Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures

Joshua Auerbach David F. Bacon Perry Cheng Rodric Rabbah
IBM Research
{josh,dfb,perry,rabbah}@us.ibm.com

Abstract

The halt in clock frequency scaling has forced architects and language designers to look elsewhere for continued improvements in performance. We believe that extracting maximum performance will require compilation to highly heterogeneous architectures that include reconfigurable hardware.

We present a new language, Lime, which is designed to be executable across a broad range of architectures, from FPGAs to conventional CPUs. We present the language as a whole, focusing on its novel features for limiting side-effects and integration of the streaming paradigm into an object-oriented language. We conclude with some initial results demonstrating applications running either on a CPU or co-executing on a CPU and an FPGA.

Categories and Subject Descriptors B.6.3 [*Design Aids*]: Hardware Description Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.3 [*Programming Techniques*]: Concurrent Programming, Object-oriented Programming

General Terms Design, Languages

Keywords object oriented, value type, streaming, functional programming, reconfigurable architecture, FPGA, high level synthesis

1. Introduction

The search for single-CPU performance has moved beyond clock frequency scaling which has almost ground to a halt. The vast majority of effort, by both industry and academia, has focused on homogeneous multicore systems.

We take the opposite view, namely that achieving high performance with limited power consumption will require a heterogeneous architectural approach in which there is a considerable degree of specialization of the hardware resources to the desired mix of applications. This is already evident to some degree with the addition of GPUs and fixed-function accelerators for cryptography. We believe this trend will accelerate. Achieving maximal performance will require exploitation of reconfigurable hardware and other very fine-grained resources – in addition to, rather than instead of, more conventional approaches like GPUs and homogeneous multicores.

The disadvantage of such heterogeneous systems is that without sufficient levels of abstraction and compilation technology, they will be very difficult to program. Currently, CPUs are often programmed in high-level managed languages like Java; GPUs are programmed with dialects of C; and reconfigurable hardware is programmed with hardware description languages like Verilog and VHDL.

We believe exploiting heterogeneous systems will require a language, compiler, and run-time system with the following properties:

- A single language that can be compiled to a highly diverse set of computational elements;
- The capability to express many different forms of parallelism so that those computational elements can be exploited in the manner best suited to them;
- A dynamic run-time system that is capable of subdividing programs to run across a variety of computational elements, and dynamically re-partitioning programs as they change or as the system application mix changes;
- A uniform semantic model that allows transparent migration between computational elements; and
- A language sufficiently familiar to achieve a wide degree of adoption.

In this paper we present Lime, a language designed to meet these requirements. We will describe the various features of the language and how they emerge from and attempt to address the sometimes conflicting needs of such vastly different kinds of computational elements.

Our implementation has so far focused on the most divergent kinds of computational elements: traditional processor

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

cores and reconfigurable hardware (field-programmable gate arrays or FPGAs). This has placed the maximal stress on the language design. While our implementation has so far emphasized semantic completeness over performance optimization, we present preliminary results for some representative applications that can be compiled into bytecode, C, or Verilog, and co-execute across a CPU/FPGA system. We have currently written and run over 40,000 lines of Lime code.

1.1 Key Lime Features

Before diving into the details of the language, we begin by providing an overview of the language features and how they address the requirements set out earlier.

Lime is a Java-based language with significant extensions. Using Java as a starting point provides both a fully machine-independent semantics and a dynamic execution model in which the run-time system adaptively recompiles the program as needed.

While abstract semantics and dynamicity are required to allow computations to be shared across divergent computational elements, effective compilation to hardware requires the ability to conveniently express statically known code, types, and sizes. To this end Lime provides unerased generics (which can be compiled in a template expansion style to maximize compile-time knowledge), ordinal types (integer ranges with a statically known number of elements), and bounded arrays indexed by (possibly parameterized) ordinal types.

The lack of side-effects and explicit pointers make a functional programming style well-suited to hardware. Lime adds both "micro-functional" and "macro-functional" features that integrate cleanly with an imperative, object-oriented style. The micro-functional features center around recursively immutable value types. The macro-functional features are based on isolated tasks that internally allow arbitrarily complex sequential imperative code, but can be modularly replaced with purely functional code.

Lime allows the convenient expression of various styles of parallelism. Fine-grained parallelism can be expressed down to the bit-level since bits are first-class objects in Lime, and "primitive" types with bit-level parallelism can be programmed directly in the language (e.g., 27-bit fixed-point numbers). Fine- and medium-grained data parallelism can be expressed with collective and reduction operations on both built-in and user-defined array and collection types. This includes operations as fine-grained as performing a "not" operation across an array of bits.

Medium- and coarse-grained pipeline and data parallelism can be expressed using new streaming primitives which compose individual tasks into computation graphs.

Streaming features are integrated into the language with the introduction of a *task* operator that converts a method into a component in a stream computation graph. A *connect* operator then allows rich graphs to be composed from tasks. It is the responsibility of the run-time system to handle buffering, partitioning, and scheduling of the stream graphs.

Many languages that introduce streaming or message passing have suffered from a "two paradigms" problem, in which programmers must decide early whether they will code a computation in "function style" or "stream style". For instance, in StreamIt [33] stream computations are written with explicit operators that dequeue and enqueue one or more objects for the worker computation; such computations can not subsequently be used in a context where function application is being used.

In Lime, the same method body can be used in either style (the distinction being whether the `task` operator is applied). Rather than provide explicit enqueue/dequeue operators, Lime has *tuples* and *bounded arrays* that can be used to express multi-input/multi-output operations, and a `match` operator is used to connect tasks with different data rates.

To expose functional properties and strong isolation within an imperative language, Lime relies on three key concepts: *valueness*, *localness*, and *sole-reference isolation*. Valueness or immutability is declared using the `value` keyword to modify a type declaration. Localness is declared using the `local` keyword which may be applied to any method declaration. It means that the method does not access mutable static fields (although it may read certain final static fields). Finally, sole-reference isolation means that only the task has a reference to the underlying object and hence any mutation of instance fields occur as a result of the instance itself.

These properties are easy to check and verify. Using these properties, Lime can classify the world of objects into tasks that are provably functional and those that are mutable but sole-reference isolated. These properties afford a great deal of compilation flexibility in realizing efficient parallel implementations that eschews data races and implicit nondeterminism.

1.2 Contributions

A language is much more than the sum of its parts, and this paper is primarily organized so as to give the reader an overview of the language as a whole. The language is based on Java, extended with a combination of well-known and novel features. The novel features described for the first time in this paper are:

- a simple, polymorphic, statically checked mechanism for preventing non-local side-effects (Section 3);
- the composition of the effect mechanism with value types to obtain referentially transparent imperative expressions (Section 3.1);
- the integration of stream-based computation into an object-oriented language (Section 6);
- the introduction of *rate matchers*, which combined with fixed-size arrays can express non-unit-rate stream com-

putations without requiring methods to be expressed in a different paradigm (Section 7); and

- description of the compiler and run-time system with preliminary results showing that programs can be co-executed across conventional processors and field programmable gate arrays (Section 9).

Language constructs for expressing micro-functional operations and bit-level parallelism, and their compilation to FPGAs were previously described in [21]. The full language is described in detail in the language reference manual [5].

2. Lime Language Fundamentals

Two of the biggest challenges in the design of Lime were (a) to provide convenient ways to expose many levels of parallelism and (b) to enable, encourage, and simplify the use of language constructs that can be used to generate the kinds of static structures required when compiling to hardware. The latter is particularly difficult in Java-like languages which tend to have the opposite goal.

Many fundamental language features are shown in the single example of Figure 1, a partial and simplified implementation of the built-in `unsigned` type. An unsigned number is defined as a fixed-size array of bits, and each operation is then defined as a function on those bit arrays.

Since much of the computation and parallelism available in hardware occurs at the bit-level, it is essential that Lime provide constructs for exposing such operations, and yet maintain a high level of abstraction. Thus the ability to compute at the bit-level is combined in Lime with the ability to define what in almost every other language are included as "primitive" types.

2.1 Value Types

One of the most fundamental aspects of Lime is the introduction of *value types*. An object of value type is *deeply immutable*; that is, all of its fields must be both unmodifiable ("`final`", in Java parlance) and must themselves be of value type. This is in contrast to the struct value types of C#, whose fields are immutable but may point to mutable objects.

Deeply immutable objects provide a host of beneficial mathematical properties (long espoused by adherents of pure functional programming) which are exploited extensively in the other features of the language *and* in its compilation. For compilation to hardware, Lime's value classes are essential since they can be freely moved across a chip as a chunk of bits, without requiring any remote accesses.

Greatly motivated by the needs of both hardware compilation and exposure of parallelism, Lime pushes Java towards a more functional style. Value types provide a "micro-functional" portion of the language. Streaming computation (see Section 6) provides the "macro-functional" portion of the language.

Value classes are defined by adding the keyword `value` to the class definition, as in line 1 of Figure 1. All fields of

```
1 public final value class
2  unsigned<N extends ordinal<N>> {
3    private bit[[N]] data;

4    public unsigned() {
5      this.data = new bit[[N]];
6    }

7    public unsigned(bit[N] data) {
8      this.data = new bit[[N]](data);
9    }

10   public unsigned<N> this & unsigned<N> that {
11     var result = new bit[N];
12     for (N i: N.first::N.last)
13       result[i] = this.data[i] & that.data[i];
14     return new unsigned<N>(result);
15   }
16   ...
17 }

18 public typedef uint = unsigned<enum<32>>;
```

Figure 1. Unsigned number implementation in Lime

a value class are implicitly `final`, and must themselves be value classes. A default constructor is auto-generated by the compiler if it is not explicitly defined.

Value classes have no reference identity: the equality ("`==`") operator recursively compares the equality of the fields, and the `hashCode()` method is a deterministic function of the values of the fields. Synchronization operations on value classes are statically forbidden, except when they are up-cast to `Object` or non-value `interface` types, in which case a dynamic check guards against the synchronizing on a value object by throwing an exception.

2.2 Ordinals

Ordinals are finite non-negative integral types, and are fundamental to the way statically sized and bounded objects are expressed in Lime. Despite their superficial resemblance to integers, ordinals are a special kind of enumeration type. Thus they are defined using the `enum` keyword: `enum<3>` is the ordinal type with members $\{0, 1, 2\}$.

