LMS-Verify: Abstraction without Regret for Verified Systems Programming

Nada Amin* Tiark Rompf †

*EPFL: {first.last}@epfl.ch
†Purdue University: {first}@purdue.edu

Abstract

Performance critical software is almost always developed in C, as programmers do not trust high-level languages to deliver the same reliable performance. This is bad because low-level code in unsafe languages attracts security vulnerabilities and because development is far less productive, with PL advances mostly lost on programmers operating under tight performance constraints. High-level languages provide memory safety out of the box, but they are deemed too slow and unpredictable for serious system software.

Recent years have seen a surge in staging and generative programming: the key idea is to use high-level languages and their abstraction power as glorified macro systems to compose code fragments in first-order, potentially domain-specific, intermediate languages, from which fast C can be emitted. But what about security? Since the end result is still C code, the safety guarantees of the high-level host language are lost.

In this paper, we extend this generative approach to emit ACSL specifications along with C code. We demonstrate that staging achieves "abstraction without regret" for verification: we show how high-level programming models, in particular higher-order composable contracts from dynamic languages, can be used at generation time to compose and generate first-order specifications that can be statically checked by existing tools. We also show how type classes can automatically attach invariants to data types, reducing the need for repetitive manual annotations.

We evaluate our system on several case studies that varyingly exercise verification of memory safety, overflow safety, and functional correctness. We feature an HTTP parser that is (1) fast (2) high-level: implemented using staged parser combinators (3) secure: with verified memory safety. This result is significant, as input parsing is a key attack vector, and vulnerabilities related to HTTP parsing have been documented in all widely-used web servers.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords contracts, blame, memory safety, Frama-C, LMS, DSLs, verification, security

1. Introduction

How should we build systems that are performant, secure, and correct? Today, systems-level software such as network stacks, databases, and control code in embedded devices is almost always developed in C. This is bad, for at least two reasons. First, development is far less agile and productive than in higher level languages, and second, low-level code in unsafe languages invites security vulnerabilities.

High-level languages, on the other hand, rule out entire classes of vulnerabilities through built-in memory safety guarantees, and many of them provide elaborate mechanisms for programming with contracts, which enforce user-defined specifications at runtime. But while implementations of high-level languages have come a long way, programmers cannot trust them to deliver the same reliable performance as handwritten C code. Contract monitoring at runtime, in particular, can have prohibitive overhead [81, 88]. As a consequence, PL advances are mostly lost on programmers operating under tight performance constraints. Sound static verification of general-purpose C code is possible, but has an extraordinary cost in terms of user annotations, and is thus rarely done in practice. But tools are advancing fast, and formal verification is on the verge of becoming practical for larger classes of software.

Motivated by the apparent trade-off between productivity and performance, researchers have argued [72] for a rethinking of the role of high-level languages in performance critical code, with the goal of allowing developers to leverage high-level programming abstractions without the hefty price in performance. The shift in perspective that enables this vision of "abstraction without regret" is a properly executed form of generative programming: instead of running the whole system in a high-level managed language runtime, the abstraction power of high-level languages can be focused on generating and composing pieces of low-level code.

Generative programming has proven successful for numerical kernels such as FFTs [62], for DSLs in big-data processing [13], but also in domains like database query engines [48], and parsers for communication protocols [44]. In these traditional strongholds of low-level programming, generative programming has largely lived up to its promise: programmers reap the benefits of programming in a high-level language without the low-level language drawbacks.

However, previous work has only looked at productivity benefits, not at security and correctness. Since low-level code is still being generated, the safety guarantees usually associated with high-level languages (e.g. memory safety) do not carry over. And what about functional correctness? We have to trust the high-level code and the generator, either or both of which might contain bugs.

The main contribution of this paper is to show that "abstraction without regret" holds for security and verification as it does for performance. We demonstrate how generative programming frame-

works can be extended to emit low-level code that can be formally verified using existing C-level verification tools. For this, we provide a contract API to specify user-defined assertions as well as pre- and post-conditions. Since we are working in a metaprogramming environment, this facility effectively enables programming with higher-order composable contracts at generation time. But unlike contract systems in dynamic languages, our contracts result in first-order assertions in the generated code. These can be statically checked and never lead to any runtime overhead.

While verification tools for C have come a long way, they still require an inordinate amount of annotations to verify even small programs. Fortunately we can capitalize on the fact that we are constructing program pieces from within a meta-language again. By not generating target code directly but going through one or more domain-specific intermediate languages, we can restrict the programming model and include e.g. effect and aliasing information in the IR. From such an enriched IR, we show that we can generate a large subset of the required annotations such as loop invariants and separation properties automatically. This automation proves vital in reducing the annotation burden for the programmer.

Finally, generative programming does not need to be an all-or-nothing approach. It is easy to specify boundary APIs and generate only critical parts of a system, while other parts are written by hand. This is a key reason why it is attractive to target standardized specification languages such as ACSL (ANSI/ISO C Specification Language) [7].

In short, we show that generative programming is useful not just for performance, but also for verification. In our approach, we generate not just low-level code (e.g. C), but also low-level specs (e.g. ACSL annotations), so that the output can be verified (e.g. to be memory safe) by an independent tool (e.g. Frama-C). We make the following contributions:

- We review how generative programming enables "abstraction without regret" for high-performance and low-level code, stressing the key insights: (1) we can use abstractions such as higher-order functions, objects, type classes, etc. at generation time without incurring any cost at runtime, (2) we can use the meta-language type system to enforce certain desired restrictions on the generated code, in particular that functions and mutable variables behave in a second-class way (Section 2).
- We extend the programming model with higher-order, composable, contracts. Since we *generate* only top-level functions with first-order code, i.e. inlined or named function applications, we are able to (1) statically check the contracts in the low-level code, including the higher-order pieces, and (2) reliably assign "blame" in this generative setting (Section 3).
- We apply generative programming to compose not just code fragments, but also contracts and specifications. We show how to further alleviate the annotation burden by (1) associating invariants to data types via type classes, (2) generating both code and specification logic from the same high-level source, (3) inferring loop and separation properties from the IR, and (4) through custom domain-specific loop constructs (Section 4).
- We evaluate our system on several case studies that exercise memory safety, overflow safety, and functional correctness, including an HTTP parser that is (1) fast, (2) high-level: implemented using staged parser combinators, (3) secure: with verified memory safety. This result is significant, as input parsing is a key attack vector, and vulnerabilities related to HTTP parsing have been found in all widely-used web servers (Section 5).

Section 6 provides a broader perspective, and Section 7 surveys related work. Our evaluation suggests that the usual patterns of generative programming (e.g. generic instantiation, turning an interpreter into a compiler) scale surprisingly well to verification.

```
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:Inv:Iso] =
  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 equality spec + computation for specific type
implicitly[Eq[Vec[Vec[Rep[Int]]]]]
```

Figure 1. Higher-order generic source in Scala, generating low-level annotated code of Figure 2.

```
/*@
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]))); */
/*@
requires (inv_vec_Int(a1,n1) && inv_vec_Int(a2,n2));
assigns \nothing;
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;
}
/*@
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]))); */
/*@
requires (inv_vec_vec_Int(a1,an1,n1) && inv_vec_vec_Int(a2,an2,n2));
assigns \nothing;
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;
}
```

Figure 2. Emitted low-level first-order code in C with specifications in ACSL, generated from high-level generic source of Figure 1, modulo some local variable renaming for clarity.

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)

Figure 3. Representation of vectors in low-level code of Figure 2. Choice here explodes source fields (item pointer, length) into variables, and flattens nested structures akin to "struct of arrays".

As a teaser, consider the source and target code shown in Figures 1 and 2, respectively (it is not necessary to understand it in detail yet; many parts of it will be explained in Section 2). Based on a generic definition of a `Vec` class, we define generic type classes that define data layout, data invariants, and equality predicates. When we instantiate these generic templates to a concrete type, we will emit the corresponding specialized first-order C code – complete with ACSL specifications. Here in the generator, we ask for the equality instance for a vector of vectors of integers. In the generated code, we get the data invariants (as ACSL logic) and the equality predicates (as both C code and ACSL logic), for vector of integers and vector of vectors of integers, where the latter uses the former internally. The specifications gracefully handle complex internal data layouts: In the low-level code, a vector of integers is represented by two separate C values (element data and length), and a vector of vectors of integers by three, as outlined in Figure 3.

2. High-Level Low-Level Programming

How can we do low-level programming in a high-level language? The key idea of generative programming is to use the high-level language as a glorified macro system to compose fragments of low-level code in a (potentially domain-specific) intermediate language (IR). From this IR, low-level efficient C code can be emitted, possibly after further optimizations.

