Pattern Matching for C++

Yuriy Solodkyy · Gabriel Dos Reis · Bjarne Stroustrup

Algebraic Data Types

Simple expression language:

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

OCaml

```
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)
      ;;
```

```
let factorize e =
  match e with
    Plus(Times(e1,e2), Times(e3,e4)) when e1 = e3
        -> Times(e1, Plus(e2,e4))
  | Plus(Times(e1,e2), Times(e3,e4)) when e2 = e4
        -> Times(Plus(e1,e3), e4)
  |   e -> e
  ;;
```

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

```
complex ::= Pole real real
view complex ::= Cart real real
  in  (Pole r t) = Cart (r * cos t) (r * sin t)
  out (Cart x y) = Pole (sqrt(x^2 + y^2)) (atan2 x y)
```

C++

```
struct Expr          { virtual ~Expr(){} };
struct Value  : Expr { int value ; };
struct Plus   : Expr { Expr* exp1; Expr* exp2; };
struct Minus  : Expr { Expr* exp1; Expr* exp2; };
struct Times  : Expr { Expr* exp1; Expr* exp2; };
struct Divide : Expr { Expr* exp1; Expr* exp2; };

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

```
const Expr* factorize(const Expr* e)
{
    variable<const Expr*> e1, e2, e3, e4;
    if (match<Plus>(match<Times>(e1,e2),
                    match<Times>(e3 |= e1==e3,e4))(e))
        return new Times(e1, new Plus(e2,e4));
    if (match<Plus>(match<Times>(e1,e2),
                    match<Times>(e3,e4 |= e2==e4))(e))
        return new Times(new Plus(e1,e3), e4);
    return e;
}
```

```
double power(double x, int n)
{
   variable<int> m;
   if (match<int>(0)(n))     return 1.0;
   if (match<int>(1)(n))     return x;
   if (match<int>(m*2)(n))   return sqr(power(x,m));
   if (match<int>(m*2+1)(n)) return x*sqr(power(x,m));
}
```

```
template <typename T>
    using Cartesian = view<std::complex<T>>;
template <typename T>
    using Polar     = view<std::complex<T>, polar>;

std::complex<double> c; double a,b,r,f;