Ordinals have successor and predecessor operations (written "`+++`" and "`---`", respectively), can be compared by order (e.g. with "`<`"), and can be added to or subtracted from each other (in which case the arithmetic is performed modulo the size of the ordinal).

2.3 Bounded Arrays and Value Arrays

In Lime, there are two different kinds of arrays: mutable arrays and value arrays. Lime's mutable arrays are similar to Java's arrays.

Value arrays, like value classes, are *deeply immutable*, meaning that the arrays elements may not be changed *and*

the element types must themselves be value arrays or value classes. Value arrays are written with a double-bracket notation. For instance, an unbounded value array of `int` elements is written as `int[[]]`.

An array with a statically fixed size is called a *bounded array*. Bounded arrays are heavily used in Lime, both to express fixed-size components of value classes, as well as to describe the input and output rates of stream operators (see Section 7).

A bounded array type is written with its ordinal bounding type inside of the square brackets as in `bit[enum<3>]`, which is a 3-element array of bits. When the bounding type can be expressed without using type variables, it is commonly abbreviated to just a constant expression, as in `bit[3]`. A bounded array is indexed by integer expressions that are statically confined to be within range. The two easiest ways to achieve this are (1) to use either integer literals or (2) to use expressions of the ordinal bounding type (which are implicitly turned into integers whose range is known). This leads to more reliable code and reduces the need for exception logic in hardware.

Bounded arrays are subclasses of unbounded arrays, and can thus be implicitly cast to their unbounded form. Thus a variable of type `bit[3]` can be assigned to a variable of type `bit[]`, and vice-versa if an explicit cast is used. In Lime, the unbounded array types are abstract; all arrays are actually instances of bounded arrays parameterized by their size.

Bounded and value arrays are used extensively in the definition of `unsigned` in Figure 1. The only field of the class (line 3) is a bounded value array of bits. In this case, the bounding type of the array is the generic parameter `N` (Lime generics are described in more detail below). There is also a constructor (line 7) which takes a mutable bounded array of bits, and uses it to create a new unsigned number. It uses the array constructor ("`new bit[[N]](data)`"), which inputs the mutable bounded array and returns an equivalent immutable bounded array, which is then used to initialize the newly constructed object.

Finally, bounded arrays are used in the definition of the "`&`" operator on lines 10–15. The bounded mutable array `result` is defined on line 11, and its elements are initialized on lines 12–13. Finally, on line 14, the constructor of line 7 is invoked to create a new `unsigned` value from the bits assembled in the `result` array.

2.4 Unerased Generics

As can be seen from our example, generics of fixed size are extremely important for the generation of hardware. Unfortunately, Java's implementation of generics uses *erasure*, in which all instances of a generic class are compiled into a single class in which the generic parameters have been erased. This has three implications: (1) the information needed by the hardware backend (or indeed by a highly optimized software backend) to generate specific types is lost; (2) safety is compromised, since erasure loses information, and (3) expressivity is lost, since any operation requiring the exact type of the generic parameter must be disallowed. For instance, the expression `new T[10]`, where `T` is a generic parameter, is not permitted in Java.

In Lime, generics are compiled without erasure, obtaining semantics similar to those of NextGen [28]. We rely on this property in Figure 1 in lines 5, 8, and 11, which construct new bounded arrays where the bounding type is a generic parameter. In addition, and unlike NextGen, Lime generics are capable of being parameterized by primitive types.

Compilation strategies for generics have typically suffered from a tension between time- and space-efficiency. Parameterization (where the type parameters are included as parameters to the methods) produces the most space-efficient code (since there is only one copy) but loses any opportunity for type-specific optimization. Template generation produces one instance of the generic class for each unique combination of type parameters in the source code; this maximizes opportunities for optimization, but can result in huge space overheads.

In Lime, we use a hybrid implementation strategy. In software, the default compilation is parameterized; in hardware, the default is templatized. However, where needed templatized compilation can be used in software, and in some restricted cases, parameterized compilation can be used in hardware.

2.5 User-Defined Operators

Lime allows programmers to define the behavior of standard unary and binary operators as they apply to new types. "Operator overloading" as practiced in C++ was deliberately avoided in Java since many felt its overuse resulted in confusing code. In Lime, however, the ability to define basic operators associated with arithmetic and logical operations is essential to the strategy of bringing object orientation down to the bit level and on a par with primitive types. As an example, the "`&`" operator for the `unsigned<N>` type, is shown on lines 10-15 of Figure 1. The following unary and binary operators can be user-defined:

```
~   !   +++   ---   +   -    *   /    &    |   ^   %
<   <=   >   >=   &&   ||   << >>   >>>   :: []
```

The operators `+++` and `---` are the successor and predecessor operators introduced in Section 2.2. and the range operator `::` is explained in Section 2.6. Lime does permit the array indexing operator "`[ ]`" to be redefined both for access (when not the target of an assignment) and for setting an element. The equality operators ("`==`" and "`!=`") may *not* be redefined in Lime. Value types have compiler-generated equality operators that check for recursive value equality. The language also prohibits the overloading of "exotic" operators such as the dot operator for method call

Certain binary operators implicitly define their corresponding compound operator. For instance, the "`+`" operator implicitly defines the "`+=`" operator. More subtly, the pre-

and post-increment/decrement operators are respectively defined by the successor and predecessor operators.

2.6 Ranges

Ranges are a convenience feature to support iteration over subranges of value types (assuming the type provides ordering operations). Examples include Lime ordinals and value `enum`s as well as the Java integral primitive types.

The expression `x::y` has the type `lime.lang.range<T>` where `T` is the least upper bound type of `x` and `y`. The type `range<T>` implements the `Iterable<T>` interface from Java, but is itself a `value` type. Ranges are thus usable in the "for-each" style loops introduced since Java 5 and also as values in their own right.

Ranges are particularly useful in conjunction with the constructs `.first` and `.last` which resemble fields and apply to all bounded types (most notably, ordinals). Unlike ordinary fields, the `.first` and `.last` "special selectors" (also `.size`) are dispatched virtually through type variables. An example is line 12 of Figure 1, which loops over the range of the ordinal `N` which is a type parameter.

Lime also provides a special shorthand when the iteration is over the entire range of a value type. For example, the loop on line 12 of the example could be written "`for (N i)`" (with the limits `N.first` and `N.last` implied).

2.7 Typedefs

Given that Lime permits the definition of new "primitive types," it is convenient to be able to give those types short intuitive names, especially when the types are defined using generic types which tend toward verbosity. Thus, Lime provides `typedefs` similar in flavor to what are offered in the C language. An example is shown on line 18 of Figure 1.

Semantically, a type definition is just one step above a lexical macro and is substituted for the defined symbol prior to any other semantic analysis. However, the scoping and visibility for typedefs follows that for class definitions. Thus, the typedef in the example is `public` just as `unsigned<N>` is public (as a result, it actually needs to be in a source file of its own). Typedefs at package scope, class scope, and local scope are also possible.

2.8 Local Type Inference

Lime programs can make heavy use of the expanded generic types. Parameterized types tend to be verbose, and some of the types introduced by task programming are especially verbose (see Section 6.3). As a compensating convenience, Lime provides a limited form of local type inference to avoid the need to present verbose type declarations twice. The pseudo-type `var` in a field or local variable declaration causes the type to be inferred from the initializer. When the variable or field being defined is also `final` one simply uses `final` rather than `final var`.

An example is shown on line 11 of Figure 1 where the `var` keyword is used to avoid having to type `bit[N]` twice on the same line. Inference is limited to these contexts and there is no attempt to infer types across multiple expressions.

2.9 Java Compatibility

A key design decision in developing Lime has been to make it Java compatible. Most legal Java programs become legal Lime programs without change. *All* legal Java programs can be imported as Lime programs by a purely syntactic transformation that is extremely non-intrusive and trivially performed by a development tool. Lime reserves twelve additional keywords. These identifiers, if present in a Java program, can continue to be used by escaping them with a "`" (backtick) character. Generic types and methods have an expanded semantics in Lime, but the original Java semantics can be obtained by use of the "~" (tilde) character. Similarly, arrays have been enhanced in Lime but one can escape back to Java arrays with a tilde. Usually the tilde escape proves unnecessary and can be removed, conferring some Lime advantages immediately. Then, Lime language features can be gradually added making the program more readily parallelized by the compiler. Lime programs are also binary compatible with Java in that they can freely invoke Java methods; if Lime-specific types are avoided in public signatures, then Java code can call back into Lime.

3. Local Method Checking

The goals of Lime require a classification of methods into those that are pure functions, those that are stateful but isolated (receiving new information only as arguments and providing results only as return values), and those that have arbitrary side-effects. For modular compilation and efficient implementation, this classification should be decidable by examining single methods without computing a call-graph or similar structure and in the presence of inheritance and virtual dispatch.

More precisely, a stateful isolated method is an instance method that may read and mutate all information reachable from the "`this`" instance. It may not read or mutate any other information, thus excluding all information reachable from `static` fields of any class. It may read, but not mutate, argument information. It may not retain a reference to its return value or anything reachable from it.

We rejected the idea of translating these properties directly into checkable keywords like `pure` or `isolated`. While both properties are easy to state as requirements, enforcing them as unitary properties in an imperative language would require complex checking. To ensure that a method is pure in the absence of an independent immutability property means analyzing all of its actions in context (what the method is doing and which objects it is doing it to). To ensure that a method is stateful but isolated is similarly difficult since one must enable one large class of accesses while disabling another large class which will break isolation.

Lime substantially simplifies and modularizes the required checking for both pure functions and stateful isolated methods by decomposing both of these properties into two simpler properties. The first is *valueness* which is already captured by the Lime value types (see Section 2.1). Valueness is useful in its own right and can mostly be checked by examining field declarations and by directly generating the code for equality checking (constructor code is also checked to prevent exposure of uninitialized `final` fields). The second property is *localness*, which simply means that no side-effects can be propagated through static fields or information reachable from them. In the presence of valueness, localness is readily checked method-by-method, examining only field accesses and method calls. Two different compositions of valueness and localness yield the desired purity and isolation properties as explained below.

The method modifiers for localness are called `local` and `global` (explained next) and `glocal` (explained in Section 3.2). A method labeled `local` must not write any static fields and must not read any static fields except *repeatable static fields* as defined in Section 3.1. A `global` method may freely access all static fields.

