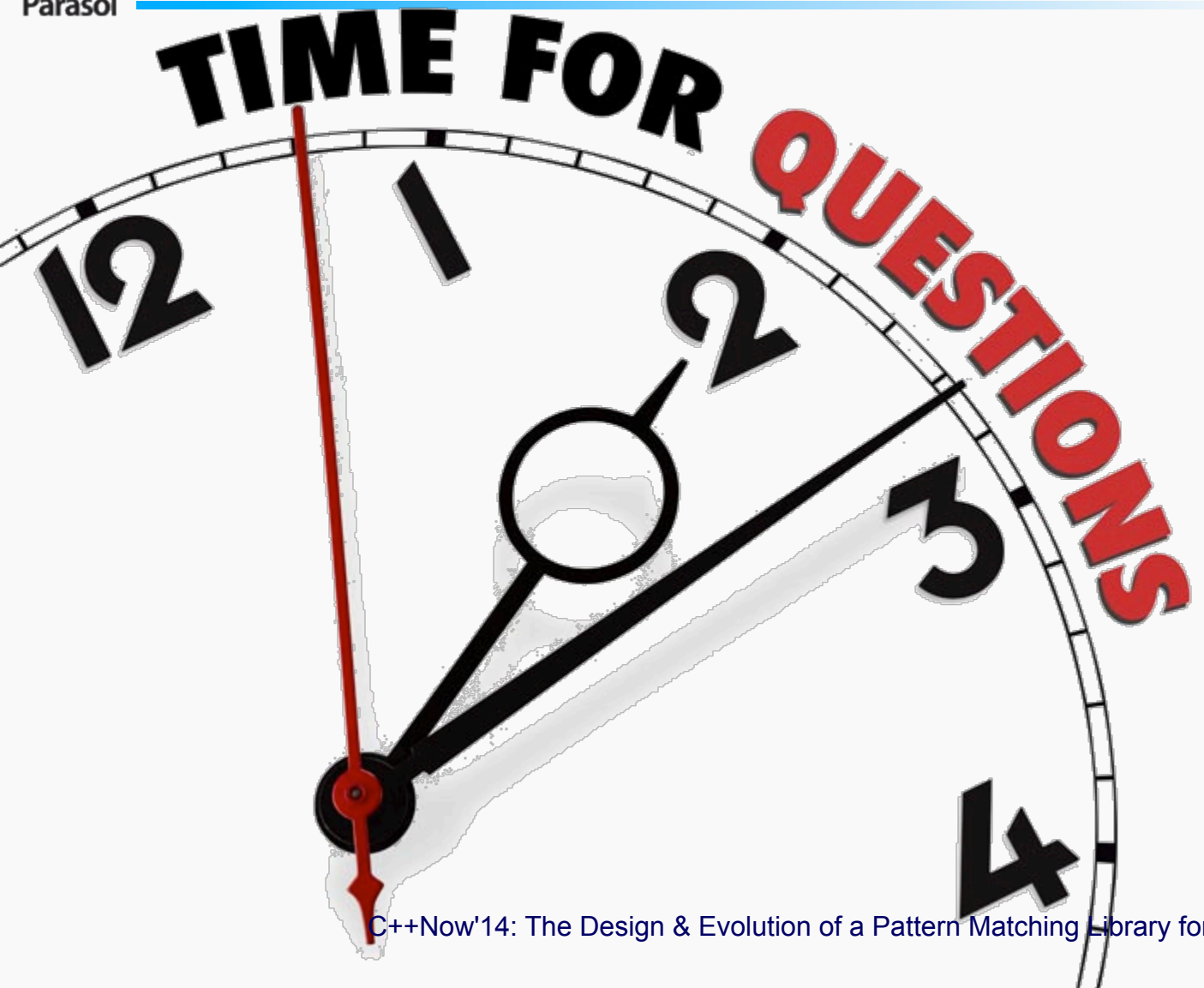
References

- Y.Solodkyy. "Simplifying the Analysis of C++ Programs" Ph.D. Thesis. Texas A&M University. August 2013. [pdf, slides]
- Y.Solodkyy, G.Dos Reis, B.Stroustrup. "Open Pattern Matching for C++: Extended Abstract" In Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity (SPLASH '13). ACM, New York, NY, USA, pp. 97-98. [pdf, slides, notes, poster, project]
- Y.Solodkyy, G.Dos Reis, B.Stroustrup. "Open Pattern Matching for C++" In Proceedings of the 12th international conference on Generative programming: concepts & experiences (GPCE '13). ACM, New York, NY, USA, pp. 33-42. [pdf, slides, notes, poster, project]
- Y.Solodkyy, G.Dos Reis, B.Stroustrup. "Open and Efficient Type Switch for C++" In Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12). ACM, New York, NY, USA, pp. 963-982. [pdf, slides, notes, poster, extras, project]
- All of the above is available on my university page: https://parasol.tamu.edu/~yuriys/

Old: http://parasol.tamu.edu/mach7/
Soon: https://github.com/solodon4/SELL
E-mail: yuriy.solodkyy@gmail.com

Thank You!

Parasol



Acknowledgements

Abe Skolnik
Andrew Sutton
Bjarne Stroustrup
Emil 'Skeen' Madsen
Gabriel Dos Reis
Gregory Berkolaiko
Jaakko Järvi
Jason Wilkins
Jasson Casey
Michael Lopez
Karel Driesen
Lawrence Rauchwerger
Luc Maranget
Peter Pirkelbauer
Sergiy Butenko
Suhasini Subba Rao
Vahideh Kamranzadeh
Xavier Leroy

Microsoft

Approaches to Dealing with AST

Parasol

	Extensibility of Functions	Extensibility of Data	Type Safe	Multiple Inheritance	Relational	Nesting	Retroactive	Local Reasoning	No Control Inversion	Redundancy Checking	Completeness Checking	Speed in Cycles
Discriminated Union	+	–	–	–	–	–	–	+	+	–	*	39
Tag Switch	+	*	–	–	–	–	–	+	+	–	–	45
Type Testing	+	+	+	+	–	–	+	+	+	–	–	577
Polymorphic Exception	+	+	+	+	–	–	–	+	+	*	*	17K
Virtual Functions	–	+	+	+	–	–	–	–	–		–	29
Visitor Design Pattern	+	*	+	+	*	–	–	+	–		–	55
Open Multi-methods	+	+	+	+	+	–	+	–	–		+	38
Open Type Switch	+	+	+	+	+	–	+	+	+	*	–	62
Open Pattern Matching	+	+	+	+	+	+	+	+	+	*	–	70