RC25004 (W1005-136) May 28, 2010
Computer Science

IBM Research Report

LIME
The Liquid Metal Programming Language

Language Reference Manual

Joshua Auerbach, David F. Bacon, Perry Cheng, Rodric Rabbah
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Contents

List of Figures

Chapter 1

Introduction

Lime is a general-purpose programming language that extends the Java programming language in several ways. The Lime extensions are designed to make it feasible to realize implementations of large portions of a program in hardware via direct synthesis into a programmable or reconfigurable logic fabric such as a Field Programmable Gate Array (FPGA). Residual portions of the program doing hard-to-synthesize (and usually infrequent) operations such as dynamic class loading continue to run as bytecodes in a hosting JVM. For rapid iteration during development, the entire language will always run as bytecodes.

The increasing availability of systems with FPGAs offers an opportunity to customize processing architectures according to the applications they run. An application-customized architecture can offer extremely high performance with very low power compared to more general purpose designs. However, FPGAs are notoriously difficult to program, and are generally programmed using hardware description languages like VHDL and Verilog. Such languages lack many of the software engineering and abstraction facilities that are taken for granted in modern Object-Oriented (OO) languages. In hybrid architectures which couple conventional CPUs to programmable (e.g., FPGA) or fixed-function accelerators (e.g., cryptography engine), additional complexity is introduced by the fact that the CPU and accelerators are programmed in completely different languages with very different semantics.

Lime allows hybrid systems (e.g., CPU+FPGA architectures) to be programmed in a single high-level OO language that maps well to both architectures. While at first glance it may seem that conflicting requirements for programming these different kinds of systems create an inevitable tension that will result in a hodgepodge language design, it is our belief that when the features are provided at a sufficiently high level of abstraction, many of them turn out to be highly beneficial in both environments.

By using a single language we open up the opportunity to hide the complexity of domain crossing between CPUs and accelerators. Furthermore, we can fluidly move computations back and forth between the types of computational devices, choosing to execute them where they are most efficient or where we have the most available resources.

Our long-term goal is to "JIT the hardware"—to dynamically select methods for compilation and synthesis to hardware, potentially taking advantage of dynamic information in the same way that multi-level JIT compilers do today for software.

An early version of Lime, focusing on the value types, was first described in [3]. The current language (as of June, 2010) is described in [1], including new features for streaming, bounded types, local/global modifiers, and so on. That paper provides motivations, examples, related work, and a good introduction to the language. This manual is designed more as a reference.

1.1 Java Compatibility

Lime is based on Java and there is an intended level of compatibility with Java. From a source standpoint, all Java programs either already are legal Lime programs or can be converted to Lime programs via a purely syntactic transformation. Lime keywords, if used already as identifiers in the Java program, can be marked with Lime's lexical escape convention. Lime's differing semantics for generic types and arrays can be initially suppressed by marking such constructs with Lime's compatibility escape convention. Once this has been done, the program can be gradually transformed to take more and more advantage of Lime's enhancements.

From a binary standpoint, code compiled with a Java compiler can interoperate with code compiled by a Lime compiler in many ways; Java compatibility is described in detail in Section 20. In addition, as particular Lime features are introduced, sections titled "Java Compatibility" will discuss particular issues raised by those features.

1.2 Package lime.lang

Just as Java has a package java.lang which is implicitly imported on-demand into every compilation unit, Lime has an equivalent package lime.lang. In a Lime program, *both* lime.lang and java.lang are implicitly imported on-demand into every compilation unit. Importing either of these packages explicitly is not an error but is not necessary.

1.3 The Lime Development Kit

Lime provides a set of classes (collectively called the LDK or "Lime Development Kit") similar in spirit to the Java Development Kit. In particular, Lime has its own set of hardware friendly collections and also supplies hardware-friendly scalar types. Types in lime.lang and lime.util are discussed as appropriate in this manual but it is not a goal of the manual to provide a complete exposition of the LDK. Either a discussion of the LDK or javadoc-style documentation may be provided in a future publication.