By default, the instance methods and constructors of value types are considered to be `local`, while all static methods and the instance methods and constructors of non-value types are considered to be `global`. The defaults may be overridden by using the explicit keywords.

A `local` method may only invoke other `local` methods. A `global` method may invoke either `local` or `global` methods. A `local` method may not be overridden or implemented by a `global` method. These rules together ensure that the property asserted by `local` can be checked without recursively investigating callees, and without knowing the complete inheritance hierarchy.

To ensure full checking, `local` and `global` apply equally to concrete and abstract methods (including the methods of interfaces). In Figure 2, the interface `IRandom` provides a single method, `nextInt()`, which generates a random integer. `nextInt()` is declared `global` since that is the most permissive qualifier. There are two implementations: `Random` and `PseudoRandom`. The first implementation calls `nanoTime()` and therefore accesses global system state, and so the method is also declared `global`. However the second implementation, `PseudoRandom`, uses a (very primitive) deterministic function (for the purpose of illustration) to generate pseudo-random numbers; therefore, its `nextInt()` method is declared `local` and can be called in any context.

Recall that one motivation was to determine when methods are pure functions. The rules for determining this property are as follows:

1. A `local static` method whose arguments are all value types is necessarily a pure function. It cannot communicate through static fields, it has no `this` argument, and its explicit arguments have no mutable instance fields.

```
 1 public interface IRandom {
 2   global int nextInt();
 3 }

 4 public class Random implements IRandom {
 5   public global int nextInt() {
 6     return (int) System.nanoTime();
 7   }
 8 }

 9 public class PseudoRandom implements IRandom {
10   int nextValue = 17;

11   public local int nextInt() {
12     nextValue = nextValue * 317 + 9;
13     return nextValue;
14   }
15 }

16 public final value class
17  unsigned<N extends ordinal<N>> {
18    ...
19    static glocal unsigned<N> random(IRandom g) {
20       return new unsigned<N>(g.nextInt());
21    }
22 }

23 public class BoundedMap<K extends ordinal<K>, V> {
24   protected final V[K] mapping = new V[K];
25   ...
26   public glocal boolean containsValue(V val) {
27      for (K key)
28         if (val.equals(mapping[key]))
29            return true;
30      return false;
31   }
32 }

33 class Test {
34   static final boolean repeatable = hasFoo();

35   static local boolean hasFoo() {
36     var g = new PseudoRandom();
37     var u = unsigned<32>.random(g);
38     var map = new BoundedMap<enum<20>,
39                              unsigned<32>>();
40     map[enum<20>.random(g)] = u;
41     return map.containsValue(u);
42   }
43 }
```

Figure 2. Example of Local, Global, and Glocal Methods.

2. A `local` instance method of a value type whose arguments are all value types is a pure function. Although there is a `this` argument, that argument is a value too.

A second motivation was to determine when methods, though stateful, are isolated. This requires controlling the

reachability of objects from instance fields. The desired property can be computed by first defining an *isolating constructor* as follows: a constructor is *isolating* if it is `local` and has only value arguments. Instances that are created with isolating constructors are guaranteed to have no aliases initially stored in their instance fields. We can then guarantee that a method is stateful but isolated *with respect to a particular instance* if (1) the instance was created with an isolating constructor, and (2) the method has only value arguments and a value return.

A pure function may return non-values (since they cannot also store a reference to what they return) but returning a non-value from a stateful isolated method could break isolation by creating an alias. Stateful isolation with respect to a particular instance is useful when we can guarantee that a task holds the only reference to the instance. This final guarantee is quite easy to achieve as described in Section 6.2.

3.1 Repeatable Static Fields

Recall that a `local` method may read certain `static` fields deemed to be "repeatable." The concept of a repeatable expression is thus an extension of the concept of a compile-time constant expression (which in Java is limited to primitive types). Repeatable expressions allow complex initialization of any value type with complex imperative code.

A static field is repeatable if it is recursively immutable (i.e., a `final` field of value type) and all accesses to it during the lifetime of the computation produce the same result.

The recursive immutability property is clearly needed to avoid introducing new opportunities for side-effects. Repeatability is added to avoid the subtle requirement that would otherwise exist that all `static final` fields that are read by `local` methods must be initialized before those methods can be allowed to execute. It eschews any non-determinism due to class loading order, and it makes it substantially easier to relocate computations to hardware elements that lack a distinguished "class initialization" phase.

In Lime, the only sources of non-determinism in a single thread are limited to `global` methods, such as `System.nanoTime()` in the example in Figure 2. So, the repeatability property is obtained if the following properties hold in addition to recursive immutability.

1. The field must have an explicit initializer.
2. Its initializer must be an expression that would be legal in a `local` method.
3. The field's initializer may not refer directly or indirectly (via other static variables) to the field itself (this would make the value dependent on the order in which classes are initialized).

3.2 Local/Global Polymorphism

It was said earlier that abstract methods (and more generally, methods "high" in the class hierarchy that are intended to be overridden) must commit to being either `local` or `global`. This is not quite true: Lime provides a third option, motivated by the following observation. The fact that `local` can override `global` but not vice versa would tend to favor using `global` for abstract or frequently overridden methods. But, the fact that `global` can call `local` but not vice versa favors making methods `local` whenever feasible so that they can be used in more contexts. Thus, a user designing a class hierarchy to be used in multiple contexts has a conflict if only those choices exist.

In fact, abstract methods which have, as yet, no actual behavior, can later be implemented in either a `local` or a `global` fashion, as in the `IRandom` class hierarchy of Figure 2. But, simply declaring a "neutral" category would do no good by itself, since a conservative analysis would have to preclude calling any "neutral" method in a `local` context.

Lime solves this problem with a context-dependent category, called `glocal`, with rules and a supporting static analysis that allows many calls to `glocal` methods to be treated as `local` calls. This promotes the more extensive use of `local` methods without compromising efficiency or re-usability.

Our solution is based on the observation that, for many methods, whether the method accesses global state is a function of its input arguments. For instance, Figure 2 lines 16–22 show the `random()` method of the `unsigned` type. It is a `glocal` method that accepts a random generator of type `IRandom`. On line 37, `unsigned`'s `random()` method is invoked with a parameter of type `PseudoRandom`, all of whose methods are `local`. Therefore, we can treat the `random()` method as `local` in the calling context at line 37.

Lime's `glocal` modifier exploits the greater information available at call sites by defining *localizing calls* and defining `glocal` methods so that calls to them are *localizable*.

At the call site, a `local` method may only call a `glocal` method when all of its parameters are *localizing*. An actual parameter localizes a formal parameter when all of the accessible non-final instance methods of the formal parameter are overridden to be `local` in the static type of the actual parameter. When used in combination with generics, a call to a `glocal` method is localizing only when all of the generic and method parameters are localizing.

At the method definition, a `glocal` method has the same restriction as a `local` one in its access to static fields. In addition

1. calling `final global` methods is prohibited (such a call can never be localized);
2. calling non-final `global` methods is prohibited except when (a) the receiver is a parameter of the method being defined, or (b) its type is determined by a type variable (only calls indirected in such a fashion can hope to be localized);
3. calling other `glocal` methods is only allowed when it would be allowed for a `global` one by the previous rule *or* when such a call is *localizing*;

4. a `glocal` method may not override a `local` method and a `global` method may not override a `glocal` one (this ensures that the strongest necessary check is always performed at call sites).

In Figure 2, `unsigned.random()` is a valid `glocal` method by rule 2(a): it calls a non-final `global` method of its parameter `g` of type `IRandom`. The call on line 37 is localizing because all methods of `PseudoRandom` are `local`.

The locality-polymorphism provided by `glocal` is also essential for generic classes, as shown in the fragment of the definition of the class `lime.util.BoundedMap` on lines 23–32. This class implements a mapping where the key is of an ordinal type, allowing a very efficient implementation with a fixed-size bounded array. The values held by the mapping, on the other hand, are of unconstrained type (type `V`).

Consider the implementation of the `containsValue()` method on lines 26–31: it iterates over the `mapping` array checking whether it contains the passed parameter `val`. It does this by calling the `equals()` method, which, since `V` is of unconstrained type, is considered to be `Object.equals()`. Although it is considered good style for `equals()` to be a `local` method, we do not constrain it as such (in particular, doing so would break Java compatibility). Thus the call to `equals()` on line 28 is not `local`, but it is `localizable` by rule 2(b). The call to `containsValue()` on line 41 is localized because the actual type of the generic parameter `V` is `unsigned<32>`, which is a `value` type whose `equals()` method is therefore guaranteed to be `local`.

The analysis that supports `glocal` is conservative in two ways. First, the simple name of the parameter must be the receiver of the call. Second, all accessible non-final instance methods of a substituting type must be `local`, regardless of which of those methods are actually called. Because calls can occur in subroutine methods as well as the one being analyzed, the additional analysis to support greater precision would be complex and non-modular, but more to the point, the reasoning required on the part of the programmer to understand what is safe and what isn't would likely be quite challenging. Over time, experience with the language may lead us to attempt a more precise definition and analysis.

To maximize reuse, it is highly desirable that classes be usable in a `local` method, and as localizing parameters. In particular, the three overridable public methods of `Object` – `equals()`, `hashCode()`, and `toString()` – should be `local` whenever possible. Since a value type causes all of these methods to become `local`, substituting a value type for the `Object` type is always localizing (e.g., `BoundedMap`).

For non-values, the programmer must take care to ensure the same property holds wherever possible. The easiest way to do so is to have a non-value class extend `lime.lang.Mutable` rather than `Object`. The former provides `local` implementations of the three key `Object` methods. The `equals()` method simply uses the "`==`" method on its arguments (checking for object identity); the `toString()` method returns the class name appended with an "`@`" and the object's hash code, and the `hashCode()` method produces a hash code computed entirely from the immutable values of the fields of the object.

Note that the default `Object.hashCode()` method, as well as `System.identityHashCode()`, are `global` methods. Because their values change unpredictably with each instance of an object, they can not be used to create repeatable static fields. On the other hand, `Mutable`'s `hashCode()` method can be used in computing a repeatable static.

4. Collective and Data-Parallel Operations

Lime encourages programmers to use collective operations by making them convenient and general. They are not inherently data parallel in that there is no explicit prohibition against using collective operations with mutable types where there are dependencies between elements. The semantics are defined as if there was an implicit iteration (in order) over the subject collections. However, collective operations applied in the presence of the value and local properties will reveal parallelism to the compiler without the need to analyze loops or perform interprocedural analysis.