if (match<  complex<double>>(a,b)(c)) // default
if (match<Cartesian<double>>(a,b)(c)) // same as above
if (match<    Polar<double>>(r,f)(c)) // view
```

Syntax

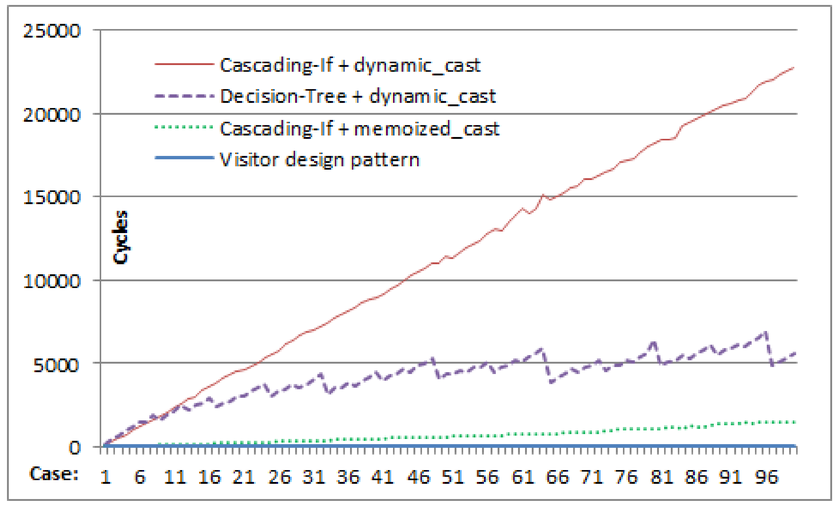
match statement	M ::=	$Match(e)\ [Cs^*]^*\ EndMatch$
case clause	C ::=	$Case(T\,[,x]^*)$
	|	$Que(\ T\,[,\omega]^*)$
	|	$Otherwise([,x]^*)$
target expression	T ::=	$\tau \mid l \mid \nu$
view	ν ::=	$\text{view}\langle\tau, l\rangle$
match expression	m ::=	$\pi(e)$
pattern	π ::=	$\_ \mid \eta \mid \varrho \mid \mu \mid \varsigma \mid \chi$
extended pattern	ω ::=	$\pi \mid c \mid x$
tree pattern	μ ::=	$\text{match}\langle\nu\vert\tau\,[,l]\rangle(\omega^*)$
guard pattern	ϱ ::=	$\pi \models \xi$
n+k pattern	η ::=	$\chi \mid \eta \oplus c \mid c \oplus \eta \mid \ominus\eta \mid (\eta) \mid \_^{wildcard}$
wildcard pattern		
variable pattern	χ ::=	$\kappa \mid \iota$
value pattern	ς	$^{value\langle\tau\rangle}$
xt variable	κ	$^{variable\langle\tau\rangle}$
xt reference	ι	$^{var\_ref\langle\tau\rangle}$
xt expression	ξ ::=	$\chi \mid \xi \oplus c \mid c \oplus \xi \mid \ominus\xi \mid (\xi) \mid \xi \oplus \xi$
layout	l ::=	c^{int}
unary operator	$\ominus \in$	$\{*, \&, +, -, !, \sim\}$
binary operator	$\oplus \in$	$\{*, /, \%, +, -, \ll, \gg, \&, \wedge, \vert, <, \leq, >, \geq, =, \neq, \&\&, \vert\vert\}$
type-id	τ	C++[19, §A.7]
statement	s	C++[19, §A.5]
expression	e^{τ}	C++[19, §A.4]
constant-expression	c^{τ}	C++[19, §A.4]
identifier	x^{τ}	C++[19, §A.2]

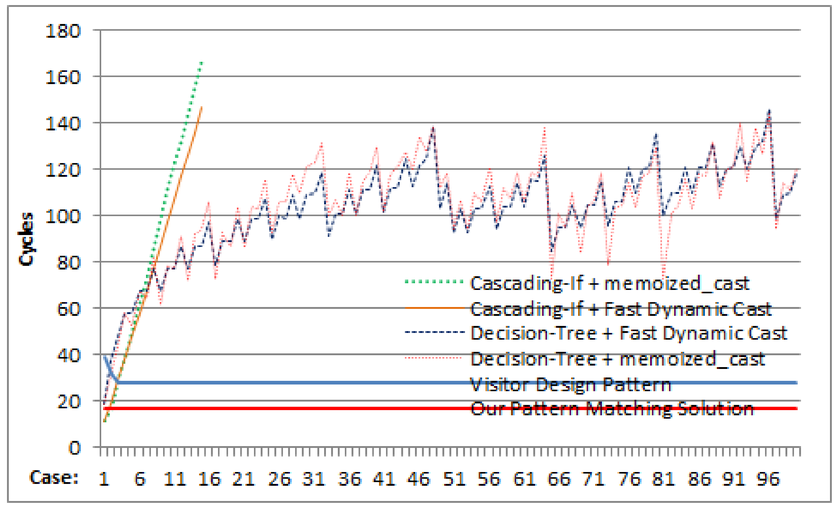
Parasol

Smarter computing.

Texas A&M University

Efficient Type Switching





Performance Evaluation

42% pattern matching is faster than visitors.
42% visitors are faster than pattern matching

		G++/32				MS Visual C++/32				MS Visual C++/64			
Syntax		Unified		Special		Unified		Special		Unified		Special	
Encoding		Open	Tag	Open	Tag	Open	Tag	Open	Tag	Open	Tag	Open	Tag
	Repetitive	55%	116%	55%	216%	4%	61%	4%	124%	14%	20%	0%	47%
	Sequential	1%	43%	3%	520%	9%	13%	3%	34%	2%	34%	1%	14%
	Random	0%	29%	1%	542%	17%	18%	18%	43%	27%	7%	27%	16%
Forward	Repetitive	67%	88%	67%	79%	10%	16%	10%	31%	5%	5%	6%	9%
	Sequential	87%	250%	90%	259%	153%	168%	153%	185%	130%	132%	145%	118%
	Random	28%	32%	27%	31%	19%	11%	18%	24%	5%	2%	6%	10%