1.4 More About the Manual

Although the exposition in this manual is informal and example driven, there is an attempt to state all the known semantic issues clearly and to call out any new grammar (including new reserved words or lexical symbols). Grammatic productions are written as extensions to the grammar presented in the chapters of the Java Language Specification, third edition (not the summary grammar of the appendix, which is different). Lexemes to be taken literally are in single quotes, e.g. `'task'`. Other lexical tokens are capitalized (e.g. `IDENTIFIER`). Non-terminals are in normal text. Otherwise, grammar descriptions avoid meta-syntactic conventions other than the standard BNF meta-operators "`::=`" and "`|`", and the subscript "$_{opt}$" meaning optional.

Any quoted word that appears in a grammar section in this manual should be assumed to be a lexically reserved keyword. Lime adds several such keywords that are not in Java, such as **task**, **local**, **global**, **universal**, and **split**.

Chapter 2

Generics

Generic types in Lime are a forward-compatible superset of Java's generic types, with increased expressive power. Since use of the new features inhibits interoperability with Java, Lime allows selected generic types to be labelled as Java compatible. Thus, there are two kinds of generic types in Lime.

- *Java* generics are compiled by erasure as in Java and are fully interoperable with Java.
- *Lime* generic types support primitive types as type arguments and permit parameterized types and type variables to participate in runtime operations such as array allocation, instance allocation, casts, and instanceof. They also allow type variables to be used in the definition of static members.

A generic type is classified by the following rules.

1. If a generic type was compiled by a Java compiler and is present only in class file form, it is Java generic.
2. If a generic type was compiled by the Lime compiler and is not otherwise marked as being a Java generic, it is a Lime generic.
3. The "~" character between the class or interface name and its type parameters (e.g. **class** Foo~<T>{...} marks the type as a Java generic.

The marking of a generic type occurs only on the type *definition*. Both kinds are *used* in the same way: one says Foo<String> x, not Foo~<String> x.

The special mark indicating a Java generic may only appear on a top-level (non-nested) type. It extends by implication to all nested types (including nested **static** types).

The same distinction between Java and Lime generics applies to generic methods. In this case, the "~" character comes before the method-level type parameters. A Lime generic instance method may not override or implement a Java generic instance method. For example, if a Lime class implements java.util.Collection<T> the method <T> T[] toArray(T[] a) method must be implemented as ~<T> T[] toArray(T[] a).

The following sections describe the semantic differences between Lime generics and Java generics and cover some special topics related to generic types in Lime. The semantics of Lime generics are similar in flavor to NextGen [4] and LM [5] but are implemented differently, support primitive types as type arguments, and include a strategy for backward compatibility with Java (albeit at the expense of having two kinds of generics).

2.1 The Set of Reifiable and Instantiable Types

The Java specification (3rd edition, section 4.7) defines what it means for a type to be *reifiable*. Only the reifiable types are permitted as the item type of an array allocation, or as the right-hand-side of **instanceof**. If a non-reifiable type is used in a cast, the cast is allowed but a warning is issued and the cast has a different effect at runtime than the apparent intent. Somewhat confusingly, a type doesn't have to be reifiable to be *instantiable* (a different concept, considered below). The reifiable property captures the fact that the type is distinct from other types for purposes of runtime checking.

In Lime, we retain the rule that a type must be reifiable to be used in array allocations and instanceof, or to be runtime-effective when used in a cast. However, we redefine "reifiable" to take into account the difference between Java generics and Lime generics.

To review, in Java, a type is reifiable if it conforms to one of the following.

- It refers to a non-generic type declaration.
- It is a parameterized type in which all type arguments are unbounded wildcards.
- It is a raw type.
- It is a primitive type.
- It is an array type whose component type is reifiable.

Note that a type variable is never reifiable in Java, and most parameterized types are not reifiable either.

In Lime, the rules are modified as follows (changes are emphasized).

- It refers to a non-generic type declaration (as in Java).
- *With caveats discussed below* it may be a parameterized type in which all type arguments are unbounded wildcards (as in Java).
- It is a raw type derived from an *Java* generic type *(raw types cannot be derived from Lime generic types).*
- It is a primitive type (as in Java).
- *It is a type parameter of a Lime generic type.*

- *It is a parameterized type derived from a Lime generic type with all type arguments being reifiable types.*

- It is *Lime* array type whose component type is reifiable by these expanded rules or a Java-compatible array type whose component type is reifiable by the older Java rules (Lime has both an expanded and a Java-compatible syntax and semantics for arrays, as is explained in section 12).