C as an Embedded DSL in Scala We base our development on Scala and LMS (Lightweight Modular Staging) [73]. The main idea of LMS is to provide a type-based embedding of the IR into the host language (Scala). A special type constructor `Rep[T]` is used to mark staged expression of type `T`. In other words, a value of type `Rep[Int]` or `Rep[String]` denotes a *representation* of an `Int` or `String` value in generated code.

Scala dialects provide flexible forms of overloading that extend to language built-ins like conditionals [71]. An expression such as

```
if (a < b) a else b
```

Will be desugared into method calls:

```
__ifThenElse(a.<(b), a, b)
```

And with LMS, two overloaded versions exist:

```
def __ifThenElse[T](c: Boolean, a: => T, b: => T): T
def __ifThenElse[T](c: Rep[Boolean], a: =>Rep[T], b: =>Rep[T]):Rep[T])
```

The Scala compiler's local type inference essentially performs a local binding-time analysis to determine which parts of an expression are evaluated right away and which parts are staged, becoming part of the generated IR.

To continue our example, if either `a` or `b` are `Rep[Int]` values, `a < b` will have type `Rep[Boolean]`. In this case a conditional will be generated. If `a` and `b` are of type `Int`, expression `a < b` will be of type `Boolean` and one of the conditional branches will be picked at staging (i.e. IR generation) time.

In traditional multi-stage programming approaches, staging is often homogeneous: the host language and the target language of code generation is the same. Here, we are mostly interested in generating C code from Scala, so we are dealing with a heterogeneous embedding. This gives us some freedom to provide types that have no direct meta-language equivalent. For example, for low-level systems programming we might need to deal with pointers, and add a corresponding `Pointer[T]` type to our embedding. In the same vein, we can add facilities like `malloc` and `free` for manual memory management, which do not have an equivalent in Scala.

In our setting, the target language is more restricted than the meta-language, as C does not provide features like closures, pattern matching, or managed memory. For this reason, and because analysis and verification of higher-order constructs is fundamentally more difficult, we want to generate mostly first-order entities.

Our IR is designed to be unparsed to C almost directly. Compared to C, we remove function pointers, and the ability to take the address of a value `x` via `&x`. These restrictions enable us to maintain some useful separation properties in the IR by design. We add support for *nested* functions, which requires a lambda lifting transform before unparsing (see Section 2.3).

2.1 Deep Linguistic Reuse

Working in a generative setting, the first key insight is that we can use arbitrary facilities of the meta-language at generation time, without any overhead in the generated code.

Classes and Objects On top of the primitives provided by the IR, we can, e.g., define a data structure for vectors, using high-level object-oriented abstractions in the meta-language for compositionality, similar to what we did in Figure 1:

```
class Vec[T](val a: Rep[Pointer[T]], val n: Rep[Int]) {
  def apply(i: Rep[Int]) = a(i);  def length = n }
```

In the generated code, all the abstraction overhead of classes and methods will be stripped away, leaving just the low-level pointer manipulations.

Shallow Functions We have seen primitive operations like `<` and conditionals above. But how can we effectively work with functions in generative programming? We first note that we can transparently use meta-language functions for composition:

```
def min(a: Rep[Int], b: Rep[Int]) =
  if (a < b) a else b
```

In this case, function `min` is a meta-language, i.e. staging-time, function of type `Rep[A]=>Rep[B]`. Such a function will be inlined when called, without leaving a trace of the function abstraction in the generated code.

Since such functions are purely staging-time abstractions, we can freely define higher-order functions, too, for example to traverse a `Vec`, and to compute aggregates:

```
def infix_foreach[T](xs: Vec[T])(f: Rep[T]=>Rep[Unit]) =
  for (i <- 0 until xs.length) f(xs(i))
def infix_fold[T](xs:Vec[T])(z:Rep[T])(f:(Rep[T],Rep[T])=>Rep[T]) = {
  var acc = z; xs.foreach(x => acc = f(acc,x)); acc }
```

If we now call:

```
val xs = ... // type Vec[Int]
xs.fold(Int.MaxValue)(min)
```

Then we will generate the following C code:

```
int* x0 = ...;    // xs.buf
int x1 = ...;     // xs.size
int x2 = INT_MAX; // acc
for (int x3; x3 < x1; x3++) {
  int x4 = x2;
  int x5 = x0[x3];
  int x6;
  if (x4 < x5)  x6 = x4;
  else          x6 = x5;
  x2 = x6
}
```

As we can see, the generated code is completely first-order. There are no functions present, and much less higher-order ones.

Generic Programming with Type Classes Most likely, we will reach a point where we would like to extend the functionality of Vec objects further. Perhaps we would like to compare two Vecs for equality. This requires that the elements of a Vec are comparable, too: a prime example of generic programming, which we can solve elegantly using type classes [91].

We begin by defining the type class signature, with a Rep[Boolean] result for the comparison:

```
trait Eq[T] { def eq(a: T, b: T): Rep[Boolean] }
```

And we provide a helper method to construct instances from a given comparison function:

```
def equality[T](f: (T, T) => Rep[Boolean]) = new Eq[T] {
  def eq(a: T, b: T) = f(a,b)
}
```

We also add an implicit enrichment

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

which enables us to call a deep_equal b when a and b are objects of a type T, for which a type class instance Eq[T] exists.

To instantiate equality for Vecs, we need to create the corresponding Eq instance and make it available as implicit. Note the use of deep_equal to compare the elements of the Vec:

```
implicit def vecEq[T:Eq] = equality[Vec[T]] { (x, y) =>
  x.n == y.n && ((0 until x.n) forall {i => x(i) deep_equal y(i)}) }
```

Interestingly, type classes can be used without any modification in a generative setting. The key insight is that type classes come with a natural stage distinction, given by the decoupling of data (which, in our case is staged) and behaviour (executed at staging time). Thus, we can reuse the host language once more and get this powerful pattern for free in staged code.

2.2 First-Class Now, Second-Class Later

The second key insight is that, through clever programming patterns, we can leverage the meta-language type system to ensure key desirable restrictions on generated code.

Higher-order programming and first-class entities are fundamentally linked. In general, language features are said to be first-class if they can be used without restrictions [79], and second-class if they can be used only in certain ways but not others. For example, second-class functions can only be called, but not stored in mutable variables or returned from other functions. Second-class mutable variables can be read and written, but cannot be stored in other variables. On the flip side, first-order code can use only second-class patterns while higher-order code uses first-class patterns. For example, first-order functions can use only second-class functions, while higher-order functions use first-class functions.

An extremely useful design pattern in generative programming is that second-class constructs from the target language may be abstracted over and used as first-class objects *at generation time*. First-class objects in generated code must be of type Rep[T], but we are free to provide additional abstractions that use Rep types internally but make them accessible only through a purely second-class API. We will see several examples below.

Deep Functions Above, we have discussed shallow functions, which only exist at staging time, and can be effectively used as macros, to achieve the effect of inlining. But of course inlining is not always the desired choice. Sometimes we want to generate actual function definitions. A truly first-class function would have type Rep[A=>B]. But since there is no C equivalent, it would not make much sense to support such abstractions. Instead, we introduce an operator fundef that creates a function definition, but its return type is again Rep[A]=>Rep[B]. This means that the defined function can only be called, but it cannot be referenced as a first-class value in the generated code. Thus, fundef is an abstraction to create second-class functions, with restricted usage patterns.

If we change the code above to use fundef:

```
val min = fundef("min") { (a: Rep[Int], b: Rep[Int]) =>
  if (a < b) a else b
}
val main = fundef("main") { (xs: Vec[Int]) =>
  xs.fold(Int.MaxValue)(min)
}
```

Then we will generate the following C code:

```
int min(int x4, int x5) {
  int x6;
  if (x4 < x5) x6 = x4; else x6 = x5;
  return x6;
}
int main(int *x0, int x1) { // x0 = xs.buf, x1 = xs.size
  int x2 = INT_MAX;
  for (int x3; x3 < x1; x3++) {
    int x4 = x2;
    int x5 = x0[x3];
    int x6 = min(x4,x5);
    x2 = x6
  }
  return x2;
}
```

Mutable Variables and Buffers Similar considerations as with functions apply when working with mutable state. In many cases we want to work with local variables in the generated code, but treat them also as first-class staging-time objects. The type of variables we use is plain Var[T], i.e. not a Rep type. But read and update methods on Var objects take and return Rep types, providing a second-class staging-time abstraction over generated code.

