

Metaprogramación

Computando en tiempo de compilación

Marcelo Arroyo

U.N.R.C.

FCEIA, JCC - 2010

- 1 **Introducción**
 - Metaprogramación
 - Herramientas de metaprogramación
- 2 **Lenguajes de dos niveles**
 - C++ templates
 - Converge
 - Template Haskell
- 3 **Lenguajes de dominio específicos**
 - DSLs en C++
- 4 **Programación orientada a lenguajes**
 - Lenguajes extensibles
 - Ventajas y desventajas de la metaprogramación
 - Conclusión

Metaprogramación

Conjunto de técnicas y herramientas para manipular programas en tiempo de compilación

Metaprograma

- escrito en algún **metalenguaje**
- el programa manipulado se denomina **lenguaje objeto**
- un lenguaje **reflexivo** permite ser su propio metalenguaje

Implementación

- Macros
- Sistemas de transformación de programas

Metaprogramación

Conjunto de técnicas y herramientas para manipular programas en tiempo de compilación

Metaprograma

- escrito en algún **metalenguaje**
- el programa manipulado se denomina **lenguaje objeto**
- un lenguaje **reflexivo** permite ser su propio metalenguaje

Implementación

- Macros
- Sistemas de transformación de programas

Metaprogramación

Conjunto de técnicas y herramientas para manipular programas en tiempo de compilación

Metaprograma

- escrito en algún **metalenguaje**
- el programa manipulado se denomina **lenguaje objeto**
- un lenguaje **reflexivo** permite ser su propio metalenguaje

Implementación

- Macros
- Sistemas de transformación de programas

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Aplicaciones

- Programación genérica
 - C++ STL
 - Bibliotecas C++ Boost
- Optimización a nivel de aplicación
 - Biltz++: Object Oriented Scientific Computation
- Independencia de herramientas externas
- Evaluación parcial
- Programación orientada a aspectos
- Lenguajes de dominio específico (DSLs)
 - fp++
 - lp++
 - ag++

Lenguajes de programación

- LISP: quasiquote expressions
- C++ templates
- Template Haskell
- Converge
- Scala

Herramientas de generación de procesadores lenguajes

- ADF-SDF Meta Environment
- Stratego/XT
- JetBrains MTS
- AntLR, ...

Lenguajes de programación

- LISP: quasiquote expressions
- C++ templates
- Template Haskell
- Converge
- Scala

Herramientas de generación de procesadores lenguajes

- ADF-SDF Meta Environment
- Stratego/XT
- JetBrains MTS
- AntLR, ...

C++ templates: Características

- Soporte para programación genérica (tipos y funciones parametrizadas)
- Especialización de templates y pattern matching
 - Templates recursivos
 - Permiten implementar evaluación parcial
 - Computación estática
 - Implementación de instropección (estática)
 - Static checking

C++ templates

Fragmento Turing-computable

C++ templates

Computación estática

```
template<int n>
struct factorial {
    static const int result =
        n*factorial<n-1>::result;
};

template<>    // specialization
struct factorial<0> {
    static const int result = 1;
};

int fac12 = factorial<12>::result;
```

C++ templates

Generación de código

```
template<int i>
inline float meta_dot(float a[], float b[])
{
    return meta_dot<i-1>(a,b) + a[i] * b[i];
}
```

```
template<>
inline float meta_dot<0>(float a[], float b[])
{
    return a[0] * b[0];
}
```

```
float x[3], y[3], z = meta_dot<2>(x,y);
// z=x[0]*y[0]+x[1]*y[1]+x[2]*y[2]
```

C++ templates

Calculando tipos: traits

```
template typename<T>    — default. T -> T
struct avg_traits {
    typedef T type;
};
template typename<>    — int -> float
struct avg_traits<int> {
    typedef float type;
};
typename avg_traits<int>::type r;
r = sum_array(a,N)/N;
```

Converge

Un ejemplo

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| ${x}*${expand_power(n-1,x)} |]

func mk_power(n):
  return [|
    func (&x):
      return ${expand_power(n, [| &x |])}
  |]

power3 := $<mk_power(3)>
-- power3 = func(x): return x*x*x+1
```

Splice

$\$<expr>$ se evalúa (expande) en compilación

Quasi-quotes e inserciones

- Quasi-quote: $[| expr |]$ denota un AST
- Inserciones:
 - $\${e}$ evalúa una expresión y retorna el AST resultante dentro del *quasi-quote* en el que aparecen.
Renombra las variables apareciendo en e por nombres *frescos*
 - $\$c{e}$ idem al anterior pero sin renombre de variables, permitiendo la captura de variables libres
 - $\$p{e}$ pragma: evalúa la expresión y descarta su resultado

Splice

$\$<expr>$ se evalúa (expande) en compilación

Quasi-quotes e inserciones

- Quasi-quote: $[| expr |]$ denota un AST
- Inserciones:
 - $\$\{e\}$ evalúa una expresión y retorna el AST resultante dentro del *quasi-quote* en el que aparecen.
Renombra las variables apareciendo en e por nombres *frescos*
 - $\$c\{e\}$ idem al anterior pero sin renombre de variables, permitiendo la captura de variables libres
 - $\$p\{e\}$ pragma: evalúa la expresión y descarta su resultado

Metaprogramación en Template-Haskell

Template-Haskell

Macro-procesador escrito en Haskell e integrado en compiladores e intérpretes por bootstrapping.

Ejemplo