Note that the subset of Lime semantics involving only Java generics has the same semantics as Java. The Lime generics are more expressive than the Java generics except that you (usually) can't express the raw type equivalent to a Lime generic by giving the type without type arguments. There are a few minor exceptions to this rule, for purely syntactic reasons noted in the relevant sections, but the resulting raw type can never become the type of an expression, and therefore, the "unchecked conversion" described in the Java specification section 5.1.9 cannot occur.

Lime is similar to Java in not providing full reification of wildcard types. This leads to a subtlety in the case of parameterized types all of whose parameters are unbounded wildcards. Such a type is considered reifiable in Java, but this works only because raw types are reifiable in Java and the all-unbounded-wildcard case has the same runtime representation as a raw type. In Lime, no type using wildcards can be fully reifiable since Lime does not represent all the information present in a capture conversion at runtime. However, Lime follows Java in allowing the unbounded wildcard in **instanceof** and in casts (although not as the item type of an array). This is safe since the test is merely asking whether the type is *some* parameterization of the generic type, which is readily answerable with the information at Lime's disposal.

To summarize what we've said so far, with examples, consider the following.

```
import java.util.List; /* from Java */
class Foo<T> { /* A Lime generic */
  ...
  /* All of the following are legal inside the declaration of Foo */
  new T[10];
  if (x instanceof T) ...
  new Foo<T>[10];
  if (x instanceof Foo<T>) ...

  /* The following is legal even outside the declaration of Foo */
  new Foo<String>[10];
  if (x instanceof Foo<String>) ...

  /* The following is runtime effective and will issue no warning */
  a = (T) x;
  b = (Foo<String>) x;
  c = (Foo<T>) x;
```

```
    /* Illegal since List is a Java Generic */
    new List<String>[10]; // error
    new List<T>[10];

    /* The following will cause a warning and will cast to the raw type */
    d = (List<String>) x
    e = (List<T>) x;

    /* The following is illegal because only Java generics have raw types */
    new Foo[10];
}
class Bar~<S> { /* a Java generic */
    /* The following are illegal because S is not reifiable */
    new S[10]; // error
    if (x instanceof S) ... // error

    /* The following are still illegal even though Foo is a Lime generic because S is not
       reifiable */
    new Foo<S>[10]; // error
    if (x instanceof Foo<S>) ... // error

    /* But the following is allowed */
    new Bar[10]; // raw type

    /* The following will cause a compiler warning and will not check anything at runtime
       */
    a = (S) x;
}
```

Now consider Java's rule for what types are instantiable. This is described in section 15.9 of the specification. Note that not all reifiable types are instantiable and not all instantiable types are reifiable. Of course, to be instantiable, a class has to be non-abstract, visible, etc. But, in addition, any wildcard type arguments at all will render a class non-instantiable and naked type variables (e.g. **new** T()) are never instantiable. On the other hand, a parameterized type (which is not reifiable) is nevertheless instantiable (but subject to erasure).

Lime semantics are superficially similar in that wildcards also render a type non-instantiable and naked type variables are often (though no longer always) non-instantiable. But, the resulting objects in Lime carry unerased type information in many more cases, because many more types are reifiable. Consider the following expression.

```
x = new Foo<T>();
```

Such an expression is instantiable in both languages. However, if Foo is an Java generic type *or* T is a type parameter of an enclosing Java generic, then Foo<T> is not reifiable and hence the resulting object will have only erased type information. It might, for example, give an overly conservative answer to x **instanceof** Foo<String> because whether or not T was bound to String has been lost. However, when a reifiable type is instantiated, the object carries full unerased type information, so, the larger set of reifiable types in Lime means a larger set of objects carrying such information.

In addition, Lime permits the expression **new** T() (without arguments) for the special case where T is constrained to be a value type (as defined in section 6), because values are guaranteed to have default constructors and Lime considers the dispatching to such constructors to be virtual. Otherwise, Lime must disallow instantiation of naked type variables for reasons explored more fully in section 2.6.

2.1.1 Default Initial Value

The expression T.**default**, where T is any reifiable type variable, refers to the default value of the type denoted by T in each instantiation (ie., the value to which fields of the type would be initialized in the absence of an explicit initializer). For ordinary reference types, this is **null**. For primitive types (see section 2.2) this is the appropriate form of zero or **false**. For value types (see section 6), T.**default** == **new** T(). This supports type parameters that are general enough to encompass both values and mutable classes.