Collective operations (indicated by "`@`") may be applied to operators, instance methods, and static methods. On infix and prefix operators, the "`@`" precedes the operator. On instance method calls, the "`@`" takes the place of the "`.`" in the method call. On static methods, the "`@`" appears next to a specific argument which is to supply the `Collector` as explained below.

The "`&`" operator defined in Figure 1 may actually be written more compactly as follows:

```
public unsigned<N> this & unsigned<N> that {
  return new unsigned<N>(this.data @& that.data);
}
```

The "`@&`" pair is like a "map" operation in that it extends its subject operator ("`&`") to apply (in this case pair-wise) between all the elements of an array or collection.

Collective operations may be used with methods of any number of parameters, and the parameters may be a combination of either collections of the parameter type, or single instances of the parameter type. In the latter case, the parameter expression is evaluated exactly once and used as the argument of every individual operation. For example:

```
string[] v1 = { "x", "y" };
string[] v2 = v1@toUpperCase(); // v2=={"X", "Y"}

int[] a1 = { 1, 2, 3 };
int[] a2 = a1 @+ 1; // a2=={ 2, 3, 4 }
int[] a3 = a1 @+ a2; // a3=={ 3, 5, 7 }
```

If the arrays are not of the same size, a domain conformance exception is thrown. Lime's array types and the collection types supplied by Lime's development library all implement the appropriate interfaces to ensure participation. The interfaces are `Indexable` and `Collectable`. The

`Indexable` interface specifies that the collection has some index value type and an indexing operator that returns an object of the element type. Furthermore, it has a `domain()` method which returns an object that allows iteration over all of the valid indices of the objects contained in the collection.

The `Collectable` interface extends `Indexable` with a method that returns a *collector*. A collector is an object that is used to gather the results together to produce the new collection resulting from the operation. A collector must implement the `Collector` interface which provides an indexed store operation to allow the individual results to be collected, and a `result()` method that returns a new collection[1] which must also be `Collectable`.

Data-parallelism. A collective operation is data parallel if the elements are all values and the collector meets the stateful isolation requirements described in Section 3. The compiler may be able to determine other data parallel cases, but that is more dependent on the use of various optimizations that were not the specific focus of this language feature.

Reductions. Lime also provides facilities for performing *reduction*, namely applying a binary operator, instance method or static method across the elements of a collection to generate a single result of the element type.

Reduction can be applied to any `Iterable` class. The method must take two arguments of the same type and produce a result of that type. Thus an instance method of a class `T` must have the signature `T foo(T)` and a static method must have the signature `T foo(T,T)`. A reduction is indicated by the use of `@@` as the examples below illustrate.

```
int a = { 1, 2, 3 };
string[] fooletters = { "f", "o", "o" };

// reduction with binary operators:
int sum = @@+ a; // sum==6
string foo1 = @@+ fooletters; // foo1=="foo"

// reduction with instance methods:
string foo2 = fooletters@@concat; // foo2="foo"

// reduction with static methods:
int max = Math.max(@@a); // max==3
```

5. Enabling Features for Stream Computing

Tuples. Lime's model of stream computation is based on turning methods into compute nodes in a graph. This implies a way to turn the multiple arguments of a method into a single structure that can be transmitted. A similar need exists on the return side. Both goals are accomplished by Lime's *tuple types*. Tuples are shallowly immutable collections of heterogeneous objects.

A tuple type is written as a comma separated parenthesized list of types with an initial backtick. Tuple expressions are similarly written. For example, `` `(3, "foo") `` is a tuple expression of tuple type `` `(int, string) ``. A tuple type is a value type if and only if all of its member types are value types since the tuple type itself only guarantees shallow immutability.

Tuple types are anonymous and defined structurally. That is, two tuple type declarations that employ the same types in the same positions are the same type and will interoperate.

The elements of a tuple can be accessed using a dot (".") followed by a 0-based integer literal indicating its position in the tuple. Projection of multiple fields may also be more succinctly expressed with a left-hand tuple of variables. Element-wise type compatibility and widening is provided for tuples. These features are shown by the following:

```
var triple = `(1, 3.14159, "blah");
double pi = triple.1;   // project one field
`(a, b, c) = triple;    // project all fields
`(double, String) x = `(1, "as"); // 1 is widened
`(long, Object) y = `(1, "as");   // "as" is upcast
```

Closed World. In a language designed both to be Java compatible and to support direct execution in hardware there is a tension between the Java development model (separate compilation, dynamic class loading, no "closed world" assumption) and the requirements of hardware (it is impractical to support dynamic class loading in an FPGA).

Our strategy for generating hardware-related artifacts necessarily employs some closed-world assumptions. With one notable exception, we accomplish these by extra linguistic means: we require that all Lime source and object artifacts be stored in a directory structure similar to the Java classpath. We support version-based invalidation and regeneration of hardware artifacts in our toolchain.

To avoid dynamic class loading in hardware, one must be able to explicitly seal parts of a class hierarchy so that the subtypes of a given class is finitely enumerable during hardware synthesis. The necessary support takes the form of an `extendedby` keyword, which follows the standard Java `extends` and/or `implements` clauses of class and interface declarations.

```
public interface Animal extends Comparable
    extendedby Cat, Dog { ... }
public final class Cat implements Animal {  ... }
public class Dog implements Animal
    extendedby Hound, Poodle { ... }
```

When an `extendedby` keyword is present on a type, the following consequences follow.

1. Exactly the types listed must exist and must directly extend or implement (as appropriate) the present type.
2. No other types may extend or implement the present type.

It is expected practice that a type named in an `extendedby` clause but that does not itself have an `extendedby` clause will be `final`. If this practice is not followed, the compiler will issue a warning that the type extension subhierarchy is not closed. Otherwise, hardware synthesis can proceed in a "closed world" fashion by exploiting *closed types*, which are

[1] It is usually of the same type as the original collection, but need not be.

types that are either `final` or `extendedby` a list of types all of which are closed.

It is never an error to write a Lime program that does not use `extendedby`. If used where appropriate, portions of the program will have the necessary closed world property to increase the likelihood of efficient synthesis to hardware.

6. Task Programming Model

Lime offers language features for expressing task, data, and pipeline parallelism. They are based on the creation of dataflow graphs that perform computation on streams of data. The approach exposes algorithmic data-locality and communication topologies to a compiler that can then decide on the best implementation choices depending on the target platform. A dataflow graph, also known as a stream graph, consists of nodes that perform computation and edges that imply an exchange of data between connected nodes.

In Lime, nodes are *tasks* which read data from an input *port*, apply a worker method to the data, and commit the results to an output *stream*. A task's worker method is applied repeatedly as long as there are input data available on the port. A connected set of tasks form a *closed world* if the types that can enter them from the outside (via initialization or via ports) are all closed types and there is no explicit reflection (e.g., `Class.forName`) in any of the worker method.

An example stream graph is illustrated in Figure 4. It describes the processing steps required to decode a JPEG image and convert it to an RGB bitmap. There are four processing stages: *bitstream parsing*, *channel decoding*, *color space conversion* and *descrambling*. The first stage parses an encoded string of bits to extract the decoding properties and the sequence of macroblocks comprising the image. A macroblock (mb) represents an 8x8 block of pixels from a particular color channel as a 64 pixel array. There are three color channels, one for luminance (Y) and two for chrominance (Cb and Cr). The parser produces a sequence of n macroblocks arranged as

$$(mb_1^Y, mb_1^{Cb}, mb_1^{Cr}) \ldots (mb_n^Y, mb_n^{Cb}, mb_n^{Cr}).$$

Each of the macroblocks is channel decoded to reconstruct the original image[2]. The channel decoding consists of five steps: *zigzag decoding*, *DC coefficient decoding*, *inverse quantization*, *inverse discrete cosine transform*, and *value centering*. Each of these steps is a function $f : block \rightarrow block$ where `block` is a bounded value array of 64 `pixel` values (i.e., `pixel[[64]]`). We can summarize the channel decoding using function composition as

$$center \circ iDCT \circ deQuantize \circ dcDecode \circ zigzag$$

which in an imperative programming style may be coded as shown in Figure 3 lines 11-18. The `decode` method applies the five transforms sequentially for every macroblock. The

[2] JPEG encoding is usually lossy hence the reconstruction is approximate.

```
 1 typedef pixel = unsigned<24>;
 2 typedef block = pixel[[64]];

 3 public class Channel {
 4   static  final Decode d = new Decode();
 5   private final Coefficient dc;
 6   public  final Quantization q;

 7   public Channel(Color c) {
 8     dc = new Coefficient();
 9     q  = new Quantization(c);
10   }

11   public block decode(block mb) {
12     return d.center(128,
13               Transforms.iDCT(
14                 q.deQuantize(
15                   dc.dcDecode(
16                     d.zigzag(mb)))));
17   }
18 }
```

Figure 3. Lime pseudocode for JPEG decoder in an imperative programming style.

transforms are stateless except for `dcDecode` which decodes the first pixel of each macroblock relative to the value of the corresponding pixel in a previous macroblock. Each of the transforms is carried out by a *l*ocal method that produces a value that is consumed by another method. The composition of methods in this way lends itself well to pipeline parallelism where each of the five methods is concurrently operating on different (but successive) macroblocks in the same color channel as illustrated in Figure 4. However, the introduction of pipeline-parallelism in an imperative language is intrusive, requiring the incorporation of buffering stages between method-calls or the use of other schemes that ultimately obfuscate the program. This in turn leads to rigid implementations that are difficult to modify or tune for performance. In addition, the language semantics in an imperative programming model ensure eager evaluation and the resultant code dictates a specific execution order or schedule that may not be easily ported between architectures.

The JPEG example also affords opportunities for data parallelism. Data parallelism is readily apparent from the stream graph: each color channel decoder may operate on its corresponding macroblocks independent of the other decoders. In this case, threads could have been used to achieve coarse-grained parallelism. However, threads are often too heavy-weight an instrument for this purpose. In general it is better to let the compiler or an independent scheduler decide how to assign computation to threads.