Nevertheless, we can treat variables as first-class entities while *generating* code. In particular, we can emit a flexible amount of local variables, based on some input data. For example,

```
List(1,2,3).map(x => var_new(x)) //: List[Var[Int]]
```

generates code:

```
int x1 = 1; int x2 = 2; int x3 = 3;
```

Or we can define staging-time functions that take variables as arguments, enabling programming patterns similar to ML-style first-class references:

```
def swap[T](a: Var[T], b: Var[T]) = {
  val t: Rep[T] = a
  a = b; b = t
}
```

The key difference to first-class references however is that their second-class nature bounds the lifetime of variables in the generated code. As fundef's type signature does not permit returning a Var[T] value, its lifetime cannot exceed its meta-language scope.

We can generalize from individual variables to second-class Buffer[T] objects, which leads to a simple but effective form of region-based memory policy.

Controlling Object Representation with Type Classes Consider again our Vec example. Right now Vec objects are second-class, like variables, so they cannot be arguments or return values of deep functions. This is rather limiting, and we might also want to nest Vecs, but we cannot: due to their second-class status, we can only put Rep values into Vecs.

To *make* Vecs first-class, type classes come to the rescue again. Based on ideas from polytypic programming [42, 84, 78] we introduce a type class Iso[T], which defines a mapping between Vec and the Rep types it consists of:

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

We create the corresponding instance for Vec as follows:

```
implicit def vecIso[T:Iso]: Iso[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)}
)
```

To go one step further and allow nesting of Vec objects, we need to generalize our low-level representation of pointers from Rep[Pointer[T]] to Pointer[T], with a suitable Iso instance for Pointer[T]. There are different options how such nested data can be stored. The most common variant would be as array of structs (AoS), but for performance, it is often desired to transpose the layout and store data as struct of arrays (SoA). The Iso implementation can pick either, but using explode_struct we settle for SoA as illustrated in Figure 3, in the code shown in Figure 2.

The internal definition of fundef requires an Iso instance for the argument and return type, in order to generate the proper function signatures. A first approximation of its signature would be:

```
def fundef[A:Iso,B:Iso](f: A => B): A => B
```

However we can take this pattern further and introduce a distinction between argument and return types. This is useful to ensure a "downward funarg" policy for certain types like stack-allocated buffers, which guarantees that object created inside a function cannot escape this scope. Thus, the actual signature of fundef is:

```
def fundef[A:DemiIso,B:Iso](f: A => B): A => B
```

Where DemiIso has the same functionality as Iso but serves as a marker for types like Var or Buffer that can only appear in function argument but not in return position. We treat Iso as strictly more general than DemiIso by defining an implicit conversion from Iso to DemiIso, but for types like Var or Buffer we only implement DemiIso, not Iso. This guarantees the desired behavior with respect to stack-bounded object lifetimes.

To wrap up this section, we highlight the insight that with type classes, we can control the shape of data (as in normal programming) but also the shape of generated code. In particular we can enable or restrict values of specific types to be nested, or to appear in function argument or return types.

2.3 IR Transformations and Unparsing to C

At the IR level, we can implement various analyses and optimizations before we generate target code. Unparsing to C is straightforward, since the IR is already low-level and designed to map to C in a direct way. One aspect still requires further attention:

Lambda Lifting Since Scala supports nested functions, invocations of fundef may also be nested, and refer to objects, including Vars, from enclosing scopes:

```
var a = ... // Var[Int]
val f = fundef { x: Rep[Int] => a += x }
f(2)
```

Of course, C does not support nested functions, so we have to apply a lambda lifting transformation on the IR level before emitting C. This lambda lifting pass takes variables into account and supports non-local variables by passing pointers. For the input above, the generated code will look (roughly) like this, with a pointer to variable a added to function f's argument list.

```
int f(int *a, int x) { *a += x; }
int a = ...
f(&a,2)
```

Passing a pointer to variable a to function f is safe, because the variable can never escape its defining scope, and because we do not support function pointers as first-class entities. Similarly, variables cannot be stored in other variables, and they cannot be returned from functions. Together, these constraints maintain our stack-bounded lifetime guarantees for second-class values.

2.4 Problems

All in all, generative programming provides a valuable approach to raise the level of abstraction in low-level systems development. Programmers obtain some nice high-level guarantees, including type safety, and certain properties about scoping, for example that variables and pointers do not escape. But how reliable are those guarantees? Since we are still generating low-level C code, we have to take into account bugs and inconsistencies at all levels of the stack.

First, even though we are working in a safe high-level language, we are giving up built-in properties like memory safety the very moment we decide to generate C code. We are no longer protected against out-of-bounds memory accesses, buffer overflows, or other kinds of errors that are all too often the source of security vulnerabilities. Of course it would be possible to generate appropriate checks, but the associated performance overhead would most likely not be acceptable.

Second, we cannot trust the generator to do the right thing, especially if one or more levels of IRs are involved in the generation. A concrete example we have come to know about was the data loading code in a database query engine built using LMS. While the data field abstraction was properly implemented to work with fixed-length strings, programmers had not accounted for the additional NULL terminator that is present in C strings, but not in the high-level abstraction.

Third, we do not know if the program itself does the right thing, since we have no principled way of incorporating verification of functional correctness properties into our program generators, except perhaps by generating costly runtime checks.

The central thrust of this paper is to show that these shortcomings can be fixed in an elegant and principled way, by generating C code along with specifications that can be automatically verified by independent tools. Our development uses Frama-C [21] and the ANSI C Specification Language (ACSL) [7] but other tools like VeriFast [40] would be viable alternatives.

3. Contracts

Specifications and in particular function contracts play a key role in program verification. The key idea of LMS-Verify, our generative programming system geared towards verifiable code, is to not only generate straight executable C code, but generate contracts and assertions alongside. These could be checked dynamically, but we seek full static verification in order to eliminate all possible runtime overhead. To achieve static verification, we need to generate enough assertions, loop invariants, and function pre- and post-conditions so that existing verification tools are able to reason about every possible behavior of a certain piece of generated code.

Contracts for Deep Functions In their simplest model, function contracts consist of pre- and post-conditions for a given function definition. We can add corresponding functionality to fundef:

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

The pre- and post-conditions are just regular staged expressions (type Rep[Boolean]), but since they occur in specification context they will not generate executable C code but ACSL annotations:

```
/*@
requires (\valid(x0+x1) && \valid(x0+x2));
ensures ((x0[x1]==\old(x0[x2])) && (x0[x2]==\old(x0[x1])));
assigns x0[x1], x0[x2];
*/
void inswap(int* x0, int  x1, int  x2) {
  int x4 = x0[x1];
  int x5 = x0[x2];
  x0[x1] = x5;
  x0[x2] = x4;
}
```

While in general most kinds of expressions can either generate C or ACSL code there are certain exceptions: side-effecting statements can only generate C code, and some expressions like `old` and `valid` are only available in specifications. Using such an expression in an incompatible context will result in an error at staging time.

Shallow Contracts Some dynamic languages such as Racket [29] offer elaborate APIs for higher-order composable contracts [30]. It turns out that we can use the same pattern we used for working with higher-order functions at generation time again for contracts: we use meta-language abstractions to compose computations that will produce the right assertions in the generated code.

In essence, adding pre- and post-conditions to a function just corresponds to eta-expansion, inserting an appropriate check before or after the function is executed. However, just a simple check is not enough. Since pre- and post-conditions form a contract between caller and callee of a functions, we would like to make sure that we can blame the right party for any failure.

Blame Assignment To assign blame for contract violations, we use Scala's implicit `SourceContext` facility [71]. Whenever the Scala compiler encounters a method call that requires a `SourceContext` argument maked as `implicit`, the compiler will synthesize such an object and initialize it with the source file and line number at the method call site.

```
def getPos(implicit pos: SourceContext) = pos
getPos() // automatically create SourceContext object
```

We can use this approach to define our own class of function objects `FuncShallow` that will store the source location of their definition as well as obtain the caller's source location when invoked.

```
trait FuncShallow[A,B] { o =>
  val calleePos: SourceContext
  def apply(x: A)(implicit callerPos: SourceContext): B
}
```

Base on this function class, we add contracts via pre- and post-conditions.

Contracts are added by eta-expanding and guarding pre/post conditions with an `assert`, blaming `callerPos` or `calleePos` respectively for any contract violations that might happen.

```
def shallow[A,B](f: A => B)(implicit pos: SourceContext) =
  new FuncShallow[A,B] {
    val calleePos = pos
    def apply(x: A)(implicit callerPos: SourceContext): B = f(x)
  }
```

We implement pre-conditions as follows:

```
  def require(pre: A => Rep[Boolean]) = new FuncShallow[A,B] {
    val calleePos = o.calleePos
    def apply(x: A)(implicit callerPos: SourceContext): B = {
      // blame precondition on caller
      asserts(pre(x))(callerPos)
      o.apply(x)(callerPos)
    }
  }
```

Post-conditions using `ensures` are analogous to the implementation for pre-conditions but they blame the callee instead of the caller. Note that the contract implementations use an `asserts` statement that looks very similar to a normal runtime check, but instead it will generate an ACSL assertion in the output C file. For blame assignment, we emit source information in the generated code using `#line file` directives (more below).