2.1.2 Instanceof with Type Parameter

The **instanceof** operator normally is used to test whether an object is an instance of a type. Lime generics allow type parameters to be used as the type being tested.

However, Lime also allows a type parameter to be used on the left-hand side of an **instanceof** operator, as in

```
if (! (E instanceof RandomlyGenerable<E>)) {
    throw new ContainedValueNotRandomlyGenerableException(this);
} else {
    RandomlyGenerable<E> seeder = (RandomlyGenerable<E>) new E();
}
```

2.2 Primitive Types

The eight Java primitive types are valid substitutes for the type parameters of Lime generics if they meet the restrictions implied by the **extends** clause of the type parameter. The discussion of how a type parameter's **extends** clause can possibly apply to primitive types is deferred to Section 6.10. Lime does not guarantee that primitive types are never boxed when used in this way, but the goal is to achieve efficiency consistent with a non-boxed (hence

probably specialized) implementation of those cases. In any case, any boxing that is done is completely invisible; the primitive type is never exposed in its boxed form.

2.3 Restrictions on the Use of Java Generics in Lime

Because Java generic types are compiled by erasure, the stronger type safety guarantees of Lime and the weaker ones of Java are sometimes in conflict. In many cases, the implementation can compensate by checking for Lime reification and comparing to the expected type at critical program points, but one property in particular (discussed in section 6.2) is that Lime's **value** types may not be **null**. Checking for this property in a pervasive fashion is inconsistent with reasonable efficiency. Consequent, it is illegal to instantiate a Java generic type with a Lime value type or with a Lime reified type variable that could later be instantiated with a value type. More details are provided in section 6.2.

2.4 Reflection, Classes, and Class Literals

Even though a larger set of types is reifiable in Lime, Lime does not guarantee that every distinct type is represented by a distinct Class object. Rather, some degree of erasure (not specified in this language definition) may occur in the *implementation* of types that are reifiable only in Lime, such that types related by erasure may share a Class object. For example, if Foo is a Lime generic class, then **new** Foo<String>().getClass() and new **new** Foo<**int**>.getClass() may return the same object. The type-dependent runtime operations are supported in some other way (e.g. by having instances carry type information).

The motivation for this restriction is that giving each distinct reifiable type its own class object will imply either that every instantiation is uniquely compiled to a class file (too space inefficient) or that Lime runs on a modified JVM (too confining in the short run).

The lack of a guarantee at the Class object level means that reflective variants of type-dependent operations might not give the same answer as the language-supported variant. That is, Lime does not guarantee that **new** T().getClass().isAssignableFrom(x.getClass()) is exactly equivalent to x **instanceof** T. The latter is guaranteed to give a precise answer, but the reflective variation can only be as precise as the degree to which class objects are actually distinct.

To discourage the use of reflection on reifiable types not supported by unique class objects, Lime makes no change to Java's restrictions on the class literal expression. T.**class** and Foo<String>.**class** are illegal in Lime just as they are in Java, regardless of whether the left-hand sides are reifiable. Furthermore, to allow class literals to be used in the more limited ways implied by the other restrictions, Lime does not forbid the use of a raw type with a class literal (that is, even if Foo is a Lime generic type, the expression Foo.**class** is permitted and is essentially equivalent to Foo<?>.**class**).

2.5 Type variables in Static Members

In Java, and hence in a Java generic type, the scope of a type parameter introduced in the class or interface header includes the type's non-static members only. In a Lime generic type, the scope of each type parameter includes the static fields and methods. That is, type parameters can be the types of static fields and the argument and return types of static methods. They can also be used in the initializers of static fields or the bodies of static methods.

If a static method is itself generic (has its own type parameters), then the method may use *both* the method type parameters and the class type parameters.

There is a specific exception to this rule for the special case of a static main method that is intended to be the startup method for an application. It is not a common practice to place such methods in generic classes, but, if one were to do so in a Lime generic class, the main method would be specifically excluded from using the class's type parameters. This restriction is necessitated by the technical details of running in an unmodified JVM.

The semantics for static members using type variables as if each distinct instantiation of a Lime generic type has its own generation of static members, properly parameterized. This does not necessarily imply that there is a distinct physical class file or class object for each such instantiation.