Lime overcomes these challenges by introducing language features that enable lazy evaluation, encapsulate computation, abstract away communication, and provide strong isolation guarantees that make it feasible to realize scalable

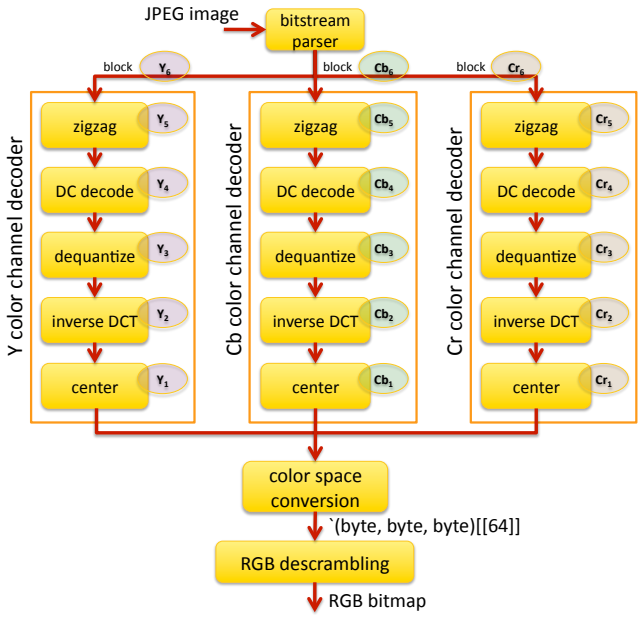

Figure 4. Pipeline and data parallelism in JPEG. Macroblocks are illustrated as circles. Each macroblock is labeled with the color channel it belongs to. Pipeline parallelism here refers to the concurrent processing of different macroblocks in a single color channel, and data parallelism refers to the concurrent processing of macroblocks from different color channels so while the left most pipeline is processing the Y blocks, the middle pipeline processes the Cb blocks and the right most pipeline processes the Cr blocks.

parallelism and to synthesize Lime programs into custom hardware. The subsequent sections both explain task programming in some depth and return repeatedly to the JPEG decoder example to show how the task programming features apply to that program.

6.1 Task and Connect

Lime facilitates the expression and discovery of coarse-grained parallelism by introducing the `task` and "`=>`" (connect) operators. These features free the programmer from specific implementation and performance details, and empower the compiler (and language runtime) with far more freedom in realizing high-performance executions for a wider range of architectures.

The `task` operator creates an execution entity that reads data from an input *port*, applies a *worker* method to the data, and commits the results to an output *stream*. The port type is derived from the worker method's parameter list. Similarly, the stream type is the return type of the method[3]. The port is a tuple type if the number of arguments is greater than one, otherwise it is the type of the method's single argument.

[3] Methods with empty parameter lists or void returns can be used to create sources and sinks, as described later.

The "`=>`" (connect) operator connects the stream (output) of one task to the port (input) of another. A connection between tasks can viewed as a double-ended queue with a task writing at one end and another reading at the other end. The language forbids connections between ports and streams that are already connected to avoid common programming errors or scheduling complications that may arise from simultaneous connections. This prohibition sometimes arises in hardware description languages as well, for example when a wire is connected to multiple sources which can affect the value read from that wire.

A connection between tasks also requires that the receiving port is of the same or wider type than the providing stream. If the types do not quite match but are compatible, Lime provides an operator to *match* the data types as detailed in Section 7.

The JPEG channel decoder pipeline for a single color channel can be expressed as follows in Lime.

```
task Decode().zigzag
  => task Coefficient().dcDecode
  => task Quantization(component).deQuantize
  => task Transforms.iDCT
  => task Decode().center(128, block);
```

The `task` operator constructs a new task whose worker method is bound to the method specified after the dot. The task worker method is invoked repeatedly as long as data are available on the task port. Methods in Lime are agnostic to their eventual use: a method may be used as a worker method in a task, or as a method invoked from an instance object. One can think of the `task` operator as supplying a system wrapper method that is aware of the task port and stream. The wrapper reads a number of data items from the input port, invokes the intended method using the appropriate parameters, and writes the returned results to the output stream. The three ways of applying the task operator are illustrated above.

Static Methods. In general, the `task` keyword may be followed by the name of a static method, qualified by its signature to make the reference unambiguous. `Transforms.iDCT` is a task created from a static method.

Non-value Instance Methods. Tasks with mutable state are created from non-value classes that provide an *isolating constructor*. The tasks constructed from the `dcDecode` and `deQuantize` methods are examples of stateful tasks. The `task` operator in this case performs the equivalent of a `new` operation on the class using the isolating constructor, and then instantiates a task that executes the specified method for each value supplied on its input port. The task holds the *sole* reference to the object thus assuring that state is not shared between tasks. This eschews data races and non-determinism when executing tasks concurrently. The task creation expression for stateful tasks must name both the isolating constructor and its arguments if any, as well as the instance method to be used, all in a single expression. Thus, `task Quantization(component).deQuantize` must be thought of

as a single grammatic production that cannot be broken up. Attempting to create a task from an already-instantiated object using a method that is not isolated as in the following is a compile-time error.

```
Quantization dq = new Quantization(component);
... => task dq.deQuantize // error
```

Value Instance Methods. Any isolated instance method of a value class *may* be used to create a task. The rules for task construction from value instance methods are the same as those for task creation from static methods except that the method or operator is associated with a specific value instance, as in the following.

```
final Decode d = new Decode();
...
task d.zigzag
  => ...
  => task d.center(128, block);
```

The example above also illustrates constant task parameters, namely the pixel value 128 is bound as a constant first parameter to the `center` method. The missing parameters are specified by their type, and the supplied parameters are specified as expressions.

For generality, we allow the form of `task` expression that is required for non-value instance methods to be used for isolated methods as in the example (`task Decode().zigzag`). This form of task construction also permits operators including user-defined operators of user-defined value classes and primitive types. The operator must be one that is legal for the value, either because the value is of primitive type, or because it is a user-defined value type that implements the operator. The set of operators that can be used in this way is exactly the set of normally immutable operators that may be user defined (Section 2.5) except indexed assignment.

Task and connect expressions are a convenient and natural way of describing dataflow computation, compared to the imperative function composition paradigm. The former allows for a textual representation of the dataflow graph that matches the graphical representation whereas the latter does not. Another worthy property of task and connect expressions is that they simply direct the construction of dataflow graphs, but they do not cause any evaluation of the computation to occur. That happens when tasks are started as described in Section 6.5. This two step process of establishing a dataflow graph and then later starting the evaluation is attractive because it decouples the expression of parallelism from its implementation, allowing not only the compiler but also the runtime to transform and refine the input dataflow graph into implementations suitable for the intended architecture. Just as function inlining reduces function call overhead and increases the opportunities for instruction level parallelism, task fusion—where pairs of connected tasks are fused into one—can lower communication costs and lead to more load-balanced implementations that are more efficient to run [14, 15].

6.2 Filters and Isolation

Tasks with exactly one port and exactly one stream are called *filters*. User filters are constrained to produce one output value for each input value, but there are system filters that are not so constrained. The tasks that perform the channel decoding for a single color channel comprise a filter of port and stream type `block`. Filters have the type `Filter<Tin extends Value, Tout extends Value>` and hence we can define a filter-generating method equivalent to `Channel.decode` (Figure 3 lines 11-17) as shown.

```
Filter<block, block> decodeFilter(Color c) {
  return task Decode().zigzag
      => task Coefficient().dcDecode
      => task Quantization(c).deQuantize
      => task Transforms.iDCT
      => task Decode().center(128, block);
}
```

Filters and most but not all tasks have the property that they are *isolated* from the rest of the system in that they only observe *values* via their ports and only generate *values* via their streams. Therefore, most tasks can not perform side-effects on other tasks or on the program heap (the exceptions are sources and sinks, which are discussed in Section 6.4).

Filters satisfy one of two isolation properties: *inherent isolation* or *sole-reference isolation*. An inherently isolated task is constructed from a pure function[4] whose return type is either a value or `void`. Such tasks are stateless by definition. A stateful task on the other hand must be sole-reference isolated. It is constructed from a stateful isolated method as defined in Section 3 by adding the constraint that the task must hold the sole reference to the isolated instance.

Sole-reference isolation works even if the class defines `global` instance methods, because the local method will (by definition) not call global methods and isolation assures that no other code will either. Sole-reference isolation also works if the class has mutable `public` fields because, even if the chosen instance method accesses such fields, no other code will. Since the object starts with no aliases, the task holds the sole reference, and non-values cannot enter the worker method, it follows that no new aliases can be created.

6.3 Split and Join

In addition to filters, Lime provides `split` and `join` system tasks to facilitate the expression of task and coarse-grained data parallelism. The split task, also known as a splitter, transposes a one-dimensional stream into a multi-dimensional one. Specifically, it inputs a stream of tuples and outputs the members of each tuple on the members of a tuple of streams. Similarly, a join task or joiner individually inputs values from the members of a tuple of streams and outputs a stream of tuples formed from those inputs.

The JPEG decoder offers an opportunity for data parallelism amongst the three color channel decoders: if the

[4] Recall from Section 3 that Lime recovers pure functions by composing valueness and localness.

stream of macroblock tuples produced by the bitstream parser is transposed to produce three separate streams, then it is possible to apply a separate channel decoder filter to each stream.

```
 1 var parser   = ...; // task not shown
 2 var filterY  = decodeFilter(Color.Y);
 3 var filterCb = decodeFilter(Color.Cb);
 4 var filterCr = decodeFilter(Color.Cr);

 5 var splitter = task split `(block, block, block);
 6 var joiner   = task join  `(block, block, block);

 7 parser
 8   => splitter
 9   => task [ filterY, filterCb, FilterCr ]
10   => joiner
11   => ...
```

In the example code above, lines 2-4 create the three independent filters, and lines 5-6 define the tasks that transpose the macroblock streams. The overall task graph can then be assembled using these building blocks as shown on lines 7-11. The splitter produces three separate streams each of type `block`. It is connected to a compound task consisting of the three decoding filters. The compound task operator `[...]` constructs compound tasks from simple task types (i.e., task with one port and one stream). The compound task created on line 9 is such that each filter port connects in order to one of the splitter streams, and each of the filter streams connects in order to a joiner port. The output of the joiner is again a stream of `‘(block, block, block)`.

Splitters and joiners are reified by the following types respectively `Splitter<Tin extends Value, Touts>` and `Joiner<Tins, Tout extends Value>`. These make it possible to create and save intermediate components of a task graph or to write utility methods that generate them.