Higher-Order Contracts Straightforward contracts have their limits, even though we can make them tremendously more useful by being able to generate assertions instead of writing them all by hand. A key limitation is that we cannot easily state properties about functions that are passed as arguments or returned from other functions.

The idea of higher-order contracts is to use eta-expansion again, and insert code before and after the function, which will inspect its arguments and results. In the implementation, nested contracts are linked to their parents, and the blame assignment flips between caller and callee whenever we enter a pre-condition, i.e. increase the depth to the left of a function arrow. This approach follows exactly the standard algorithm for blame assignment in dynamically monitored contracts [30].

Here is an example, taken from [60], that transforms a function from even to even numbers into a function from odd to odd numbers:

```
val e2o = shallow { f => shallow { x => f(x+1)-1 } }
  .requiring { f => f.require(x => even(x)).ensure(x => r => even(r)) }
  .ensuring  { f => f.require(x => odd(x)).ensure(x => r => odd(r)) }
```

Note the difference between `require` and `requiring`: the former takes as argument a predicate of type `A => Rep[Boolean]` and asserts it before calling the wrapped function. The latter (`requiring`) takes as argument a function `A => A` and uses it to merely modify the argument of the wrapped function (adding nested contracts being the prime use case). The same applies to `ensures` and `ensuring`.

The example can also be written in shorter form as:

```
val e2o = shallow { f => shallow { x => f(x+1)-1 } }  // line 37
  .requiring { f => f.contract(even,even) }           //      38
  .ensuring  { f => f.contract(odd,odd) }             //      39
```

And we might use `e2o` as follows:

```
val double = shallow { x: Rep[Int] => 2 * x }         //      42
val f2 = e2o(double)                                  //      43
f2(x/*...defined in elided outer scope*/)             //      44
```

Below is the generated code, with line numbers for every expression. The `#line` directives in front of `asserts` lead the verifier to report contract violations in the correct source locations, according to blame assignment:

```
#line 44 "BlameTests.scala"
//@assert ((x0%2)==1);
#line 37 "BlameTests.scala"
int x6 = x0 + 1;
#line 37 "BlameTests.scala"
//@assert ((x6%2)==0);
#line 42 "BlameTests.scala"
int x11 = 2 * x6;
#line 43 "BlameTests.scala"
//@assert ((x11%2)==0);
#line 37 "BlameTests.scala"
int x16 = x11 - 1;
#line 37 "BlameTests.scala"
//@assert ((x16%2)==1);
```

Here, line 37 is the definition of `e2o`, and lines 42-44 are the three lines of its use. We can see that all parts of the contract, including the higher-order ones, have become first-order assertions. As to blame assignment, we can see that blame correctly switches between the function definition and the call site, blaming the call `e2o(double)` for any failures of `double` to return an even number, but blaming the body of `e2o` for any failures of invoking `f`, and therefore `double`, with an even number.

Mixing Deep and Shallow Contracts We have looked at contracts for "deep" functions, which are part of the `fundef` API, and "shallow" functions, which are generation-time macros.

Since deep functions take `Rep` arguments, or require an `Iso` instance, we do not have to deal with passing a shallow function as an argument to a deep function. However, one can pass a deep function as parameter to a shallow function. If the shallow function imposes any contract on its function parameter, then the deep function will be wrapped with code that performs the corresponding checks.

Inductive Predicates Similar to shallow/deep functions on the C level, we have the same choice for assertions. Either we can abstract at staging time and inline, or we can generate definitions in ACSL.

We already saw ACSL predicates in the context of specifying invariants and deep equality (Figure 2). For deep equality, we saw that we can generate both C code and a corresponding ACSL predicate from the same Scala source. Here is another small example:

```
predicate("is_pos", { x: Rep[Int] => x > 0 })
```

This source generates an ACSL predicate and C code that implements the logic, as ensured by the post-condition:

```
//@ predicate is_pos(int x0) = (x0>0);
/*@
assigns \nothing;
ensures \result <==> is_pos(x0);
*/
int is_pos(int x0) {
  int x2 = x0 > 0;
  return x2;
}
```

This predicate `is_pos` can now be used as part of any other spec or code, and will refer to the generated ACSL or C definition, which helps in making generated specifications more readable.

ACSL also supports inductive definitions, as a set of cases with the usual least fixed point semantics. We use this facility for example in our case study on sorting, to specify that the sorted output is a permutation of the input (see Section 5.1).

Properties of Contract Embedding Our generative embedding of contracts achieves much of the expressiveness of state-of-the-art dynamic contract systems [30]. In particular, contracts are composable, higher-order, dependent (they can express dependent relationships between the domain and range), potentially recursive, and they support blame assignment. Our embedding is heavily inspired by soft contract verification [59], but goes beyond it in guaranteeing that *all* dynamic contract checks are eliminated. Like TreatJS [45], our contracts guarantee *non-interference* of contracts with normal program code: we generate assertions in a specialized, side-effect-free, language. We control precisely what is necessary to compute only for verification vs for normal program execution. Computations only used in assertions are not emitted as part of the C code. Thus, adding or deleting contracts has no effect on the computation.

4. Generative Verification

Through contracts, we have already seen some of the power of the generative approach extended to verification. Here, we delve deeper into generative patterns for verification that further alleviate the annotation burden.

Type Classes for Invariants Recall from Section 2.2 that the type class `Iso` controls object representation. An important insight is that type classes can be used not only for shaping code, but also specifications. In particular, they can fill in parts of contracts automatically.

We define a core type class `Inv` for representation invariants:

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

which we instantiate for `Vec[T]` as follows:

```
implicit def vecInv[T:Inv] = invariant[Vec[T]] { x =>
  x.a.valid(0 until x.n) && ((0 until x.n) forall {i => x(i).valid}) }
```

The helper function `invariant` acts as default constructor analogous to `equality` for `Eq` as in Section 2.1. The given `Vec[T]` invariant requires that the data pointer `a` is valid for the full length `n`, and that the nested invariant for `T`, specified by `Inv[T]`, holds for all elements of the vector.

The type classes `Inv` and `Iso` co-operate mutually and with `fundef` so that representation invariants of parameters and return values are enforced in pre- and post-conditions of code functions. Type classes compose and interact transitively as we would expect, and as exemplified by the invariants for `T` and `Vec[T]` above. Furthermore, in the generated code for the teaser (Figure 2), the equality predicates require representation invariants in their C implementations (though not their ACSL definition, deliberately).

Type classes provide a discipline for structuring genericity – relying on parametricity by default and explicit functionality as needed – well-suited for composing and specializing specifications, and for automating simple contracts, cutting down on repetitive manual annotations.

Code and Specifications from a Single Source Back in our teaser example (Figures 1 and 2), the definition of equality generates both an ACSL logic predicate and a C implementation. Another example was `is_pos` in Section 3. Thanks to a single reified IR, we can generate either target from a single IR expression, depending on the context (spec vs code). Thus, we get a big benefit in terms of DRY ("Don't Repeat Yourself"). But how can we be sure that the generated code and the spec actually match? In the generated code in Figure 2, the post-conditions of the `eq_vec_` functions ensure that the specification is actually implemented.

It is important to stress that we cannot always turn code into specs. For example, side-effecting code has no logic counterpart. Thus we need a way to degrade gracefully. In the case of `Eq`, the earlier definition from Section 2.1 is refined to:

```
trait Eq { def eq(a,b); def eq_spec(a,b) = eq(a,b) }
```

Method `eq_spec` is what is used inside specifications. It defaults to the same expression that generates code, but it can be overridden to formulate the spec in another way, if it cannot be directly generated from the code. For example, the code might use caching over the pure `eq_spec` – a possible pattern to abstract over again.

The expression `(0 until n).forall {i => ...}` is another interesting case where we capitalize on the benefits of working with an IR instead of generating target code directly. If this expression is used as part of a spec, it will result in an ACSL `forall` expression:

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

But if it is used as part of executable code, then an iterative algorithm is generated that enumerates the range and *computes* the predicate. Another way to look at this is that for predicates that fit the given pattern, we automatically synthesize decision procedures. The take-away here is that a functional programming style lends itself to generating both spec and code from the same source expressions since it can be read both logically and computationally.

Deriving Specs from the IR Loop properties and aliasing invariants required for low-level verification (`invariant`, `variant`, `assigns`, `separated`, ...) are particularly tedious. But in many cases the information we need is already captured as part of internal effect and aliasing information by LMS, and we "just" need to expose it as ACSL annotations.

