# Generative Programming:
## from High-Performance to Safety-Critical Applications
### LMS-Verify:
### Abstraction Without Regret for Verified Systems Programming

Nada Amin & Tiark Rompf

EPFL & Purdue University

October 31, Mobile!2016

# LMS = Lightweight Modular Staging

- Int, String, T
  *"execute now"*

- Rep[Int], Rep[String], Rep[T]
  *"generate code to execute later"*

- if (c) a else b → __ifThenElse(c,a,b)
  *"language virtualization"*

- extensible IR, dead-code elimination, transformers, loop fusion, . . .
  *"batteries included"*

# (David Wheeler's Aphorism)

"Any problem in computer science
can be solved by
another level of indirection
– except problems caused by
too many levels of indirection."

# Staging

- Use
  Indirection
  to solve
  Performance Problems,
  too

# LMS Projects / Collaborations

- Delite (Stanford)
  DSLs and Big Data on heterogeneous devices
- Spiral (ETH)
  Fast numeric libraries
- LegoBase (EPFL DATA)
  Database and query processing
- Lancet (Oracle Labs)
  Integrate LMS with JVM/JIT compilation
- Hardware (EPFL PAL)
  Domain-specific HW synthesis
- Parser Combinators (EPFL LAMP)
  Protocols and dynamic programming

# Data Transfer, Validation, Parsing

- Distributed devices, clients & servers, for various platforms such as mobile & internet of things, need to **communicate**.
- Protocol **messages** need to be transferred and parsed.
- Messages can be **invalid** and even malicious, causing bugs and vulnerabilities through unexpected behavior.

- HTTP parser staging and comparison with Nginx based on Jonnalagedda et al. OOPSLA'14.

# Efficient, hand-optimized HTTP parser

```
switch (s) {
  case s_req_space_before_url:
    if (ch == '/' || ch == '*') {
      return s_req_path;
    }
    if (IS_ALPHA(ch)) {
      return s_req_schema;
    }
    break;

  case s_req_schema:
    if (IS_ALPHA(ch)) {
      return s;
    }
    if (ch == ':') {
      return s_req_schema_slash;
    }
    break; // ...
```

- Originally part of Nginx, later Node.js
- 2000+ lines of code
- Callbacks for header names/values triggered
- State-machine like code
- "Flat" code, loops/conditions

# Staged Parser Combinators

```
def status: Parser[Int] =
  ("HTTP/"~decimalNumber)~>wholeNumber<~(wildRegex~crlf) ^^
  (_.toInt)
def header: Parser[Option[(String,Any)]] =
  (headerName<~":")~(wildRegex<~crlf) ^^ {
    case hName~prop => collect(hName.toLowerCase, prop)
  }
def headers = rep(header)
def response = status ~ headers
```

- ▶ 200+ lines of code
- ▶ Fairly easy to change behaviour of a parser
  e.g. ~ vs. <~ vs. ~>
- ▶ Generates low-level Scala code that is fairly competitive with Nginx.
- ▶ Can also generate C code like Nginx.

# Performance: Success! What about Security?

# Security Advisories

- Nginx: CVE-2013-2028
  nginx 1.3.9 through 1.4.0 allows remote attackers to cause a denial of service (crash) and execute arbitrary code via a **chunked Transfer-Encoding** request with a large chunk size, which triggers an integer signedness error and a stack-based buffer overflow.

- Apache: CVE-2002-0392
  Apache 1.3 through 1.3.24, and Apache 2.0 through 2.0.36, allows remote attackers to cause a denial of service and possibly execute arbitrary code via a **chunk-encoded HTTP** request that causes Apache to use an incorrect size.

