Improving Flexibility and Efficiency in Programming Languages

a Natural Approach

PhD thesis, 2015

Lucian Radu Teodorescu

Sparrow programming language

efficiency, flexibility, naturalness

inspired from C++

genealogy

Lisp[32]
ML
SML[21]
Caml
OCaml
F#[14]
SASL
Miranda
Haskell[46]
Scala[41]
Scheme[38]
C Lisp
JavaScript[6]
Erlang
Sparrow
Prolog[42]
Java[2]
Smalltalk
D[30]
Algol
Simula
C++[3]
C#[5]
Basic
Visual Basic[9]
Fortran[31]
Objective-C[4]
Ruby[20]
CPL
C[1]
Go[43]
BCPL, B
Perl[12]
PHP[7]
Ada[35]
Python[8]
COBOL[22]
Pascal[19]
Delphi[11]
1956
1960
1970
1980
1990
2000
2010

Naturalness
meta
programming
Flexibility
Efficiency

the recipe

simple language constructs

+

growth mechanism
(static metaprogramming)

powerful, extensible language

Sparrow compiler

traditional compiler

Sparrow compiler

Sparrow compiler — demo

see “speak.mp4” video

embedding languages
operators
generic programming
efficiency
hyper-metaprogramming
move computations to compile-time

Flexible operators

no predefined operators

not even: **+**, *****, **/**, **=**

define
custom operators

++complex

v1 + v2

v1 cross v2

v1 length

`length` v1

1 .. 100

can be piped

prefix	infix	postfix
- vec	a ** exp	it ++
++ it	1 .. n	1 ..
`norm` vec	r map \f	r sum
	r filter \pred	

more examples

```
1..10                    // 1 2 3 4 5 6 7 8 9
1..10 ../ 2              // 1 3 5 7 9
1..10 sum                // 45
1..n filter \isOdd map (fun n = n*n)
obj method               // obj.method()
```

user defined
type reference

@Int

```
fun[ct] pre_@ (t: Type): Type;
```

precedence and associativity

```
setOperPrecedence("*", 550);
setOperPrecedence("**", getOperPrecedence("*") + 1);
setOperRightAssociativity("**");
```

	C++	Java	Scala	Haskell	Sparrow
Overloading existing operators	●	○	●	●	●
Prefix op. with custom symbols	○	○	○	○	●
Postfix op. with custom symbols	○	○	●	●	●
Infix op. with custom symbols	○	○	●	●	●
Prefix op. with identifier	○	○	○	○	●
Postfix op. with identifier	○	○	●	○	●
Infix op. with identifier	○	○	●	●	●
Custom precedence and associativity	○	○	○	●	●

Conclusions

Sparrow operators

defined in the library

any combination of symbols + identifiers

can be piped

custom precedence and associativity

flexible and natural

Generic programming

generic programming

~ templates, the easy way ~

```
class Vector(valueType: Type)
{ ... }
```

parameters like for functions

```
fun swap(a, b: @AnyType)
{
    var tmp = a;
    a = b;
    b = tmp;
}
```

a single set of parameters

```
fun swap(a, b: @AnyType) if isValid( a.swap(b) )
{
    a.swap(b);
}
```

adding conditions is straightforward

```
fun[rtct] find(range: AnyType, value: @AnyType): typeOf(range)
{
    while ( !range.isEmpty() && !(range.front() == value) )
        range.popFront();
    return range;
}
```

dependent result types

concepts

~ generalisation of types ~

“ a concept is a **predicate** that expresses a set of requirements **on types** ”

B. Stroustrup, A. Sutton

```
concept Range(x)
    if typeOf(x.RetType) == Type
    && typeOf(x.isEmpty()) == Bool
    && typeOf(x.front()) == x.RetType
    && isValid(x.popFront())
    ;
```

just a predicate

```
fun[rtct] find(range: Range, value: @AnyType): typeOf(range)
{
    while ( !range.isEmpty() && !(range.front() == value) )
        range.popFront();
    return range;
}
```

using concepts
as easy as using types

More examples

ranges as iterators

```
fun pre_++ (r: Range): r.RetType {…}
fun post_++ (r: Range): r.RetType {…}
fun pre_-- (r: BidirRange): r.RetType {…}
fun post_-- (r: BidirRange): r.RetType {…}

fun pre_! (r: @Range) = r.isEmpty();
fun pre_!! (r: @Range) = !r.isEmpty();
fun pre_* (r: @Range): r.RetType = r.front();
```