Instances of these types are created by the `task split` and `task join` operators as shown in the example. The argument to these operators must be a *splittable type*, meaning a value type that is either a tuple or a bounded value array. The `Touts` type of a splitter and the `Tins` type of a joiner are derived such that a tuple type is split to a tuple of streams or joined from a tuple of ports, and a bounded value array type is split to a bounded (non-value) array of streams or joined from a bounded array of ports. Writing out the exact `Splitter<...,...>` or `Joiner<...,...>` types is tedious and normally avoided by using `var`.

6.4 Sources and Sinks

In addition to filters, Lime allows for `Source` and `Sink` tasks which need not be isolated. Such tasks are necessary for performing I/O and globally side-effecting operations.

The simplest form of a (source) task is created from a simple value using the `task` operator followed by the `value` keyword and an expression that evaluates to a value (e.g., `task value bit.zero`). Such a construction results in a source task that produces an infinite stream of that value. Source tasks are parameterized types `Source<Tout extends Value>` and may be constructed directly from a value or from methods with no parameters. Similarly, sinks are created from methods with void returns and have the type `Sink<Tin extends Value>`.

When a source or a sink is isolated, the `getWorker()` method can be used to access the internal state of a task when it is no longer running. The method returns the object whose instance method is the task worker (or null if the work method is static) but only if the task is no longer running. An example of a sink task in JPEG decoder is the RGB descrambler which comprises the final stage of the decoding process. The descrambler writes the decoded macroblocks into a two-dimensional array of RGB pixels.

```
 1 typedef rgb = `(byte, byte, byte);

 2 class RGBDescrambler {
 3   public rgb[][] bitmap;
 4   public local void descramble(rgb[[64]] mb) {
 5     ...
 6   }
 7 }

 8 Sink<rgb> sink =
 9      task RGBDescrambler(...).descramble;
10 ...
11 rgb[][] bitmap =
12      ((RGBDescrambler) sink.getWorker()).bitmap;
```

The `getWorker` method is used to access the results of the computation, namely the RGB bitmap in this case. Without such a method, the `descramble` method would have to perform I/O, meaning that it could not be `local` and would not be treated as isolated. Lime programmers can use either isolated or non-isolated sources and sinks based on a convenience versus performance tradeoff.

6.5 Running and Terminating Tasks

We have shown examples of tasks, created with the `task` operator. The operator returns a handle for the task. This handle is always a subclass of the type `Task`, such as `Filter`, `Source`, etc. The `Task` interface itself is not generic and has only methods related to running and terminating a task.

Typically, one starts a task by calling its `start()` method and awaits completion by calling `rendezvous()` which blocks the caller until the task is in the terminated state. Starting a task has the side effect of starting all the tasks connected to it. The `start` operation can throw an exception if the graph is found to be cyclic at the time it is started. The fact that explicitly cyclic graphs are illegal should not be taken as meaning that feedback within a graph is impossible. Such feedback must employ the messaging techniques described in Section 8.

Connecting a task in an already-started graph to another task causes the new task to be started as part of the running graph. Connecting two graph segments that were previously

started is allowed (as long as no cycle is created). This action may temporarily pause both graphs to calculate a new schedule before the resultant fused graph is restarted.

A task stops running when an uncaught exception of any kind occurs during the execution of the task worker method. By convention, tasks are expected to indicate "normal" termination by throwing `Completion` or one of its subclasses.

Task termination propagates in either an orderly or less orderly fashion depending on the reason for termination. If the task terminates due to throwing `Completion` or a subclass thereof, then termination is "orderly," meaning that there is an attempt to consume all queued values. If the graph has a single source and that task is the first to terminate, the graph will drain as much data as possible before terminating.

If termination is caused by a `Throwable` that is not a `Completion`, the tasks of the graph are each stopped as soon as feasible, generally upon return from their current execution. This may leave data on some connections.

Lastly, if more than one task terminates independently, before the propagation of termination from one reaches the other, or if an exception occurs while a task is draining its port, the result is non-deterministic and generally follows the abrupt termination model.

7. Rate Matching

The tasks described in the previous section had compatible port and stream types so that connecting one task to the another was straightforward. The channel decoding tasks for example all have the same type `Filter<block, block>`. In practice however it is necessary to connect tasks from types that are not strictly compatible but only compatible modulo the degree of aggregation. Lime addresses such situations using a language feature called *rate matching*.

In JPEG, each macroblock output from the three channel decoders has to be spliced with the others, one pixel at a time. This is because the color space conversion accepts a tuple corresponding to the color channels of a pixel and coverts the colors from the YCbCr space to the RGB space. In other words, the stream produced by the task in Section 6.3 (lines 7-10) is of the type `` `(block, block, block) `` whereas the desired type is `rgb` or `` `(pixel, pixel, pixel) ``. A rate matcher ("`#`") in this case automates the disaggregation of the blocks into pixels and produces the desired type with very little programmer effort:

```
=> task [ filterY, filterCb, filterCr ]
=> task [ (block # pixel),
          (block # pixel),
          (block # pixel) ]
=> task join `(pixel, pixel, pixel)
=> task Color.toRGB => ...;
```

In general, the `#` operator is enclosed in parentheses with an optional port type parameter written before it and an optional stream type parameter written after it. The left and right types may be omitted if they can be readily inferred across the connect operator, in which case the parentheses may be elided as well. The input and output types of a matcher are related by sharing a *base value type*, as summarized in Table 1. Because the rules are recursive, rate matching can handle multiple levels of aggregation and disaggregation in a single operation[5].

value type	base value type
value array	base type of array element type
bounded value array	base type of array element type
homogeneous tuple	common base type of elements
heterogeneous tuple	itself
all other value types	itself

Table 1. Base value type rules for matchers.

An example of a disaggregating matcher is the rate matching from block to pixels as in `(block # pixel)` since the aggregate type is on the left and the base type on the right. Each matcher inputs a macroblock from its port, buffers all of its array elements internally, and produces a single pixel at a time on its stream. For each input macroblock, the matcher will run 64 times, and thus it has an input:output rate of 1:64. This is in contrast to user-filters which have an input:output rate of 1:1[6].

JPEG also features an aggregating matcher which is needed between the color space conversion task and the RGB descrambler shown in Figure 4. The latter inputs an entire macroblock of `rgb` pixels whereas the former outputs a single `rgb` pixel at a time. To make the connection work, we interpose a matcher to aggregate values into a single array for the sink as follows.

```
=> task Color.toRGB
=> (rgb # rgb[[64]])
=> task RGBDescrambler(...).descramble;
```

In addition to aggregation and disaggregation, a matcher may be used solely for the purpose of re-shaping arguments, for instance `(pixel[[64]] # pixel[[8][8]])` does not change the rate of the data. This kind of usage eases the use of a native "shape" of the data for the method at hand.

Matchers are generally realized as buffers, and the bias in the language toward types of known sizes aid the compiler and scheduler in appropriately allocating storage for the connections between tasks joined through a matcher. In the case of hardware synthesis, matchers, especially when connected to joiners, may sometimes be synthesized directly into wires that route the base values directly between tasks without buffering values at all. This is especially attractive when the base type is a bit or a relatively small-sized value. Such direct routing generally increases bit-level parallelism that is difficult to achieve in software.

The examples shown so far are fixed rate matchers. However, if the input to the matcher is of variable size and the

[5] All levels are flattened for this purpose.

[6] Note that the rate is with respect to the input tuple and the output tuple. A filter may input a tuple comprised of multiple elements and produce a tuple of multiple elements but the input rate is 1 tuple, as is the output rate.

output is fixed then the matcher will have a variable rate. A fixed size output is normally declared by using a type with inherently fixed size (tuple or bounded array). It is also possible to specify matchers with a stream type of statically unknown size (unbounded array) and supply the actual size of each output at runtime as a (non-constant) expression which is evaluated when the matcher is instantiated. The syntax for this uses the contextual keyword `size`[7]. As an example, the disaggregating matcher `(block # pixel)` is equivalent to `(pixel[[64]] # pixel [[]], size 1) => (pixel[[]] # pixel)`.

There are other modifiers that affect the behavior of a matcher, including for example the ability to implement a sliding window over a stream. Refer to [5] for more details.

8. Messaging

The JPEG decoder implementation presented in Section 6 omitted a detail related to the setting of quantization coefficients needed during channel decoding. The dequantization task uses image-specific coefficients that are not static. The task is stateful (allowing for the coefficients to change) but the information is not available at construction time. It is however known to the bitstream parser once the graph has started. Furthermore, this is a one time operation so attempting to pass this information through the graph, although possible, is difficult and leads to inefficient code.

Lime overcomes this limitation while maintaining isolation properties with a feature called *messaging*. It allows two tasks to communicate using a side "channel" that is outside the normal flow of data along port-to-stream connections. Messaging works using a restricted broadcast model in which the timing of message delivery is controlled, a requirement in certain applications including JPEG decoding.

Messaging involves three steps: declaring a message interface type, implementing the receiver, and implementing the sender. A Lime message type is an interface with the additional `message` modifier. All of the methods of a message interface are considered `local` and must return `void` and only have value arguments. Tasks specify that they can receive a message type by implementing its interface. For example, the `Quantization` class might be defined as shown.

```
public message interface Quantizers {
  public void setQC(int[[Color]] ac,
                    pixel[[]][64]] qt);
}

public class Quantization implements Quantizers {
  pixel[[64]] quantizers; // default values
  ...
  public local void setQC(int[[Color]] ac,
                          pixel[[]][64]] qt) {
    quantizers = new block(qt[ac[component]]);
  }
}
```

[7] Contextual keywords are not reserved words outside this one context.

A method indicates that it may send a particular message type by using the `sends` clause as part of its declaration. In JPEG decoding, the bitstream parser could indicate that it sends the quantization coefficients as shown below.

```
 1 public class BitstreamParser {
 2   ...
 3   public block nextBlock() sends Quantizers {
 4     if (pixelsProcessed < pixelsExpected) {
 5       if (!messageSent) { // only send once
 6         setQC(acCoefficients, qtCoefficients);
 7         messageSent = true;
 8       }
 9       return parsedMacroblock();
10     } else
11       throw new StreamUnderflow(); // done
12   }
13 }
```

The call to `setQC` on line 6 invokes the message interface method. Every task created from a class that implements the message interface will receive that message.