Note that static members are statically resolved and so references to them must be qualified by the full type, including type arguments.

```
class Foo<T> {
  ...
  static T[] vals = new T[10];
  static T getVal(int i) {
    return vals[i];
  }
}
...
int x = Foo<int>.vals[2];
String y = Foo<String>.vals[2];
```

In the previous example, there are two distinct copies of the field vals, once associated with Foo<**int**> and the other with Foo<String>. They have different types and also different contents.

2.5.1 Type Parameters in Static Nested Classes

Static nested classes are not considered to be in the scope of the type parameters of the enclosing class. Thus, if it is desired to parameterize a nested static class, it must define its own type parameters, and the uses of that nested class elsewhere in the enclosing class can instantiate the nested class either with concrete types or which its own type parameters.

2.6 Some Non-Obvious Limitations of Type Variables in Lime Generics

Since the type parameters of Lime generics are reifiable in Lime, we need to make clear that these are still variables (the substituting type is unknown at the time the generic class is compiled). In conjunction with some other principles, this fact may lead to some limitations that will not be immediately obvious to initial Lime programmers.

Principle 1: Generic types should be type-checked on their own without knowing all uses.

Principle 2: Any instantiation of a generic type that was found correct under principle 1, should be capable of being checked by comparing the type arguments to the **extends** clauses (if any) of the type parameters. Instantiations that pass those checks should be legal.

Principle 3: Although Lime establishes new semantics for generic types, it respects Java semantics for the resolution of type members, specifically, that only instance methods are dispatched virtually, while static methods, all fields, and all constructors are dispatched non-virtually.

Thus, even if T is a reifiable type variable, the static field and method references T.x and T.y() have the same semantics as in Java, both in Java generics and in Lime generics. The x and y referred to are those of a type appearing in the **extends** clause of T and not the (unknown) fields or methods of the many possible substituting types.

Also, given object reference z of type T, the expression z.q where q is now an instance field, not a static field, is similarly a reference to the field q defined in a type appearing in the **extends** clause of T and not some unknown field q that might be introduced by a possible substituting type.

Lime specifically holds that the default constructors of value types are virtually dispatched, which is why **new** T() is legal when T is known to be some value type. All other constructors are treated as in Java, which is why the expression **new** T(...) cannot be accepted in general.

As will be further discussed in section 7, the special selectors .size, .first, .last, and .range of the bounded types are also virtually dispatched, hence T.last (e.g.) has its expected meaning when T is known to be a bounded type.

2.7 Source Availability

2.8 Ordinal Parameters

Syntactically, in Lime, a constant expression evaluating to a positive integer can appear as a type argument in a parameterized type. This does not imply that Lime supports integer parameters in a semantic sense; rather, such expressions are converted to ordinal types as will be describe in section 9.1.

2.9 Java Compatibility

Instances of generic classes can only be passed to Java code if they match a signature that Java understands. More specifically:

1. A Java generic class developed in Lime can be published for use by Java code (which must itself be invoked from Lime) as long as its public API does not include any Lime constructs (such constructs can be used internally in private members and the implementation of methods).

2. A Lime generic class can only be passed to Java code by casting it to a type that Java understands, either Object, or a Java generic supertype. This, in turn, only works on the assumption that the Java code does not try to reflectively probe the instance and use members that were not known to it when it was compiled. For example, Java code that attempts to serialize a Lime generic object, then deserialize it and pass the result back to Lime, may end up violating invariants important to Lime but unknown to Java.

3. Static methods of a Lime generic class (whether or not they actually refer to class-level type parameters in a way that would be illegal in Java) are not callable from Java code. The only exception is a properly formed main method, which is kept in a strictly Java-compatible form so that it can be invoked by the JVM's application launcher.

Note that while it is legal for a Lime generic to extend or implement a Java generic, this must be done consistent with the rule that a Java generic cannot be instantiated with a value type. For example, a Lime generic collection class whose type parameter T can be any type may *not* implement java.util.Collection<T>, because T permits the substitution of a value type. On the other hand, if the collection declares the type variable as T **extends** Mutable< T> (the Mutable type, discussed in section 14.5, precludes substitution by any value type), then it is permitted to implement java.util.Collection<T>.