range functions

```
fun rangeSize(range: Range): SizeType {…}
fun rangeSize(range: RandomAccessRange): SizeType = range.size();

fun find(r: Range, value: @AnyType): typeOf(r) {…}
fun copy(r1, r2: Range): typeOf(r2) if isValid(*r2 = *r1) {…}
fun retro(range: BidirRange) = …
fun repeat(value: AnyType) = …
fun repeat(value: AnyType, count: SizeType) = …
```

numeric ranges

```
fun numericRange(start, end, step: Number) = …
fun numericRange(start, end: Number) = …

fun .. (start, end: Number) = …                    // [start, end)
fun ... (start, end: Number) = …                   // [start, end]
fun post_.. (start: Number) = …                    // [start, ∞)
fun ../ (range: Range, step: Number) = …           // set the step
```

functional style

```
fun filter(r: Range, pred: AnyType) = …

fun map(r: Range, pred: AnyType) = …
```

	C++	SML	Haskell	Java	C#	$\mathcal{G}$	Sparrow
Multi-type concepts	○	●	●	○	○	●	○
Multiple constraints	○	◐	●	●	●	●	●
Associated type access	●	●	◐	◐	◐	●	●
Constraints on assoc. types	○	●	●	◐	◐	●	●
Retroactive modeling	○	●	●	○	○	●	○
Type aliases	●	●	●	○	○	●	●
Separate compilation	○	●	●	●	●	●	○
Implicit instantiation	●	○	●	●	●	●	●
Concept dispatching	●	○	○	○	○	◐	●
No run-time overhead	●	○	○	○	○	○	●
Constraints on type usage	◐	○	○	○	○	○	●
Spec. on user constants	◐	○	○	○	○	○	●
Arbitrarily complex cond.	◐	○	○	○	○	○	●

Conclusions

generics and concepts

simple design

easy to use

powerful and flexible

fast

General efficiency

1 for structures

C++

```
for ( int i = 0; i<100; ++i )
{
    cout << i << endl;
}
```

Sparrow

```
for ( i = 0..100 )
{
    cout << i << endl;
}
```

C for
C++ for
C++ ranges
C++ while
Sparrow while
Sparrow for
0.2
1.2
2.34
0.2
1.2
2.34
0.21
1.2
2.34
0.2
1.2
2.34
0.2
1.2
2.34
0.2
1.2
2.34
20000
50000
70000
0
0.5
1
1.5
2
2.5
3
execution time (seconds)

2

Compute the **sum of squares** of all the **odd** numbers belonging to the **first *N*** Fibonacci numbers.

```
fun fib(n: UInt): UInt {
    var p: UInt*UInt = 0 ~ 1;   // Last 2 entries
    for ( i = 0..n )
        p = p._2 ~ (p._1 + p._2);
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

```
fun fib(n: UInt): UInt {
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

```
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

first ’n’ natural numbers, starting with 1

```
fun fib(n: UInt): UInt {
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

```
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

get corresponding Fibonacci numbers

```
fun fib(n: UInt): UInt {
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

```
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

retain only odd numbers

```
fun fib(n: UInt): UInt {
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

get their square

```
fun fib(n: UInt): UInt {
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

```
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

sum the squares

```
fun fib(n: UInt): UInt {
```

Compute the sum of squares
of all the odd numbers belonging to the
first N Fibonacci numbers.

```
    return p._2;
}
fun isOdd x = x%2 != 0;
fun sqr n = n*n;
fun sum(r: Range) = foldLeft(r, (fun x,y = x+y), r.RetType(0));

var res = 1...n map \fib filter \isOdd map \sqr sum;
```

shorter than plain English; uses only problem terms

C++

```
int fib(int n) {
    int a = 0;
    int b = 1;
    for ( int i=0; i<n; ++i ) {
        int aOld = a;
        a = b;
        b += aOld;
    }
    return b;
}

int res = 0;
for ( int i=1; i<=n; ++i ) {
    int f = fib(i);
    if ( f % 2 != 0 ) {
        res += f*f;
    }
}
```

not using
problem terms

n=70000
C
C++
Sparrow classic
Sparrow ranges
1.45
1.46
1.45
1.45
0
0.5
1
1.5
2
execution time (seconds)

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

Collatz sequence

if the number is even, divide it by two;
otherwise multiply it by three and add 1

Ex: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
            takeWhile (fun s = s < 500) rootMeanSquare;
```

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
          takeWhile (fun s = s < 500) rootMeanSquare;
```

all Collatz sequences

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
            takeWhile (fun s = s < 500) rootMeanSquare;
```

get their lenghts

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
          takeWhile (fun s = s < 500) rootMeanSquare;
```

take while the length is less than 500

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
            takeWhile (fun s = s < 500) rootMeanSquare;
```

compute root-mean-square

What is the root mean square of the lengths of all the Collatz sequences up the first one that has a length greater than or equal to 500?