For most practical purposes, performance-oriented low-level code does not make arbitrary memory accesses but follows some reliable patterns. We introduce `a.mutable` as a specification internal to LMS-Verify, for a function argument `a`. The meaning is that mutable function arguments are separate from one another, and separate from immutable data.

```
def zero = fundef("zero") { a: Vec[Rep[Int]] =>
  requires { a.mutable }
  for (i <- 0 until a.n) a(i) = 0
  ensures { res => (0 until a.n).forall { i => a(i) = 0 }}
}
```

On the IR level, we track modifications, and report any violation of these separation properties as staging-time error. The generated post-conditions ensure that the separation invariants are preserved. The internal implementation is only a slight extension of the pre-existing LMS effect system [70], which already keeps track of certain sharing and separation properties for the purpose of optimizations. We also use information from these `mutable` annotations to generate loop invariants with ACSL `assigns` annotations.

Domain-Specific Loop Constructs Of course it is unreasonable to expect that we can infer everything we need for all potential use cases. Another powerful pattern to *programmatically* put invariants in the right places is to introduce higher-level domain-specific abstractions, which combine code, in particular loop patterns, with respective invariants. In the zero snippet above, the expression `for (i <- 0 until a.n) a(i) = 0` desugars into `Range(0,a.n).foreach { i => a(i) = 0 }`. The foreach implementation on `Range` objects can be seen as such a domain-specific abstraction. Internally, it is implemented like this:

```
def foreach(f: Rep[Int] => Rep[Unit]) = {
  var i = 0; while (i < n) { loop_invariant(i >= 0 && i < n); f(i) } }
```

Thus, higher-level abstractions (here, foreach) can automatically insert specific loop invariants, based on domain knowledge (here, about `i`). Together with `a.mutable` and `Inv[Vector[Int]]`, this loop invariant is enough to validate the postcondition in method zero. We will see another example in Section 5.2.

5. Case Studies

We report on four individual case studies:

1. Generic sorting: verify non-trivial properties (e.g. output is in-place sorted permutation of input) and generate specialized versions of a generic library parameterized by element data type, vector data layout, comparator operation. Library generation is an important use-case for generative programming – see Spiral [62] for example.
2. Linear algebra: alleviate the annotation burden by automatically inferring loop and data properties from the intermediate representation and from higher-level knowledge. We showcase scenarios often encountered in DSLs like OptiML [82].
3. Regular expressions: apply the common generative programming pattern of deriving a compiler from an interpreter via staging. We show that we can co-specialize specifications along with the interpreter code.
4. Parser combinators: this case study targets a particularly relevant real-world use case: verifying code that processes untrusted input. Protocol and data format parsing is one of the main sources of security vulnerabilities in the wild. We show that we can write an HTTP parser in a high-level style and generate low-level fast C code verified to be free of memory and overflow errors.

We present an overview of the required verification effort in Figure 4. For each case study, we list the lines of proofs (LoP) vs lines of code (LoC) in the Scala source, as well as in the generated C. Note that since we are working in a generative setting, we can generate *many* C variants, including specifications, from a *single* Scala source. For some use cases (e.g. sorting) the generator takes more LoC than a single generated target, but the source LoC are amortized over multiple generated versions. In other cases (e.g. HTTP parser) the generated to generator LoC ratio is already high ($\geq 10\times$) for a single version. For regular expressions, the generated code size can blow up exponentially as the regular expression grows in backtracking complexitiy. In general, the LoP to LoC ratio is lower than one might expect in the generated code. This is mainly due to the fact that all intermediate expressions are named: each one occupies a separate line in C code, but not in ACSL specs.

module / instance	lang.	LoP	LoC	#	s.
1. **Selection Sort**	Scala	41	115		
verified sorted & in-place permuted					
ints by $\leq$	C	88	26	70	7
ints by $\geq$	C	88	26	70	7
int pairs by first proj.	C	116	43	89	11
int pairs by lex.	C	130	52	97	20
int vectors by length	C	128	49	109	15
2. **Linear Algebra**	Scala	32	97		
verified misc. safe & spec.					
matrix $+,\times,\cdot$	C	104	51	101	22
member	C	23	22	18	2
index where	C	26	17	20	3
3. **Regular Expressions**	Scala	3	41		
verified memory safe					
annotated interpreter	C	32	35	72	5
`^a`	C	4	21	2	1
`a`	C	10	39	21	11
`a$`	C	10	41	24	12
`ab.*ab`	C	16	155	104	28
`aa*`	C	16	80	62	13
`aa*bb*`	C	28	192	160	31
`aa*bb*cc*`	C	52	416	356	100
`aa*bb*cc*dd*`	C	100	864	748	320
`aa*bb*cc*dd*ee*`	C	196	1760	1532	1040
`aa*bb*cc*dd*ee*ff*`	C	388	3552	3100	4820
4. **HTTP Parser**	Scala	2	118		
staged parser combinators	Scala	5	197		
verified memory & overflow safe					
without chunking	C	95	1517	276	27
with chunking	C	133	2630	2404	650

Figure 4. Verification effort for case studies: lines of proof/specification (LoP) & lines of code (LoC) in Scala code generator and generated C code, which Frama-C verifies using given number of goals (#) in given seconds (s.).

5.1 Generic Sorting

We write an in-place sort routine – we choose selection sort, though another in-place algorithm such as insertion sort or even quick sort would work as well. This case study is inspired by a hand-written verified sorting function [19], but is considerably more general. Namely, the code is parameterized on an element type `T` and on an instance of the type class `Ord[T]`, which defines an ordering, made available through the comparison function `cmp`:

```
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) }
}
```

The sort routine operates on a generic `Vec`, representing an indexed sequence of n elements. We verify that on output, the sequence is a sorted permutation of what it was on input. The `a.mutable` precondition tracks modifications of vector `a`. The first post-condition ensures that the sequence is sorted and the second that the sequence on output is a permutation of the sequence on input. The first helper is straightforward to define:

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

The second one is a bit more involved. First, note that `Permut` needs to relate two states of the *same* data structure. ACSL provides syntax `\at(a,Old)` and `\at(a,Post)` for this, which we expose directly in our programming model. We model permutations in the usual way as a a combination of zero or more swap operations. Thus, it is convenient to first define a predicate `Swapped`:

```
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)))}
}
```

where for `equal`, we can choose either deep or shallow equality. The parameters `l1` and `l2` of type `Lc` range over `Old`,`Post`, and potentially other modifiers, as in plain ACSL. The helper function `inswap` already can use `Swapped` in its post-condition. While `Swapped` captures the swapping of two *specific* elements, we actually want to express the property of swapping *any* two elements for the purpose of defining permutations. We thus add an auxiliary definition `Swapped1`:

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

In a second step, we define `Permut` as the reflexive, transitive, closure of `Swapped1`:

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

The combinator `reflexiveTransitiveClosure` will generate an inductive definition in ACSL. For the instantiation `Permut[Rep[Int]]`, the generated code looks as follows:

```
/*@ 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) ; }*/
```

The parameters `a` and `n` correspond to the fields of `Vec`, according to its `Iso` definition. The body of the `Permut_Int_step` case contains the inlined definition of `Swapped1` on the left-hand side of the implication `... ==> Permut_Int{L1,L2}(a, n)`.

Over the plain version in Frama-C, the first benefit of staging is that we are able to define and use higher-order combinators like `reflexiveTransitiveClosure`, much like higher-order functions in normal code. The second major benefit is that we can abstract over the datatype and over the comparison function very easily. In particular, the comparator `cmp` works both as C code and as ACSL specification. From the sort routine, we can derive specialized verified sort functions for any type `T` for which we have a comparator. Finally, we can also control the data layout of the sequence itself by varying the `Iso` type class for `Vec` and `Pointer`.

The definition of `Ord[T]` follows that of `Eq[T]` seen earlier. What is interesting, although standard type class practice, is that we can define higher-order instances of `Ord`, for example, in order to combine multiple separate orderings into a joint lexicographic ordering. In our generative setting, the `Ord` instances can also decide whether the comparison will be implemented as a shallow function (i.e. inlined into function `insort`) or delegate to a generated function.

Since we only verify the generated code, we only verify particular instances of the sort routine, not the generic one (which does not exist as C code, only in the generator). So what happens if we specialize the sort routine with an unsuitable comparator, for example, one that is not transitive? The generated code will fail to verify. In the generator, we do not have to explicitly require that the comparator be transitive – it will verify when it is, and will not when it isn't. However, we can further explicitly indicate this requirement in the generator, guiding blame assignment to fault the comparator instantiation rather than the sorting routine.