# Generative
# Meta
# Programming
# & Verification

`github.com/namin/lms-verify`

# Porting Staged HTTP Parser to LMS-Verify

```
type Input = Array[Char] // \0-terminated C string
def valid_input(s: Rep[Input]) =
  s.length>=0 && valid(s, 0 until s.length+1)
```

# Porting Staged HTTP Parser to LMS-Verify

| parser | requests per second |
|---|---|
| (baseline) nginx | $(0.94 \pm 0.01) \cdot 10^6$ |
| (our) staged verified | $(1.00 \pm 0.01) \cdot 10^6$ |

| module / instance | lang. | LoP | LoC |
|---|---|---|---|
| **HTTP Parser** | Scala | 2 | 118 |
| staged parser combinator lib. | Scala | 5 | 197 |
| *verified memory & overflow safe* | | | |
| without chunking | C | 95 | 1517 |
| with chunking | C | 133 | 2630 |

# Verifying HTTP Parser: Parametrized Loop Annotations

```scala
def rep[T: Typ, R: Typ](p: Parser[T], z: Rep[R],
  f: (Rep[R], Rep[T]) => Rep[R],
  pz: Option[Rep[R] => Rep[Boolean]] = None) =
Parser[R] { input =>
  var in = input
  var c = unit(true); var a = z
  while (c) {
    loop_invariant(valid_input(in) &&
      (pz.map(_(a)).getOrElse(true)))
    loop_assigns(in, c, a)
    p(in).apply[Unit](
      (x, next) => { a = f(a, x); in = next },
      next => { c = false })
  }
  ParseResultCPS.Success(a, in)
}
```

# Verifying HTTP Parser: Overflow Handling

```
def num(c: Parser[Int], b: Int): Parser[Int] =
  c >> { z: Rep[Int] =>
    rep(c, z, { (a: Rep[Int], x: Rep[Int]) => a*b+x })
  }
// vs
def num(c: Parser[Int], b: Int): Parser[Int] =
  c >> { z: Rep[Int] =>
    rep(c, z, { (a: Rep[Int], x: Rep[Int]) =>
      if (a<0) a
      else if (a>Int.MaxValue / b - b) OVERFLOW
      else a*b+x
    }, overflowOrPos)
  }
```

# Generic, Polytypic, Checked Prog. with Type Classes

```scala
trait Eq[T] { def eq(a: T, b: T): Rep[Boolean] }
def equality[T](f: (T, T) => Rep[Boolean]) = new Eq[T] {
  def eq(a: T, b: T) = f(a,b)
}
implicit class EqOps[T:Eq](a: T) {
  def deep_equal(b: T): Rep[Boolean] =
    implicitly[Eq[T]].eq(a,b)
}

trait Iso[T] {
  def id: String
  def toRepList(x:T): List[Rep[_]]
  def fromRepList(xs: List[Rep[_]]): T
}
trait Inv[T] {
  def valid(x:T): Rep[Boolean]
}
```

```
class Vec[T:Iso](val a: Pointer[T], val n: Rep[Int]) {
  def apply(i: Rep[Int]) = a(i)
  def valid = n==0 || (n>0 && a.valid(0 until n))
  def length = n
}
implicit def vecIso[T:Iso](implicit ev: Inv[Vec[T]]) =
  explode_struct("vec_" + key[T],
  {x: Vec[T] => (x.a, x.n)},
  {x: (Pointer[T],Rep[Int]) => new Vec(x._1, x._2)}
)
implicit def vecInv[T:Inv] = invariant[Vec[T]] { x =>
  x.valid && ((0 until x.n) forall {i => x(i).valid})
}
implicit def vecEq[T:Eq:Iso] = equality[Vec[T]] { (x, y) =>
  x.n == y.n && ((0 until x.n) forall {i =>
    x(i) deep_equal y(i)
  })
}
```