```
fun nextCollatz n = ife(n%2==0, n/2, n*3+1);
fun collatzSeq(n: ULong) = generate1(n, \nextCollatz) takeUntil
(fun n = n==1);

var res = (1..) map \collatzSeq map \rangeSize
          takeWhile (fun s = s < 500) rootMeanSquare;
```

concise & natural

C++

```
unsigned long long nextCollatz(unsigned long long n) {
    return n%2==0 ? n/2 : n*3+1;
}
unsigned int collatzLen(unsigned long long n) {
    unsigned int len = 1;
    while ( n > 1 ) {
        n = nextCollatz(n);
        ++len;
    }
    return len;
}

unsigned long long sum = 0;
int count = 0;
for ( unsigned int i=1; ; ++i ) {
    unsigned int len = collatzLen(i);
    if ( len >= n )
        break;
    // Main part of computing the root mean square
    sum += len*len;
    ++count;
}
double res = sqrt(double(sum) / double(count));
```

not using problem terms

n=500
C
C++
Sparrow classic
Sparrow ranges
2.33
2.33
2.28
2.11
0
0.5
1
1.5
2
2.5
execution time (seconds)

Havlak loop recognition algorithm

C++ / Go / Java / Scala / Sparrow

same algorithm

heavily uses standard library

Robert Hundt. “Loop recognition in C++/Java/Go/Scala”. In: Proceedings of Scala Days. 2011.

Sparrow is the fastest

		C++	Go	Java	Scala	Sparrow	
Code size	LOC	491	531	590	370	346	1
	norm	1.33 x	1.44 x	1.59 x	1.00 x	0.94 x	
Binary size	bytes	25 248	1 955 040	13 248	48 246	21 416	2
	norm	1.00 x	77.43 x	0.52 x	1.91 x	0.84 x	
Memory footprint	MB	165	450	983	699	195	2
	norm	1.00 x	2.73 x	5.96 x	4.24 x	1.18 x	
Compilation time	sec	1.201	0.223	1.427	5.375	3.916	4
	norm	1.00 x	0.19 x	1.19 x	4.47 x	3.26 x	

Conclusions

Sparrow general efficiency

as fast as C++

higher-level algorithms

easier to write code

Hyper-metaprogramming

“ **Metaprogramming** is the writing of programs that write or manipulate other programs or themselves ”

run-time

compile-time

metaprogramming in C++

different syntax

different programming paradigm

limitations abound

slow

hyper-metaprogramming

~ compile-time metaprogramming on steroids ~

hyper-metaprogramming

same syntax

same semantics

share same code

the same code for both run-time and compile-time

```
fun[autoCt] fact(n: Int): Int
{
    if ( n == 0 )    return 1;
    else             return n * fact(n-1);
}

fun testFact
{
    var n = 5;                  // run-time variable
    writeLn( fact(n) );         // Factorial at run-time
    writeLn( fact(5) );         // 120
}
```

hyper-metaprogramming

as easy as traditional programming

N-Queens benchmark

Sparrow

vs.

C++ MPL

vs.

C++ constexpr

Denis V Dubrov. “N Queens Problem: a Metaprogramming Stress Test for the Compiler”. In: Proceedings of the 9th ACM SIGPLAN workshop on Generic programming. 2013.

```
Sparrow Compiler v0.9.2, (c) 2014

NQueens.spr
We have 2 solutions
```

```
Linking...

Time elapsed: 0.351s
```

no more boring
compiler output

1000
100
10
1
time (seconds)
hyper-metaprogramming
C++ with MPL
C++ with constexpr
4
6
8
10
12
N

Sparrow implementation

natural (as written for run-time)
flexible (no limitations as other versions)
compile-time efficient (much faster than others)

compared to run-time?

hyper-metaprogramming
run-time interpreted
10
1
time (seconds)
4
6
8
10
12
N

Conclusions

hyper-metaprogramming

easy to use — same syntax and semantics

flexible — execute anything

fast to execute — as run-time code

Moving computations to compile-time

a typical program
can have
a static and a dynamic part

execute the static part at compile-time

$$su = \frac{1}{1 - \frac{ct}{p} + \frac{tr}{p}} \approx \boxed{\frac{1}{1 - \frac{ct}{p}}}$$

how big can the speedup be?

factorial warm-up