In order to achieve this, we add a method `valid` to the `Ord[T]` interface, which will define the property of actually being an ordering relation, namely reflexivity, antisymmetry, and transitivity.

```
trait Ord[T] {
  def cmp: (T,T) => Rep[Boolean]
  def valid = reflexivity && antisymmetry && transitivity
  def reflexivity = forall{x => cmp(x,x)}
  ...
}
```

In method `insort`, we can now add a check `ord[T].valid` to the precondition, and thus verify the desired properties up front. This will prevent any spurious verification errors in the middle of `insort` that are due to the provided comparison lacking one of these properties, resulting in more accurate blame assignment.

Finally, this case study illustrates that the overhead of being generic does not translate to much overhead for the programmer in terms of *generative* verification.

5.2 Linear Algebra Library

For this case study, we implement a small linear algebra library with operations on matrices and vectors. First, we define a class `Matrix` in the same style as the class `Vec`.

Here is the definition of matrix multiplication, which generates code that fully verifies in Frama-C. The definition is parameterized over the element type `T` of the matrix as well as a suitable type class instance specifying a ring over `T`, an algebraic structure defining (zero, +, *).

```
def mmult[T:Iso:Ring] =
  fundef("mmult") { (a: Matrix[T], b: Matrix[T], o: Matrix[T]) =>
    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)
        }}}
  }
```

Through `Iso[Matrix[T]]`, the appropriate data invariants are automatically added as pre- and post- conditions. This is why the remaining explicit pre-condition focuses on the dimension constraints that are specific to the problem at hand. This automation for data invariants also makes it convenient to call these linear algebra routines as one would an external library, without having to worry about annotations.

The framework also automatically infers the loop properties, because they are generic to for-loops and not specific to the problem at hand. For example, here is the ACSL code generated for the first loop for `(r <- 0 until a.rows)`, modulo renaming to match source instead of target:

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

Finally, we can use the host language to structure code and specification logic further. For example, we define pointwise addition and scalar multiplication, delegating to an underlying common staging-time abstraction `setFrom`:

```
def madd[T:Iso:Ring] =
  fundef("add") { (a: Matrix[T], b: Matrix[T], o: Matrix[T]) =>
    o.setFrom2({ (ai: T, bi: T) => ai + bi }, a, b) }
def smult[T:Iso:Ring] =
  fundef("smult", { (a: T, b: Matrix[T], o: Matrix[T]) =>
    o.setFrom1({ (bi: T) => 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 }}}})
```

The higher-order staging-time abstraction `setFrom` in the `Matrix[T]` class assigns element-wise, mapping the staging-time function argument `f` over a list of matrices:

```
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)
  }
}
```

The staging-time function passed in as argument is required to be pure, so it can generate both C code and ACSL logic. Hence, the staging-time abstraction can also decorate the result with extra specifications, and thanks to these, we can verify further domain-specific properties of scalar multiplication. Also, the staging-time abstraction can add its own pre-condition to require matching dimensions of the argument and output matrices.

In general, we found it a useful pattern to create our own staging-time abstractions that encapsulate verification properties. Here, notice that `setFrom` does not pass in the position of the element to the (pure) calculation, so we know that each entry is conceptually computed independently. We can also define another abstraction that passes the position (row index, column index) in the matrix, encapsulating the specific separation requirements and properties for this variant.

5.3 Regular Expression Matcher

We illustrate that one can often reason once and generically at generation time, producing specialized low-level code along with the annotations to verify it. We take a staged regexp interpreter, which compiles a regular expression to low-level code for matching a string, and by minimally annotating the staged interpreter, produce low-level code that can be verified to be memory-safe.

Our starting point is a simple regular expression matcher [46]. We view it as a tiny interpreter, where the program is the regular expression and the input is the text to match. It is well-known [35] that we can turn an interpreter into a compiler via staging, by making the program (i.e. the regular expression) known and the input (i.e. the text to match) unknown to the generator [77]. Of course, we do not necessarily get an optimizing compiler, but in the case of regexp matchers, we can exploit the connection to finite automata to optimize beyond simple specialization [76, 74].

In this case study, we simply adapt this well-known generative programming pattern of turning an interpreter into a compiler, so that the low-level specialized code verifies to be free of memory errors. It is nice that we can annotate the generic code once, and derive verified code for each regular expression – although we don't verify the generator: we have to verify the generated code with Frama-C for each specialization.

In total, we only need three straightforward annotations in the generator: we simply require that the input text is a valid string, and we annotate each loop with very simple invariants that guide Frama-C to accept that the string is always accessed within its valid range – in the generator, only two source loops are potentially reified into the generated code.

For comparison, we also annotated the original regular expression matcher interpreter, so that Frama-C can verify its memory safety. While straightforward, it is interestingly more work than annotating the generator, because Frama-C needs to ensure that the regular expression is also accessed within range, while of course, in the compiler variant, the regular expression just becomes part of the low-level code. We also had to guide Frama-C with an assertion so that it could see that a recursive call (on a smaller part of the regular expression) was indeed well-founded.

Finally, as expected, the specialized code is usually faster than the original regular expression matcher. For example, on the text `acacabc...caba` of length 1000000 where `...` are all c's, which matches the regular expression `ab.*ab`, the specialized code is about twice as fast as the original interpreter in C.

```
// trait HttpParser extends StagedParser
def status: Parser[Int] =
  (accept("HTTP/") ~> acceptNat ~> accept('.') ~>
   acceptNat  ~> whitespaces) ~>
  nat <~ acceptLine
def headerMap: List[(String, Int)] =
  ("Content-Length", CONTENT_LENGTH):: Nil
def headerName: Parser[Int] =
  ((for ((h,i) <- headerMap) yield (accept(h) ^^^ i)).reduce(_ | _)) |
  (repUnit(letter | accept('-')) ^^^ OTHER_HEADER)
def headerValue(h: Rep[Int]) =
  if (h==CONTENT_LENGTH) (nat <~ whitespaces <~ acceptNewline)
  else (acceptLine ^^^ NO_VALUE)
def header: Parser[Int] =
  (headerName <~ whitespaces <~ accept(':') <~ whitespaces) >>
  headerValue
def headers: Parser[Int] =
  rep(header, 0, { (a: Rep[Int], x: Rep[Int]) =>
    if (x==NO_VALUE) a else x })
def acceptBody(n: Rep[Int]): Parser[Int] =
  if (n<0) Parser[Int] { input => Failure(input) }
  else (repN(anyChar, n) ^^^ n) <~ acceptNewline
def http: Parser[Int] =
  (status ~> headers <~ acceptNewline) >> acceptBody
```

Figure 5. HTTP parser via staged parser combinators in Scala

```
// trait ChunkedHttpParser extends HttpParser
override def headerMap = super.headerMap :+
  ("Transfer-Encoding", TRANSFER_ENCODING)
override def headerValue(h: Rep[Int]) =
  if (h==TRANSFER_ENCODING)
    (accept("chunked") ^^^ CHUNKED) <~ whitespaces <~ acceptNewline
  else super.headerValue(h)
override def acceptBody(n: Rep[Int]): Parser[Int] =
  if (n==CHUNKED) chunkedBody else super.acceptBody(n)
def hexDigit2Int: Parser[Int] =
  digit2Int |
  (acceptIf(c => c >= unit('a') && c <= unit('f')) ^^
    (c => 10+(c - unit('a')).asInstanceOf[Rep[Int]]))
def hex: Parser[Int] = num(hexDigit2Int, 16)
def acceptChunk: Parser[Int] =
  (hex <~ acceptNewline) >> super.acceptBody
def chunkedBody: Parser[Int] =
  rep(acceptChunk, 0, { (a: Rep[Int], x: Rep[Int]) =>
    if (a<0) a
    else if (a>Int.MaxValue - x) OVERFLOW
    else a+x
  }, overflowOrPos)
```

Figure 6. HTTP parser extended with chunking

5.4 HTTP Parser

We adapt the work on staged parser combinators [44, 43] to generate C code instead of Scala code. After this initial port (which amounts to switching the code generation backend of the LMS framework), with minimal additional work, we further ensure that

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

Figure 7. HTTP parser performance results (higher is better)

the generated C code verifies completely with Frama-C, guaranteeing the absence of memory and overflow errors. Thus, we obtain a high-performance and secure HTTP parser. While HTTP parsers have caused major vulnerabilities in both Apache and Nginx related to chunk processing [12, 23, 24], we show that we can ensure safety without sacrificing performance nor productivity.

We build our HTTP parser using high-level combinators. Since we can use staging-time abstractions to structure our code without penalty, we build a basic HTTP parser (Figure 5) and extend it to handle chunk processing (Figure 6). The generated code for the HTTP parser is some thousands of lines of low-level C code, a glimpse of which we display in Figure 8.

We show performance results in Figure 7, comparing to the HTTP parser of Nginx [61], which is written in C as well. On a benchmark for parsing HTTP responses and validating the payload length, our approach is competitive, processing one million items per second.