# Instantiate for specific type

```
implicitly[Eq[Vec[Vec[Rep[Int]]]]]
```

# Generated ACSL + C for Vector of Integers

```
/*@
predicate inv_vec_Int(int* a, int n) =
  (n==0) || ((n>0) && \valid(a+(0..n-1)));
predicate eq_vec_Int(int* a1, int n1, int* a2, int n2) =
  ((n1==n2) && (\forall int i; (0<=i<n1) ==> (a1[i]==a2[i]))); */

/*@ assigns \nothing;
requires (inv_vec_Int(a1,n1) && inv_vec_Int(a2,n2));
ensures \result <==> eq_vec_Int(a1, n1, a2, n2); */
int eq_vec_Int(int* a1, int n1, int* a2, int n2) { ... }
```

```
/*@ assigns \nothing;
requires (inv_vec_Int(a1,n1) && inv_vec_Int(a2,n2));
ensures \result <==> eq_vec_Int(a1, n1, a2, n2); */
int eq_vec_Int(int* a1, int n1, int* a2, int n2) {
  int x23 = n1 == n2;
  int x35;
  if (x23) {
    int x34 = 1;
    /*@
    loop invariant (0 <= i <= n1);
    loop invariant \forall int j;
      (0 <= j < i) ==> (a1[j]==a2[j]);
    loop assigns i;
    loop variant (n1-i); */
    for (int i = 0; i < n1; i++) {
      int x31 = a1[i];
      int x32 = a2[i];
      int x33 = x31 == x32;
      if (!x33) { x34 = 0; break; }
    }
    x35 = x34;
  } else { x35 = 0/*false*/; }
  return x35;
}
```

# Generated for Vector of Vector of Integers

```
/*@
predicate inv_vec_vec_Int(int** a, int* an, int n) = (((n==0) || ((n>0) &&
  (\valid(a+(0..n-1)) && \valid(an+(0..n-1)))))) &&
  (\forall int i; (0<=i<n) ==> inv_vec_Int(a[i],an[i])));
predicate eq_vec_vec_Int(int** a1, int* an1, int  n1,
  int** a2, int* an2, int n2) = ((n1==n2) && (\forall int i; (0<=i<n1) ==>
  eq_vec_Int(a1[i],an1[i],a2[i],an2[i]))); */

/*@ assigns \nothing;
requires (inv_vec_vec_Int(a1,an1,n1) && inv_vec_vec_Int(a2,an2,n2));
ensures \result <==> eq_vec_vec_Int(a1, an1, n1, a2, an2, n2); */
int eq_vec_vec_Int(int** a1, int* an1, int n1, int** a2, int* an2, int n2)
{ ... }
```

```
/*@ assigns \nothing;
requires (inv_vec_vec_Int(a1,an1,n1) && inv_vec_vec_Int(a2,an2,n2));
ensures \result <==> eq_vec_vec_Int(a1, an1, n1, a2, an2, n2); */
int eq_vec_vec_Int(int** a1, int* an1, int n1, int** a2, int* an2, int n2)
{ int x72 = n1 == n2;
  int x88;
  if (x72) {
    int x87 = 1;
    /*@
    loop invariant (0 <= i <= n1);
    loop invariant \forall int j; (0 <= j < i) ==>
      eq_vec_Int(a1[j],an1[j],a2[j],an2[j]);
    loop assigns i;
    loop variant (n1-i); */
    for (int i = 0; i < n1; i++) {
      int *x82 = a1[i]; int x83 = an1[i];
      int *x84 = a2[i]; int x85 = an2[i];
      int x86 = eq_vec_Int(x82,x83,x84,x85);
      if (!x86) { x87 = 0; break; }
    }
    x88 = x87;
  } else { x88 = 0/*false*/; }
  return x88;
}
```

# Representation of Vectors in Low-Level Code

**vector of integers**
`int*`   element array
`int`    length
**vector of vector of integers**
`int**`  element array by outer then inner index
`int*`   array for inner length by outer index
`int`    length (of outer)

# Contracts

```
val inswap = fundef {
  (p: Rep[Array[Int]], i: Rep[Int], j: Rep[Int]) =>
  // pre-condition:
  requires{valid(p, i) && valid(p, j)}
  // post-condition:
  ensures{res => p(i)==old(p(j)) && p(j)==old(p(i))}
  // body code:
  val tmp = p(i)
  p(i) = p(j)
  p(j) = tmp
}
```

