LMS-Verify

Abstraction Without Regret for Verified Systems Programming

Nada Amin (joint work with Tiark Rompf)

LAMP, EPFL

June 29, 2016

```
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)

Generic Programming with Type Classes

```
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)
}
```

Polytypic Programming with Type Classes

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

Type Classes for Invariants

```
trait Inv[T] {
  def valid(x:T): Rep[Boolean]
}
```

Interlude: Leon-style 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
}
```

Take Away: Generative Verification with Type Classes

- a discipline for structuring genericity
  - parametricity by default
  - explicit functionality
- well-suited for composing and specializing specifications

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 $\Longrightarrow$ 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

```
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;
  */
```

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

```
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 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

```
def insort[T:Iso:Ord] = fundef("sort"+key[T]) { a: Vec[T] =>
  requires { a.mutable }
  val n = a.length
  for (i <- 0 until (n-1)) {
    loop_invariant(a.slice(0,i).sorted)
    loop_invariant((i > 0) ==>
      (a.slice(i,n).forall(a(i-1) cmp _)))
    var jmin = i
    // could also express inner loop as:
    // a.slice(i,n).minIndex
    for (j <- (i+1) until n) {
      loop_invariant(a.slice(i,j).forall(a(jmin) cmp _)))
      if (a(j) cmp a(jmin)) jmin = j
      else asserts(a(jmin) cmp a(j))
    }
    inswap(a,i,jmin)
  }
  ensures { res => Sorted(a) && Permut("Old","Post")(a) }
}
```

Functional Correctness

```
def insort[T:Iso:Ord] = fundef("sort"+key[T]) { a: Vec[T] =>
  ...
  ensures { res => Sorted(a) && Permut("Old","Post")(a) }
}

def Sorted[T:Iso:Ord](a: Vec[T]) = forall{i: Rep[Int] =>
  (0 <= i && i < a.length-1) ==> (a(i) cmp a(i+1))}
```

Seq. on output is a permutation of the seq. on input.

```
def Swapped[T:Iso:Eq](l1: Lc, l2: Lc)
  (a: Vec[T], i: Rep[Int], j: Rep[Int]) = {
  ((at(a(i),l1)) equal (at(a(j),l2))) &&
  ((at(a(j),l1)) equal (at(a(i),l2))) &&
  forall{k: Rep[Int] =>
    (0 <= k && k < a.length && k != i && k != j) ==>
    ((at(a(k),l1)) equal (at(a(k),l2)))}
}

def Swapped1[T:Iso:Eq](ls: (Lc, Lc))(a: Vec[T]) =
  exists{i: Rep[Int] => exists{j: Rep[Int] => Swapped(ls)(a,i,

def Permut[T:Iso:Eq] =
  reflexiveTransitiveClosure[Vec[T]](Swapped1, as="Permut")
```

```
/*@
inductive Permut_Int{L1,L2}(int* a, integer n) {
case Permut_Int_refl{L}:
  \forall int* a, integer n; Permut_Int{L,L}(a, n) ;
case Permut_Int_trans{L1,L2,L3}:
  \forall int* a, integer n;
  Permut_Int{L1,L2}(a, n) && Permut_Int{L2,L3}(a, n) ==>
  Permut_Int{L1,L3}(a, n) ;
case Permut_Int_step{L1,L2}:
  \forall int* a, integer n;
  (\exists integer i, integer j;
    \at(a[i],L1)==\at(a[j],L2) &&
    \at(a[j],L1)==\at(a[i],L2) &&
   (\forall integer k;
    0 <= k < n && k != i && k != j ==>
    \at(a[k],L1)==\at(a[k],L2))) ==>
  Permut_Int{L1,L2}(a, n) ;
}
*/
```

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

Take-Aways

- generative programming extends to verification...
- ... when properties to verify fit the staging-time abstractions
- exploit generative programming patterns
- 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