We only need a few changes to go from a working to a verified parser. First, like for the regular expression matcher, we require and ensure that the input remains valid:

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

We also need some annotations on loops, but we have very few of those. The combinator `rep` uses a while loop to repeatedly parse with another combinator `p`:

```
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 = 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)
}
```

The invariant ensures that the input remains valid, but also, optionally, a custom invariant that depends on the accumulated result.

Finally, we need the parser to behave adequately in case of overflow. In particular, the usual `num` combinator to parse a number fails to verify, because of possible overflow errors:

```
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 })
  }
```

Thus, we overwrite this parser combinator to explicitly handle overflow errors:

```
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)
  }
```

This final step provides the main assurance we were after: the absence of overflows that could be exploited by malicious inputs,

```
#include <limits.h>
#include <string.h>
char * p_chunked(char * x1730);
char * p_TransferEncoding(char * x1732);
char * p_ContentLength(char * x1734);
char * p_HTTP(char * x1736);
/*@
requires ((strlen(x0)>=0) && \valid(x0+(0..strlen(x0))));
assigns \nothing;
*/
int p(char * x0) {
  int x2 = -1;
  char *x3 = p_HTTP(x0); ...
  if (..) { ...
    /*@
    loop invariant ((strlen(x468)>=0) && \valid(x468+(0..strlen(x468))));
    loop assigns x468, x469, x470;
    */
    for (;;) {
      int x471 = x469;
      if (!x471) break;
      char *x473 = x468;
      char *x474 = p_ContentLength(x473); ...
      if (x514) {
          char *x515 = p_TransferEncoding(x473); ...
      } ... } ... } ... }
/*@
requires ((strlen(s)>=0) && \valid(s+(0..strlen(s))));
assigns \nothing;
ensures ((0==\result) || ((strlen(\result)>=0) &&
  \valid(\result+(0..strlen(\result)))));
*/ // elided contracts below are identical
char * p_HTTP(char * s) { ... }
/*@ ... */ char * p_ContentLength(char * s) { ... }
/*@ ... */ char * p_TransferEncoding(char * s) { ... }
/*@ ... */ char * p_chunked(char * s) { ... }
```

Figure 8. HTTP parser generated C code

as demonstrated with known vulnerabilities in popular servers like Nginx and Apache [23, 24].

6. Discussion

We aim to give a broader perspective on our approach and answer some common questions below.

Why not write code in a low-level language with an advanced type system that ensures memory safety, such as Rust? While Rust [55] seems well on track to become a viable and safer replacement for C in many situations, we still want the benefits of generative programming and "abstraction without regret": on the one hand performance without giving up on productivity and expressive high-level abstractions, and on the other hand the ability to verify functional correctness via (potentially higher-order) contracts.

Why not use a static analyzer or model checker that has more automation than Frama-C? Wouldn't this remove the need for generating all these annotations? Automatic analysis of C code is a hard problem, despite some successes. For full static verification of code that may include unbounded loops we are not aware of tools that would improve over Frama-C in a fundamental way, although using other comparable tools like VeriFast [40] would certainly be feasible.

Why not verify high-level code directly? Already, we do not trust the code generator. We would trust a high-level verifier much less: does what is verified correspond to what is generated? Program generators can be thought of as DSLs, with the implied assumption that one needs many of them, and that they can be built up from modules. Thus, the overall system will be in flux and under active development, which makes it appear safer to rely on standard tools for C. There are big teams working behind these tools, industrial best practices, fuzzing attacks on analyzers [22], and so on. Furthermore, because verification is independent, we can afford to take more risks, and be more experimental during code generation.

Why not write everything in Coq and verify the generator? This is a much harder problem, despite much ongoing and very

promising work in this area [17, 16, 92, 18, 25]. We settle for proof carrying code / translation validation instead of certified compilers. By generating C code and using an off-the-shelf verifier we obtain modularity and modular verification. We can interface with other C code, which can be verified independently. And we also get to use existing, powerful, program generation frameworks (LMS).

Translation validation has a number of practical benefits: we can perform optimizations on the LMS IR, but we are verifying the code after optimization. Any properties expressed as specifications will still hold. This aspect is similar to other recent systems like Cogent [69, 2] (which also targets C), but it is important to look at the differences, too. Cogent in particular is a stand-alone DSL where "interesting" program properties can be encoded in the type system, e.g. using linear types. The Cogent compiler then uses translation validation to generate verified code. In being a complete language on its own, Cogent does not have any relation to *generative programming*, which, as mentioned earlier, we crucially want for performance.

How do we ensure that transformations do not remove or tamper with specifications? When checking generic safety properties, such as memory safety, we can do an end-to-end check on the generated code level, and trust the safety claim regardless of the correctness of the generator and its source. For modular verification (where we rely on pre-conditions to ensure safety), we need to be more careful. For functional correctness properties beyond safety, we can either inspect and audit the generated specifications, or, better yet, add a manual layer at the C level that calls generated C functions and verifies their properties using externally provided ACSL specifications – again, a benefit of targeting a standard specification language.

7. Related Work

The present paper touches aspects of security, verification, formal methods, as well as embedded DSLs and generative programming – all vast spaces of their own. We mention the most important lines of related work below.

Memory Safety Control-flow hijacks based on memory corruption bugs remain the largest attack vector for low-level code. Plainly rewriting C code in safe high-level languages is deemed unrealistic for performance [85], and fully verifying existing C code statically is unrealistic for complexity. This dilemma has lead to an arms race between offensive and defensive techniques [85] based on various kinds of dynamic monitoring (e.g. CFI [1]), which also introduce varying levels of overhead, and are often based on simplifying assumptions that lead to incompatibilities or more sophisticated attacks. Generative programming eliminates the performance drawbacks of high-level languages, but re-introduces security concerns. This paper shows how the two can be reconciled to generate C code which can be statically verified with reasonable effort.

Parsing as Attack Vector Working with untrusted input requires care. Hence, many vulerabilities arise at this boundary, and are often overlooked. For example, after the celebrated static verification of PolarSSL 1.1.8 [90], a remote code execution vulnerability was found due to a bug in an ASN.1 parser in the X.509 module, which was not part of the verification effort [4, 67]. Exploitable bugs in HTTP parsers have been reported for all major web servers, e.g. Nginx [24] and Apache [23]. A recent paper from the security community makes a case for verified parsers, and in particular mentions parser combinators [12]. In this paper, we used parser *generator* combinators [44] to build an HTTP parser that is memory and overflow safe, and as fast as Nginx.

Static Verification Beyond "simple" security properties like memory safety, functional correctness requires verification with respect to a specification. There exists many static verification, analysis, and model-checking tools of various degrees of sophistication for a variety of languages, e.g. VeriFast [40] Frama-C [21], Astré [20], Boogie [5], Spec# [6], Dafny [53], Leon [49], BLAST [37]. Still, building fully verified software remains extremely labor intensive. Notable successes include the seL4 and Verve verified OS kernels [47, 94], the Windows device drivers project [10], and the CompCert verified C compiler [54]. A recent example of translation validation is Cogent [69, 2], a DSL for verified systems programming that generates C. The correctness of verification tools is also an important concern [22]. Nonetheless, there are clear signs that formal verification is on the verge of becoming practical for larger classes of software (e.g [58]).

We believe that our work has the potential to accelerate this process by integrating verification with existing generative programming frameworks, which enables pragmatic trade-offs, and in particular allows programmers to transparently use higher-order constructs, which are poorly supported by verification tools, at generation time without leaving a trace in the code that is actually verified. Our work is heavily inspired by research coming out of DARPA's HACMS program [31], in particular Chlipala's Bedrock system [17], related work on verified systems-level software [16, 92, 18], and the DSL-based approaches by Galois and others [28, 38, 63] – all pioneering the use of meta-programming for verification.

Contracts The "Design by Contract" programming model has its origin in Eiffel [56, 57]. Higher-order contracts were introduced by Findler and Felleisen [30], and play a significant role in the Racket ecosystem [29, 26, 80, 89]. Higher-order contract systems have also been proposed for other languages, e.g. JavaScript [45]. The performance overhead of dynamic contract monitoring is known to be large [81, 87, 88], although clever JIT compilation techniques are a promising avenue to reduce overhead [8].

A key realization is that higher-order contracts only fail at first-order checks. The present paper is greatly inspired by recent work on soft contract verification [59, 60], which uses symbolic execution to unfold higher-order function calls to reach a first-order representation, which can be statically checked. Our approach is based on the same insight, but uses staging-time evaluation to eliminate higher-order functions, instead of symbolic execution. In contrast to *soft* verification, which works well in many cases, but may leave some dynamic checks in other cases, our meta-language type distinction between normal and staged expressions *guarantees* the absence of dynamic checks.