# Code and Specification Target from Shared Source

multi-target expression:

```
(0 until n).forall {i => ...}
```

used in spec:

```
// \forall i; 0 <= i && i < n ==> ...
```

used in code:

```
for (int i = 0; i < n; i++) { ... }
```

# Multi-Target Staging-Time Abstractions $\implies$ Domain-Specific Parametric Logic

```
fundef("add", { (a: Matrix[X], b: Matrix[X], o: Matrix[X]) =>
  o.setFrom2({ (ai: X, bi: X) => ai + bi }, a, b)
})
fundef("scalar_mult", { (a: X, b: Matrix[X], o: Matrix[X]) =>
  o.setFrom1({ (bi: X) => a*bi }, b)
  // easy to prove, thanks to annotations added by setFrom
  ensures{result: Rep[Unit] => (a == zero) ==>
    (0 until o.rows).forall{r => (0 until o.cols).forall{c =>
      o(r,c) == zero }}}})
```

## Encapsulate Verification Properties

```scala
def setFrom[A:Iso](f: List[A] => T, ms: List[Matrix[A]])
  (implicit eq: Eq[T]) = {
  def r(i: Rep[Int]): T = f(ms.map{m => m.a(i)})
  def p(n: Rep[Int]): Rep[Boolean] = forall{j: Rep[Int] =>
    (0 <= j && j < n) ==> (this.a(j) deep_equal r(j)) }
  ms.foreach{ m =>
    requires(this.rows == m.rows && this.cols == m.cols)
  }
  // ... separation requirements ...
  requires(this.mutable)
  for (i <- 0 until this.size) {
    loop_invariant(p(i))
    this.a(i) = r(i)
  }
}
```

# Derive Logic via IR & LMS Effects

```
fundef("mult", {
  (a: Matrix[X], b: Matrix[X], o: Matrix[X]) =>
  requires(a.cols == b.rows &&
    a.rows == o.rows && b.cols == o.cols)
  requires(o.mutable)
  for (r <- 0 until a.rows) {
    for (c <- 0 until b.cols) {
      o((r,c)) = zero
      for (i <- 0 until a.cols) {
        o((r,c)) = o(r,c) + a(r,i) * b(i,c)
}}}})
```

for first loop:
```
  /*@
  loop invariant 0<=r<=a.rows;
  loop assigns r, o.p[(0..(o.rows*o.cols)-1)];
  loop variant n-r;
  */
```

# Naive Staged Regular Expression Matchers

| Regular Expression | Scala | 3 | 41 |
|---|---|---|---|
| *verified memory safe* | | | |
| annotated interpreter | (Hand) C | 32 | 35 |
| ^a | C | 4 | 21 |
| a | C | 10 | 39 |
| a$ | C | 10 | 41 |
| ab.*ab | C | 16 | 155 |
| aa* | C | 16 | 80 |
| aa*bb* | C | 28 | 192 |
| aa*bb* | C | 28 | 192 |
| aa*bb*cc* | C | 52 | 416 |
| aa*bb*cc*dd* | C | 100 | 864 |
| aa*bb*cc*dd*ee* | C | 196 | 1760 |
| aa*bb*cc*dd*ee*ff* | C | 388 | 3552 |
| aa*bb*cc*dd*ee*ff*gg* | C | 772 | 7136 |
| aa*bb*cc*dd*ee*ff*gg*hh* | C | 1540 | 14304 |

# Generic Sorting

| module / instance | lang. | LoP | LoC |
|---|---|---|---|
| **Selection Sort** | Scala | 41 | 115 |
| *verified sorted & in-place permuted* | | | |
| `ints by` $\leq$ | C | 88 | 26 |
| `ints by` $\geq$ | C | 88 | 26 |
| `int pairs by first proj.` | C | 116 | 43 |
| `int pairs by lex.` | C | 130 | 52 |
| `int vectors by length` | C | 128 | 49 |

# Take-Aways

- generative programming extends to verification...
- ... when properties to verify fit the staging-time abstractions
- exploit generative programming patterns
  - type classes as a discipline for structuring genericity
    - parametricity by default
    - explicit functionality
  - well-suited for composing and specializing specifications
- enlarge success story of "abstraction without regret"
  - from high-performance computing
  - to also safety-critical domains
- pleasantly surprised by
  1. how little work is required to go from working to verified
  2. how little work is required to go from one instance verified to many (i.e. up to all wanted) instances verified
- already practical with current tools (e.g. Frama-C)
- lessons may be more generally applicable