```
fun[autoCt] fact(n: Int): Int
{
    if ( n == 0 )    return 1;
    else             return n * fact(n-1);
}

fun testFact
{
    var n = 5;
    writeLn( fact(n) );      // Factorial at run-time
    writeLn( fact(5) );      // 120
}
```

```
define void @testFact()
{
    %n = alloca i32
    call void @Int_ctor_Int(i32* %n, i32 5)
    %1 = load i32* %n
    %2 = call i32 @fact(i32 %1)
    call void @writeLnInt(i32 %2)
    call void @writeLnInt(i32 120)          ; <-- look here
    call void @Int_dtor(i32* %n)
    ret void
}
```

generated LLVM code

1 regular expressions

if (regexMatch(“(a|b)*a(a|b)z”, testString)) …

static

dynamic

```
class[rtct] RegexAutomaton {
    fun ctor( re: String );
    fun[rt] ctorFromCt( src: RegexAutomaton ct );
    fun match( str: String ): Bool;
}

fun regexMatch( re: String, str: String ): Bool {
    var aut = RegexAutomaton(re);
    return aut.match(str);
}

fun regexMatch( re: String ct, str: String ): Bool {
    var[ct] autCt = RegexAutomaton(re);  // compile-time
    var aut = autCt;                     // tr part (ctorFromCt)
    return aut.match(str);               // run-time
}
```

add a compile-time specialization

C-DFA
Sparrow RT
Sparrow CT
execution time (μs)
0
20
40
60
80
5
10
15
20
25
n

$a?^n a^n$

C-DFA
Sparrow RT
Sparrow CT
execution time (μs)
0
50
100
150
5
10
15
20
25
n

$(a|b)?^n(a|b)^n z$

C-DFA
Sparrow RT
Sparrow CT
execution time (μs)
400
200
0
5
10
15
20
25
n

$$(a|b)?^{2n}(a|b)^n z$$

$a?^n a^n$
$(a|b)?^n (a|b)^n z$
$(a|b)?^{2n} (a|b)^n z$
speedup
n
0
5
10
15
5
10
15
20
25

up to 12 times faster

without changing client code

2 minimal perfect hashing

```
class[rtct] MinPerfHash {
    fun ctor( keys: @Vector(String) );
    fun[rt] ctorFromCt( src: MinPerfHash ct );
    fun search( key: String ): UInt;
}

fun mph( keys: @Vector(String) ): MinPerfHash {
    var hash = MinPerfHash(keys);
    return hash;
}

fun mph( keys: @Vector(String) ct ): MinPerfHash {
    var[ct] hashCt = MinPerfHash(keys);  // compile-time
    var hash = hashCt;                   // tr part (ctorFromCt)
    return hash;
}
```

add a compile-time specialization

CMPH
Sparrow RT
Sparrow CT
execution time (seconds)
1
0.5
0
0
50,000
100,000
150,000
200,000
250,000
number of keys

80 times faster

Conclusions

moving computations to compile-time

easy to do with hyper-metaprogramming

user convenience not compromised

can work for real-world examples

significant speedups

Embedding other languages

the problem

one size doesn’t fit all

solution

embedding other languages into Sparrow

Domain-Specific Embedded Languages

easier to write solutions to sub-problems

easier to reason about

common practice

examples

regular expressions

grammar specification

Prolog?

use the appropriate paradigm

embedding example

embed Prolog in Sparrow

as a library feature

no syntax change

yet, use Prolog syntax

```
children(sam, mary).
children(denise, mary).
children(sam, frank).
children(denise, frank).
children(frank, garry).

parent(P, C) :- children(C, P).
father(F, C) :- children(C, F), gender(F, male).
mother(M, C) :- children(C, M), gender(M, female).

ancestor(A, C) :- children(C, A).
ancestor(A, C) :- children(C, P), ancestor(A, P).
```

what we want to write

first, implement it in Sparrow

```
fun children(child, parent: @LStr)
    = child /=/ "Sam"      && parent /=/ "Mary"
    || child /=/ “Denise"   && parent /=/ "Mary"
    || child /=/ "Sam"      && parent /=/ "Frank"
    || child /=/ "Denise"   && parent /=/ "Frank"
    || child /=/ "Frank"    && parent /=/ “Gary" ;

fun parent(parent, child: @LStr) = children(child, parent);
fun ancestor(a, c: @LStr): Predicate {
    var parent: LStr;
    return children(c, a)
            || children(c, parent) && rec(ancestor, a, parent);
}
```

then, use hyper-metaprogramming

implement Prolog parser

implement semantic analyzer

implement code generation

handle errors