In general, messages can be sent downstream (in the same direction as the dataflow) or upstream, but not "sideways" which will result in a runtime illegal message exception if attempted. Thus for any sender-receiver pair, there must exists a sequence of dataflow edges that connect them in one direction or the other. This is necessary for establishing timing which can be described using the metaphor of a "sticky tag". When a task worker method returns after having called a sender method, its normal output is tagged with the sent message. Thereafter, that tag will remain on that data item and stick to any new results on which that data item participates in. Thus the tag will flow through the task graph in a downstream manner and eventually reach the receiver. Because there are potentially multiple paths connecting the sender and receiver, the receiver may observe this tag multiple times. When a tag is first observed by a receiver of that message type, the receiver method from the message interface will be called *prior* to running the worker method.

The timing for messages that are sent upstream can be described in a similar way. One might think of the path connecting the upstream receiver to the downstream sender as establishing synchronization by tagging all data items that the upstream receiver generates. This stream of tags will, in the same order though possibly with gaps, be observed by the downstream sender. When the sender issues a message, it does so after having observed the most recent tag from the receiver. The message should be delivered immediately before the sender invocation that would generate the next tag. Further details are found in [5].

Finally, note that because messages can be sent upstream, the acyclic restriction on task graphs is not so severe as to prevent all communication upstream. However, because messaging is considered an unusual channel of communication (like exceptions), it is advisable to devise the algorithm such that the frequency of messaging is low.

9. Preliminary Experiments

We present some preliminary experimental results of our compiler and runtime system on two benchmarks, JPEG decoding and DES. The former is described extensively in previous sections. DES is a popular block cipher and our implementation of it in Lime makes extensive use of bounded arrays of bits. Besides the source which generates the test data and the sink which confirms that the result is correct, the structure of DES is relatively simple, consisting of an initial permutation `doIP`, 16 stages of `Feistel` encoding, and a final `doIPm1`. The bit-width varies within the Feistel method so the bounds on the bit arrays vary greatly (64, 56, 48, 32, 28, 6, and 4).

In addition to the JPEG and DES, we have written more than 40,000 lines of Lime code. These include a Black-Scholes implementation, Fast Fourier Transforms, audio and image processing algorithms, network intrusion detection, a MIPS processor simulator, and a few other benchmarks. In addition we have developed a substantial Lime Development Kit that mirrors the Java collection classes and also provides graph building utilities for common stream processing idioms.

Our compiler generates Java bytecode, C which is compiled into DLLs, and Verilog which is compiled into FPGA bitfiles. The Lime language runtime can replace tasks dynamically, choosing among the different implementations that may exists. The bytecode backend is the most mature and is a complete implementation of the language. The C backend, which targets the generation of user-defined tasks, handles all the built-in primitive types, bounded array types, some collective operations, and all control flow constructs except for exceptions. It does not yet handle unbounded arrays, values, or objects. Nevertheless, this is sufficient to compile both DES and JPEG decoding.

The FPGA backend is currently less mature and can handle only bits, bounded arrays, and fixed iterations, making it able to compile only DES. The Lime FPGA tool-chain for which results are reported in this paper makes limited use of task- and gate-level optimizations developed in our previous work [18, 20] which demonstrated that we can compile both bit-level and streaming features to the FPGA with high performance. We are in the process of building a new and more robust hardware back-end, and in combination with the initial results in this paper from our new tool chain, we believe we have significant evidence that the current version of the language makes use of bit-level and streaming features synergistically and can be compiled efficiently to hardware.

To provide a baseline for comparison, we have coded the DES and JPEG decoding benchmarks using almost none of the Lime features, most notably by totally avoiding tasks. This roughly corresponds to a vanilla Java implementation in which the programmer has not performed any low-level optimization such as bit packing. We then measure the total time spent in each stage of the algorithms as comparably as we can. The correlation is close but not perfect because a stream version of an algorithm will typically be finer-grained and will incur additional overhead from method and timer invocation. In the stream versions (Lime, C, and FPGA), we factor out communication costs such as marshalling and crossing JNI boundaries to measure the execution time doing the real work.

	Time (ms)			
		Lime		
task name	**Baseline**	**bytecode**	**C**	**FPGA**
plainTextSource	0.4	7.9	–	–
doIP	0.2	0.4	0.006	0.002
Feistel	83.3	78.9	0.105	0.114
doIPm1	0.2	0.3	0.008	0.002
CipherTextSink	0.5	23.0	–	–

Table 2. Task performance data for DES.

In Tables 2 and 3, each row corresponds to a user-defined task of the benchmarks. The columns under the time-heading show the total running time in milliseconds spent in a particular stage for the particular backend. The experiments were performed on a single processor 2.4 GHz Intel Core 2 Duo machine with 4GB of 1.07 GHz DDR3 memory. The FPGA results are based on timing analysis of our compiler's generated designs after place-and-route for a Virtex-5 XC5VLX50T device using the Xilinx ISE 10.1.3 toolchain.

Because sources and sinks may be global, neither the C nor the FPGA backend will generate artifacts for them. The sources and sinks are also stateful in both benchmarks. All other tasks are local and stateless unless otherwise indicated. The source and sink rows are grayed out to indicate that, in a real environment, they are most subject to change and are not part of the computation kernel.

For DES, the Baseline and Lime versions are fairly comparable in the middle stages which is unsurprising since both are bytecode versions of nearly the same method. The discrepancy in the source and sink arise from the overhead of Lime's bytecode implementation of arrays which have an additional level of boxing. The C and FPGA versions fare much better, improving the crucial Feistel stage by several orders of magnitude. The speed increase arises from successfully lowering a high-level Lime array of bits to a native version which is both packed and free from array bounds check. Although the C and FPGA versions look very similar in performance, the total time aggregation belies the pipeline parallelism inherent in an FPGA implementation. The 0.114 ms represents the latency whereas the actual bandwidth that can be achieved is 16 times greater since the entire design fits on an FPGA. On the other hand, achieving the same level of parallelism in C version without extensive transformation is hard given how fine-grained the Feistel computation is.

Table 3 shows that the Lime version for JPEG decoder is much slower than the Baseline version. The slowdown comes from abundant use of Lime arrays which in the byte-

		Time (ms)		
			Lime	
task name	**stateless?**	**Baseline**	**bytecode**	**C**
bitstream parser	no	213	39091 (455)	–
zigzag	yes	1.7	8946 (196)	6.9
dcDecode	no	0.4	299	1.4
deQuantize	no	1.2	423	2.3
iDCT	yes	118	869	14.4
center	yes	1.5	450	1.9
rgb conversion	yes	83.1	31.8	8.3
rgb descramble	no	1.1	45.8	–

Table 3. Task performance data for JPEG decoder.

code backend is quite unoptimized. In particular, runtime type manipulation overhead, boxing and unboxing of primitive types within the array implementation, and manipulation of ordinals when they are the bounding type makes the array access much slower. As confirmation, we manually modified the `bitstreamParser` and `zigzag` tasks to internally use Java arrays, thus eliminating most of the aforementioned overheads. These changes sped up the former from 39091 ms to 455 ms and the latter from 8946 ms to 196 ms. Because these transformations are entirely local, we can improve the bytecode backend and expect much of this overhead to disappear. However, there will always be some overhead as long as the system is run on an unmodified JVM. Fortunately, our intention is that the software backend will serve as a backup and that we synthesize as much as possible to one of the native backends. The most expensive part of the JPEG decoding is the iDCT stage and the C version here improves over the baseline version by a factor of about 7.

When considering the mid-stages (non-gray rows of tables) of the computation, which typically dominate in real applications, we see that the C and HDL versions are much faster than the baseline versions. We take these results as only preliminary but very encouraging. Though gratifying, it is perhaps expected that a C or HDL version of a user task would run much faster than both the Java and Lime version. What should not be taken for granted is that it is possible to bridge the semantic gap by lowering a Lime program down to C and FPGA. Rather than draw too many performance conclusions from these numbers, which should await a more mature implementations from which we will derive yet more parallelism, these results should serve as convincing evidence that the language design succeeds in remaining high-level while exposing enough parallelism, valueness, and isolation to make this type of compilation possible.

10. Related Work

There is of course a vast amount of related work for a new language and its associated compilation and run-time techniques. Here, we concentrate on the issues of synthesizability, integration of data-flow into imperative languages, and enabling these goals via the containment of side-effects.

10.1 Synthesizable High-Level Languages

There are many methodologies, both academic and commercial, to compile C programs to hardware (often called "C-to-gates" compilers, see [11] for a recent survey). C-to-gates approaches restrict the C dialect into synthesizable subsets, often prohibiting the use of pointers and pointer arithmetic, arbitrary loops, memory allocation and recursion. Although the number of C prohibitions has consistently decreased over time, there remains a substantial programming burden on developers to expose the kinds of parallelism that Lime attempts to unify into a single semantic domain.

Unlike its C-to-gates counterparts, Bluespec [1, 24] is a hardware description language with Verilog-like syntax. Bluespec uses a rule-based model of computation, founded on Term Rewriting Systems, to describe concurrency in hardware in a way that is amenable to formal analysis and synthesis. C-to-gates and Bluespec offer some productivity advantages compared to programming in hardware description languages such as Verilog and VHDL. However they are not as rich as Lime and do not offer the benefits of a dynamic and managed object-orientation language that we believe is important for effectively exploiting heterogeneous architectures.

A more object-oriented approach, also sharing Lime's philosophy of covering different forms of parallelism, is used by Greaves and Singh as described in [16] and [17]. Their work introduced Kiwi as a parallel programming library with an associated synthesis system for C# and .NET concurrency mechanisms. They do not share our goal of compiling the same source code into efficient software *and* hardware. Rather, their work aims to define hardware semantics for existing parallel programming constructs including monitors, events, message passing, and asynchronous threads. We believe that the task-based execution model in Lime is simpler, offering a greater degree of determinism and empowering simultaneous compilation to fundamentally disparate architectures.

10.2 Data-Flow Languages

The Lime task programming model resembles Synchronous Dataflow (SDF) [23] where actors in a dataflow graph have statically-known input and output rates. A weakness of SDF is the inability to address variable rate actors which arise quite often in algorithms. This limitation is rooted in the formalisms of Kahn process networks [22] which gave rise to the innovation of communicating sequential processes [19] to introduce compositional non-determinism for streaming models. In Lime, filters are actors with a 1:1 input to output rate but Lime addresses the practical need for variable rates and non-determinism with the `match` operator. The matching mechanism allows the language to retain stronger guarantees of dead-lock freedom than is otherwise possible using communicating sequential processes.