Generative Programming Generative or multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [86] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [14] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [73] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code [70]. LMS draws inspiration from earlier work such as TaskGraph [9], a C++ framework for program generation and optimization. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [15] and Hofer et al. [39], going back to much earlier work by Reynolds [68]. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [52] and Elliot et al. [27].

A number of high-performance program generators have been built, for example ATLAS [93] (linear algebra), FFTW [34] (discrete fourier transform), and Spiral [64] (general linear transformations). Other systems include PetaBricks [3], CVXgen [36] and Halide [66, 65]. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [74, 13, 75, 51, 83]. Related work on low-level systems oriented programming in high-level languages in-

cludes [33] and the Singularity operating system [50], the LegoBase database engine [48], and staged parsers for communication protocols [44].

8. Conclusions

We make the case that generative programming enables "abstraction without regret" for security and verification as it does for performance. Like for performance, we can use staging-time abstractions and exploit domain-specific and high-level knowledge to keep productivity high while enabling independent verification of the generated C code annotated with ACSL specifications. For example, we generate a low-level fast HTTP parser, verified to be safe of memory and overflow errors, from a high-level source using staged parser combinators with just a handful of lines needed to achieve safety.

Generative programming composes well with verification when the properties to verify match the abstractions used at staging time. In such cases, it is easy to also compose the annotation logic in order to ensure verification. Custom correctness properties require more effort (e.g. sorting), but this effort is not more than what would be required in a non-generative setting, and the generative setting can be further exploited to parameterize use cases over such an effort. Overall, we were 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.

By extending generative programming to verification, we hope to enlarge the appeal of the approach beyond high-performance computing to also safety-critical domains. By enabling users in such domains to leverage "abstraction without regret", we hope to accelerate the spreading of formal methods based approaches and static verification to larger classes of software.

Acknowledgments

The authors thank Viktor Kuncak and Sandrine Blazy for insightful discussions. Manohar Jonnalagedda deserves credit for helping with the HTTP parser case study, which is based on his prior work. This research was supported by NSF through awards 1553471 and 1564207.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[2] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. C. Murray, G. Klein, and G. Heiser. Cogent: Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188. ACM, 2016.

[3] S. P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In M. Katevenis, M. Martonosi, C. Kozyrakis, and O. Temam, editors, *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, page 3. ACM, 2011.

[4] ARMmbed. PolarSSL security advisory 2014-04, 2015. `https://tls.mbed.org/tech-updates/security-advisories/polarssl-security-advisory-2014-04`.

[5] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[7] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language, reference manual, version 1.11, 2009-2016. `http://frama-c.com/download/acsl.pdf`.

[8] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: a tracing JIT for a functional language. In Fisher and Reppy [32], pages 22–34.

[9] O. Beckmann, A. Houghton, M. R. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.

[10] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 2011.

[11] J. T. Boyland, editor. *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[12] S. Bratus, M. L. Patterson, and A. Shubina. The bugs we have to kill. *; login:: the magazine of USENIX & SAGE*, 40(4):4–10, 2015.

[13] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.

[14] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. GPCE, pages 57–76, 2003.

[15] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[16] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 18–37. ACM, 2015.

[17] A. Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013.

[18] A. Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 609–622. ACM, 2015.

[19] U. Costa. Correct sorting with Frama-C and some thoughts on formal methdos, Feb 2011. ulissesaraujo.wordpress.com.

[20] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[21] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. *Frama-C*, pages 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[22] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2012.

[23] CVE. 2002-0392: Apache security advisory. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0392`.

[24] CVE. 2013-2028: nginx security advisory. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028`.

[25] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deduc-

tive synthesis of abstract data types in a proof assistant. In *POPL*, pages 689–700. ACM, 2015.

[26] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16, 2011.

[27] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[28] T. Elliott, L. Pike, S. Winwood, P. C. Hickey, J. Bielman, J. Sharp, E. L. Seidel, and J. Launchbury. Guilt free ivory. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 189–200. ACM, 2015.

[29] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. A. McCarthy, and S. Tobin-Hochstadt. The racket manifesto. In T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 113–128. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[30] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In M. Wand and S. L. P. Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 48–59. ACM, 2002.

[31] K. Fisher. HACMS: high assurance cyber military systems. In *HILT*, pages 51–52. ACM, 2012.

[32] K. Fisher and J. H. Reppy, editors. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM, 2015.

[33] D. Frampton, S. M. Blackburn, P. Cheng, R. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In A. L. Hosking, D. F. Bacon, and O. Krieger, editors, *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 81–90. ACM, 2009.

[34] M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[35] Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[36] M. Hanger, T. A. Johansen, G. K. Mykland, and A. Skullestad. Dynamic model predictive control allocation using CVXGEN. In *9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, December 19-21, 2011*, pages 417–422. IEEE, 2011.

[37] T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST software verification system. In *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 25–26. Springer, 2005.

[38] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded dsls. In Jeuring and Chakravarty [41], pages 3–9.

[39] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

[40] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[41] J. Jeuring and M. M. T. Chakravarty, editors. *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. ACM, 2014.

[42] J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer, 1996.

[43] M. Jonnalagedda. Staged parser combinators and recursion, Sep 2015. manojo.github.io.

[44] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky. Staged parser combinators for efficient data processing. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 637–653. ACM, 2014.

[45] M. Keil and P. Thiemann. Treatjs: Higher-order contracts for javascripts. In Boyland [11], pages 28–51.

[46] B. Kernighan and R. Pike. A regular expression matcher. In G. Wilson and A. Oram, editors, *Beautiful Code*, chapter 1. O'Reilly, 2007.

[47] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.

[48] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[49] V. Kuncak. Developing verified software using leon. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 12–15. Springer, 2015.

[50] J. R. Larus and G. C. Hunt. The singularity system. *Commun. ACM*, 53(8):72–79, 2010.

[51] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[52] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[53] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[54] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[55] N. D. Matsakis and F. S. Klock, II. The Rust language. *Ada Lett.*, 34(3):103–104, Oct. 2014.

[56] B. Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[57] B. Meyer. Eiffel as a framework for verification. In *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 301–307. Springer, 2005.

[58] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[59] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In Jeuring and Chakravarty [41], pages 139–152.

[60] P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Higher-order symbolic execution for contract verification and refutation. 2015.

[61] nodejs & nginx. HTTP parser. `https://github.com/nodejs/http-parser`.

[62] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In J. Järvi and C. Kästner, editors, *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 125–134. ACM, 2013.

[63] L. Pike, P. C. Hickey, J. Bielman, T. Elliott, T. DuBuisson, and J. Launchbury. Programming languages for high-assurance autonomous vehicles: extended abstract. In N. A. Danielsson and B. Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL '14*, pages 1–2. ACM, 2014.

[64] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*,

18(1):21–45, 2004.

[65] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

[66] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013.

[67] J. Regehr. Comments on a formal verification of PolarSSL, 2015. `http://blog.regehr.org/archives/1261`.

[68] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.

[69] C. Rizkallah, J. Lim, Y. Nagashima, T. Sewell, Z. Chen, L. O'Connor, T. C. Murray, G. Keller, and G. Klein. A framework for the automatic formal verification of refinement from cogent to C. In *ITP*, volume 9807 of *Lecture Notes in Computer Science*, pages 323–340. Springer, 2016.

[70] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[71] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation (Special issue for PEPM'12).

[72] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and dsls in performance critical systems. In *SNAPL*, 2015.

[73] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[74] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.

[75] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. DSL, 2011.

[76] Scala-LMS. Tutorial: Automata-based regex matcher. `http://scala-lms.github.io/tutorials/automata.html`.

[77] Scala-LMS. Tutorial: From interpreter to compiler. `http://scala-lms.github.io/tutorials/regex.html`.

[78] A. Slesarenko, A. Filippov, and A. Romanov. First-class isomorphic specialization by staged evaluation. In *WGP*, pages 35–46. ACM, 2014.

[79] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[80] T. S. Strickland, C. Dimoulas, A. Takikawa, and M. Felleisen. Contracts for first-class classes. *ACM Trans. Program. Lang. Syst.*, 35(3):11, 2013.

[81] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 943–962. ACM, 2012.

[82] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[83] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, 2013.

[84] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.

[85] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[86] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[87] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, and M. Felleisen. Towards practical gradual typing. In Boyland [11], pages 4–27.

[88] A. Takikawa, D. Feltey, B. Greenman, M. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *POPL*, 2016.

[89] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008.

[90] TrustInSoft. PolarSSL 1.1.8 verification kit, 2015. `http://trust-in-soft.com/polarSSL_demo.pdf`.

[91] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[92] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In J. Flinn and H. Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 33–47. USENIX Association, 2014.

[93] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[94] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM*, 54(12):123–131, 2011.