```
compileProlog(“
    children(sam, mary).
    children(denise, mary).
    children(sam, frank).
    children(denise, frank).
    children(frank, garry).

    parent(P, C) :- children(C, P).
    father(F, C) :- children(C, F), gender(F, male).
    mother(M, C) :- children(C, M), gender(M, female).

    ancestor(A, C) :- children(C, A).
    ancestor(A, C) :- children(C, P), ancestor(A, P).
“);
```

the result

```
compileProlog(“
        fib(0,0).
        fib(1,1).
        fib(X,Y) :-
                X2 is X-2, fib(X2, Y2),
                X1 is X-1, fib(X1, Y1),
                Y is Y1+Y2.
“);
```

the result

error handling

```
Fibonacci.pl(7:5-8) : ERROR : Type mismatch: cannot combine atom with
number
> fib(sun, earth).
      ~~~
Fibonacci.pl(7:5-8) : See also atom expression:
> fib(sun, earth).
      ~~~
Fibonacci.pl(5:11-14) : See also number expression:
>     X1 is X-1, fib(X1, Y1),
            ~~~
```

natural error handling

import your own files

import “parents.pl";

holy grail of DSLs

import your own files

```
var[ct] regOk = registerFrontendFun(".pl", "handlePrologCode");
```

implementation

efficiency?

first take,
unoptimized Sparrow

try going fully native

whenever possible

efficiency

as fast as native C++

Conclusions

embedding other languages

entierly as a library feature

flexible mechanism

allows adding new paradigms

high naturalness

fast

General conclusions

Sparrow

simple design choices

very flexible

improved naturalness

allows high-level constructs

Sparrow

as efficient as C++

suitable for moving computations to CT

hyper-metaprogramming is usable, and fast

Naturalness
Sparrow
Flexibility
Efficiency

theoretical contributions

1. A new method of constructing programming languages that can integrate flexibility, efficiency and naturalness.

2. A theoretical framework for constructing hyper-metaprogramming, an innovative form of compile-time metaprogramming that is easy to use, flexible and fast.

3. By making hyper-metaprogramming a central feature of the Sparrow language, we create a novel technique of constructing programming languages. With this, we also change the status of compile-time metaprogramming from an esoteric feature to a potentially fundamental one.

4. An original method of improving execution times by moving parts of the computations to compile-time. We described both a theoretical framework and some applications.

applicative contributions (1)

1. A new programming language, Sparrow, that integrates flexibility, efficiency and naturalness. Sparrow is at least as efficient as C++, more flexible and more natural than most of the mature programming languages (e.g., C++, Scala, Java).

2. Advances in the way a programming language can define operators. We proposed a method of constructing an operators system that is more flexible than those found in existing programming languages, while still being natural and fast.

3. Contributions in the field of generic programming. We devised a method of constructing a generic programming model in programming languages that is simple, easy to use, efficient and powerful.

4. Proposed an innovative method of constructing concepts that is both simple and powerful.

5. Contributions to the way one can represent efficient computations in a natural manner. Our method is based on generic programming and a library feature called *ranges*, and lets the user implement problems in simple expressions, using the terms found in the problem description, while maintaining the same efficiency as C++.

applicative contributions (2)

6. An original method of improving execution times by moving parts of the computations to compile-time. We described both a theoretical framework and some applications. The resulting speedups are impressive: more than 12x for regular expressions and 80x for minimal perfect hashing.

7. A comprehensive analysis of the costs implied by hyper-metaprogramming. Our results show that executing algorithms at compile-time is easy and fast; almost as fast as executing them at run-time.

8. A novel technique of embedding domain-specific programming languages into a host programming language, using hyper-metaprogramming. This technique allows the implementation to be done purely at library level, without changing the compiler of the host language; at the same time the user will use the syntax of the embedded language, and the integration is seamless (e.g., homogeneous error reporting, anti-quotation for interacting between languages). The generated code is also efficient.

9. Allowing the user to extend a language with new domain-specific languages and with new programming paradigms also constitutes a major accomplishment in increasing the naturalness of programming languages; the user is able to express computations in the paradigm or language that best fits the problem.

future work

micro-compiler
more functional Sparrow
concept-oriented programming
Sparrow 2.0

Thank you

https://github.com/Sparrow-lang/sparrow