There is a rich history of languages based on dataflow and streaming (see [30] for a comprehensive survey and [31] for a recent review). Dataflow principles are central to the Lime language design, provided via the task and connect operators (to expose dataflow), value types (to disallow arbitrary mutation), and localness (to allow side-effects but in isolated and safe contexts). Seminal dataflow languages, such as Lucid [4], Id [3], and VAL [2] restrict variables to single assignment, and disallow side-effecting statements as the cost for using a dataflow model. In Lime, parts of the program are able to remain imperative while achieving the same result.

Much of the work on dataflow and streaming has focused on embedded systems and digital signal processing. This usually involves completely new languages with new operational semantics. Lime is able to incorporate similar ideas into a widely deployed language using familiar syntax and semantics. The Lime task operator is similar to the *lift* operator in Functional Reactive Programming [13] (FRP) which converts a function over a static value to a function over *time* reified as a *Behavior*: a behavior is a continuous function that can be sampled at any time to retrieve a corresponding value. FRP also provides combinators similar to the Lime connect operator to compose and create new behaviors and *Events*. An implementation of the FRP model in the Java programming language can be found in [12] which also compares FRP in its pure Haskell functional form to the Java form. In that comparison it was found that the latter is considerably more verbose. In contrast, Lime introduces local type inference and a compact grammar to cleanly integrate the streaming model into Java.

The availability of messaging constructs in the language address what might otherwise appear as strict limitations on the exchange of necessary data in a distributed model. Lime's messaging feature is similar to the teleport messaging techniques introduced by Thies et al. [34] for the StreamIt [33] programming language. StreamIt is a language based on SDF. It is geared toward compile time construction and refinement of stream graphs for efficient execution on multicore architectures. StreamIt is an elegant language but the lack of object-orientation, variable-size arrays, generics, run-time stream creation, and many other features available in Lime make it inadequate for "programming in the large".

Other recent work on stream programming for multicore architectures has shown that it is possible but not trivial to introduce pipeline parallelism, inherent to stream programs, into legacy programs. In [32] and [10], profiling and feedback tools attempt to aid the programmer in inserting pipelining stages into existing C programs. This work is akin to pipelining C programs for C-to-gates compilation. Lime encourages and facilitates the expression of pipeline parallelism more naturally and provides a gentle migration path from existing Java programs to Lime.

The idea of computing explicitly with dataflow graphs in a Java language context resembles both the Exotask [6] and StreamFlex [29] systems. Both systems combine checkable restrictions on field access with other restrictions on object reachability to achieve a dataflow style of programming with isolated nodes. Neither of the cited systems expands the Java language and so neither is as expressive as Lime. Lime introduces values as a first-class type family with both immutability and referential transparency. Exotasks and StreamFlex, in contrast, check for recursive immutability in a post-hoc fashion, which can be expensive and implies a closed world. Neither exploits referential transparency. Lime distinguishes stateless nodes from stateful ones, permitting a class of optimizations not available to the others. Exotasks control object reachability via extra-linguistic means (controlling initial conditions by instantiating graphs from templates) and StreamFlex uses an ownership type system to tightly control object lifetimes, while Lime uses valueness, localness, isolating constructors, and the inherent single-reference semantics of the `task` operator to achieve the same ends.

10.3 Containment of Side-Effects

Lime uses deeply immutable `value` types, `local` methods, and controlled initialization to compositionally obtain referential transparency and restriction of side effects to a single isolated task. Many other approaches to bounding side-effects have been used or proposed, motivated by various different goals. We describe and contrast a representative selection of these approaches.

Imperative and functional languages must each solve problems that are in some sense duals: for the former, bounding the scope of side-effects without making the language overly tedious to use; for the latter, adding the capability of side-effects without compromising the advantages of the functional approach.

For imperative languages, there was considerable optimism that compiler analysis could discover regions of code with limited side-effects that would allow a high level of optimization, even in the presence of pointers (e.g., [27]). However, these analyses proved to be fragile, and programmers could not easily comprehend the coding requirements to reliably enable a high degree of optimization.

Our work on containment of side-effects traces back to Reynolds' work on syntactic control of interference [26]. Boyland [9] includes a good summary of recent work, particularly in the context of object-oriented languages. One strand of recent work has focused on enforcing such properties as read-only [7]. However, this is insufficiently strong for the kind of isolation required by our task model: it only ensures that a method will not modify its parameter, but does not ensure that some other method does not modify it concurrently.

Ownership types (e.g., [8]) have also been the subject of considerable research. These type systems are designed to allow various forms of sharing across multiple threads, but pay a price in the complexity of the annotations and/or restrictions in expressiveness. Rather than attempting to allow

sharing of mutable state across threads or tasks, Lime makes use of sole-reference isolation. This significantly reduces the annotation burden on the programmer.

Monads [25, 35] are used in lazy functional languages to encapsulate the side effects of, most notably, input and output into IO monads which have additional arguments and return values representing the state of the world. Lime's evaluation model is eager and so there is no need to avoid side effects for deep semantic reasons. Instead, it is enough to limit the scope of side effects through the isolation mechanism.

Acknowledgments

Shan Shan Huang made substantial contributions to an earlier version of the Lime language. We thank Martin Hirzel, Doug Lea, David Ungar and the reviewers for their valuable comments.

References

[1] Bluespec SystemVerilog Reference Guide, version 3.8. www.bluespec.com, 2004.

[2] ACKERMAN, W., AND DENNIS, J. VAL: a Value-Oriented Algorithmic Language. Tech. Rep. MIT-LCS-TR-218, Massachusetts Institute of Technology, 1979.

[3] ARVIND, GOSTELOW, K., AND PLOUFFE, W. An asynchronous programming language and computing machine. Tech. Rep. 114, University of California at Irvine, 1978.

[4] ASHCROFT, E., AND WADGE, W. Lucid, a nonprocedural language with iteration. *Communications of the ACM 20*, 7 (1977), 526.

[5] AUERBACH, J., BACON, D. F., CHENG, P., AND RABBAH, R. Lime language manual (version 2.0). Tech. Rep. RC-25004, IBM Research, Oct 2010.

[6] AUERBACH, J., BACON, D. F., IERCAN, D., KIRSCH, C. M., RAJAN, V. T., RÖCK, H., AND TRUMMER, R. Low-latency time-portable real-time programming with Exotasks. *ACM Trans. Embed. Comput. Syst. 8*, 2 (2009), 1–48.

[7] BIRKA, A., AND ERNST, M. D. A practical type system and language for reference immutability. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada, 2004), pp. 35–49.

[8] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, 2002), pp. 211–230.

[9] BOYLAND, J. Why we should not add readonly to Java (yet). *Journal of Object Technology 5*, 5 (2006), 5–29.

[10] BRIDGES, M., VACHHARAJANI, N., ZHANG, Y., JABLIN, T., AND AUGUST, D. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), pp. 69–84.

[11] CARDOSO, J., AND DINIZ, P. *Compilation Techniques for Reconfigurable Architectures*. Springer, 2008.

[12] COURTNEY, A. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages* (2001), Springer-Verlag, pp. 29–44.

[13] ELLIOTT, C., AND HUDAK, P. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (1997), pp. 263–273.

[14] GORDON, M. I., THIES, W., AND AMARASINGHE, S. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA, 2006), pp. 151–162.

[15] GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, 2002), pp. 291–303.

[16] GREAVES, D., AND SINGH, S. Kiwi: Synthesis of FPGA circuits from parallel programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (2008).

[17] GREAVES, D., AND SINGH, S. Exploiting system-level concurrency abstractions for hardware descriptions. Tech. Rep. MSR-TR-2009-48, Microsoft Research, Apr 2009.

[18] HAGIESCU, A., WONG, W.-F., BACON, D. F., AND RABBAH, R. A computing origami: folding streams in FPGAs. In *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California, 2009), pp. 282–287.

[19] HOARE, C. Communicating Sequential Processes. *Commun. ACM 21*, 8 (1978), 677.

[20] HORMATI, A., KUDLUR, M., MAHLKE, S., BACON, D. F., AND RABBAH, R. Optimus: efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Atlanta, GA, USA, 2008), pp. 41–50.

[21] HUANG, S. S., HORMATI, A., BACON, D. F., AND RABBAH, R. M. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the European Conference on Object-Oriented Programming* (2008), J. Vitek, Ed., vol. 5142 of *Lecture Notes in Computer Science*, Springer, pp. 76–103.

[22] KAHN, G. The semantics of simple language for parallel programming. In *Proceedings of IFIP Congress 74* (Stockholm, Sweden, Aug. 1974), J. Rosenfield, Ed., pp. 471–475.

[23] LEE, E. A., AND MESSERSCHMITT, D. G. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers 36*, 1 (January 1987), 24–35.

[24] NIKHIL, R. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design* (2004), pp. 69–70.

[25] PEYTON JONES, S. L., AND WADLER, P. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, 1993), pp. 71–84.

[26] REYNOLDS, J. C. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona, 1978), pp. 39–46.

[27] RYDER, B. G., LANDI, W. A., STOCKS, P. A., ZHANG, S., AND ALTUCHER, R. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst. 23*, 2 (2001), 105–186.

[28] SASITORN, J., AND CARTWRIGHT, R. Efficient first-class generics on stock Java virtual machines. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France, 2006), pp. 1621–1628.

[29] SPRING, J. H., PRIVAT, J., GUERRAOUI, R., AND VITEK, J. StreamFlex: high-throughput stream programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Montreal, Quebec, 2007), pp. 211–228.

[30] STEPHENS, R. A survey of stream processing. *Acta Informatica 34*, 7 (1997), 491–541.

[31] THIES, W. *Language and compiler support for stream programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[32] THIES, W., CHANDRASEKHAR, V., AND AMARASINGHE, S. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), pp. 356–369.

[33] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. P. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), Springer-Verlag, pp. 179–196.

[34] THIES, W., KARCZMAREK, M., SERMULINS, J., RABBAH, R., AND AMARASINGHE, S. Teleport messaging for distributed stream programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, Illinois, 2005), pp. 224–235.

[35] WADLER, P. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France, 1990), pp. 61–78.