```
import Language.Haskell.TH
```

```
tupleRep :: Int -> Q Exp
```

```
tupleRep n = do id <- newName "x"
```

```
    return
```

```
        $ LamE (VarP id)
```

```
        (TupE $ replicate n $ VarE id)
```

```
tupleReplicate 3 ⇒ (\x.(x,x,x))
```

Template-Haskell

Splices: denotados como $\$id$ o $\$(expr)$

```
main = do print $(tupleRep 2) 1 — (1,1)
```

Quotation: generadores de ASTs (mónada $Q\ t$)

- $[| expr |]$, retorna un valor de tipo $Q\ Exp$
- $[| p pattern |]$, retorna un valor de tipo $Q\ Pat$
- $[| d decl-list |]$, retorna un valor de tipo $Q\ [Dec]$
- $[| t type |]$, retorna un valor de tipo $Q\ Type$

Ejemplo: $[| \backslash x \rightarrow x |]$ se traducirá a

```
(do id ← newName "x"; return $ LamE [VarP id] (VarE id))
```

Definición de parsers. C++/Boost::Spirit

EBNF

```
group ::= '(' expression ')'
factor ::= integer | group
term ::= factor (('*' factor) | ('/' factor))*
expression ::= term (('+' term) | ('-' term))*
```

EBNF en Boost::Spirit

```
group      = '(' >> expression >> ')';
factor     = integer | group;
term       = factor >> * (('*' >> factor)
                        | ('/' >> factor));
expression = term >> * (('+' >> term)
                        | ('-' >> term));
```

Programación funcional en C++

Expresiones λ en boost::lambda

$\lambda x.\lambda y.x + y \equiv _1 + _2$

```
list<int> v(10);  
for_each(v.begin(), v.end(), _1 = 1);  
sort(vp.begin(), vp.end(), *_1 > *_2);
```

FP++

```
List<int> integers = enumFrom(1);  
List<int> evens = filter(even, integers);
```

Programación lógica en C++:lc++

```

FUN1( male, string )
DECLARE( Kid, string ,2);
DECLARE( Par, string ,3);
...
lassert( male(bart) );
lassert( male(homer) );
lassert( female(lisa) );
lassert( parent(homer, bart) );
lassert( father(Dad, Kid) == parent(Dad, Kid) &&
        male(Dad) );
List<IE> l = lquery( ancestor(Anc, bart ,X) );
while( !null(l) ) {
    IE env = head(l);
    env->show();
    l = tail(l);
}

```

Gramáticas de atributos. C++::ag++

```
symbol(Expr) {
    int val;
    char type;
};
symbol(Plus) {};
symbol(Times) {};
rule r1 = Expr<0> >> (Expr<1>,Plus ,Expr<2>).
    compute(
        Expr<0>.val = Expr<1>.val + Expr<2>.val ,
        Expr<0>.type = '+'
    );
rule r2 = Expr<0> >> (Expr<1>,Times ,Expr<2>).
    compute(
        Expr<0>.val = Expr<1>.val * Expr<2>.val ,
        Expr<0>.type = '*'
    ); ...
```

Programación orientada a lenguajes

- Solución a un problema: creación de un **lenguaje** isomórfico a descripciones de usuarios
- Requerimientos:
 - Ambientes de meta-programación
 - Lenguajes de programación extensibles
- Paradigmas relacionados
 - Programación orientada a aspectos
 - Programación basada en conceptos
 - Lenguajes de dominios específicos
 - Programación intensional
 - Programación orientada a gramáticas

Extensibilidad en lenguajes de programación

Lenguaje extensible

Permite al programador la definición de sus propias extensiones sintácticas y semánticas.

Requisitos: Parser extensible, exposición y manipulación del AST durante la compilación

- Seed7
- OpenC++
- Scala
- xtc (eXTensible C)
- XL (eXtensible Language)

Metaprogramación

Ventajas

- Independencia de herramientas externas
- Integración de diferentes paradigmas
- Eficiencia (código o datos generados en compilación)
- Optimizaciones lógicas a nivel de aplicación
- Interacción con el ambiente de compilación

Desventajas

- Requiere soporte del compilador
- Algunos mecanismos inducen programas poco legibles
- Mayor tiempo de compilación

Metaprogramación

Ventajas

- Independencia de herramientas externas
- Integración de diferentes paradigmas
- Eficiencia (código o datos generados en compilación)
- Optimizaciones lógicas a nivel de aplicación
- Interacción con el ambiente de compilación

Desventajas

- Requiere soporte del compilador
- Algunos mecanismos inducen programas poco legibles
- Mayor tiempo de compilación

Gracias!!!

¿Preguntas?

Gracias!!!

¿Preguntas?