2.10 New Grammar

See section 4.5.1 of the Java Language Specification for `ActualTypeArgument` and section 4.2 for `PrimitiveType`.

```
ActualTypeArgument     ::= PrimitiveType
```

See section 15.11 of the Java specification for `FieldAccess`, section 15.12 for `MethodInvocation`, section 4.5.1 for `TypeArguments`, section 4.3 for `TypeDeclSpecifier`, and section 15.9 for `ArgumentList`.

```
MethodInvocation    ::= LimeGenericType '.'  TypeArguments_opt IDENTIFIER '(' ArgumentList_opt ')'
FieldAccess         ::= LimeGenericType '.'  IDENTIFIER
FieldAccess         ::= DefaultFieldAccess
LimeGenericType     ::= TypeDeclSpecifier TypeArguments
LimeGenericType     ::= LimeOrdinalType
DefaultFieldAccess  ::= IDENTIFIER '.' 'default'
```

The revised FieldAccess and MethodInvocation syntax permits type arguments before the dot (to convey class-level arguments) as well as after the dot, in the case of methods (to convey method-level arguments) when accessing static fields or invoking static methods.

An additional change to support ordinal parameters is shown in section 9.3, which also defines the `LimeOrdinalType` production used above.

See section 8.1 of the Java specification for `NormalClassDeclaration`, section 8.1.2 for `TypeParameters`, and section 9.1 for `InterfaceDeclaration`. The following productions are replaced (only the initial portion is shown, the trailing portion is elided).

```
NormalClassDeclaration:::= ClassModifiers_opt 'class' IDENTIFIER TypeParameters_opt ...
InterfaceDeclaration:  ::= ClassModifiers_opt 'interface' IDENTIFIER TypeParameters_opt ...
```

The affected productions are replaced as follows.

```
NormalClassDeclaration:::= ClassModifiers_opt 'class' IDENTIFIER LimeTypeParameters_opt ...
InterfaceDeclaration:  ::= ClassModifiers_opt 'interface' IDENTIFIER LimeTypeParameters_opt ...
LimeTypeParameters     ::= JavaGenericMarker_opt TypeParameters
JavaGenericMarker      ::= '~'
```

Chapter 3

Type Definitions

A **typedef** can be used to provide an alias for a type, allowing for more concise code. For instance,

```
typedef Pixel = foo.bar.baz.RGBPixel;
typedef Dictionary = HashMap<String,String>;
```

It is also good practice to use typedefs when a global decision about a fundamental type is made for the program. For instance,

```
typedef real = float;
```

allows the program to be easily adapted to use higher precision arithmetic if that becomes desirable.

Semantically, a type definition is just one step above a lexical macro: type definitions are checked for correctness and are subject to scoping and visibility rules (discussed below) but, otherwise, the referenced type is substituted for the defined symbol prior to any other semantic analysis.

Typedefs have the same scoping and visibility rules as class definitions. When introduced at top level in a compilation unit, they may be labelled as **public** (otherwise they are just package visible). If public, they can be imported and referred to from other packages by their fully qualified name. When nested inside a class but outside any method, they are implicitly static, accept the same visibility modifiers as a static nested class, and are referred to in the same way. When nested inside a method body they are visible only there, as would be the case for a local class definition.

A typedef whose right hand side is a generic type given without type arguments is treated as a raw type.

```
typedef Lst = List; /* Raw type warning */
Lst<String> x; /* Error: lst is not a generic type */
typedef LstString = List<String>; /* Correct */
LstString x; /* Correct */
```

When a symbolic name is given to an array type the behavior of the resulting type has to be understood in the context of Lime's semantics for arrays, which are discussed in section 12.

3.1 Java Compatibility

Instances of classes containing nested (member or local) **typedef** clauses cannot be passed as arguments to methods not compiled with a Lime compiler. On the other hand, the use of a symbolic name introduced via **typedef** when creating an object that is later passed to a Java method is not, by itself, a problem, as long as the actual type of the object, after resolution of all **typedef** names, is not one that violates the first rule. Assuming that JavaBaz in the following was compiled by a plain Java compiler, consider the examples.

```
class Bar {}
typedef Foo = Bar;
class Danger {
  typedef uint = int;
}
JavaBaz baz = new JavaBaz();
Foo x = new Foo();
baz.do(x); // fine
baz.do(new Danger()); // illegal
```

3.2 New Grammar

See section 7.6 of the Java Language Specification for `TypeDeclaration`. See section 8.1.1 for `ClassModifiers`. See section 4.1 for `Type`.

```
TypeDeclaration ::= LimeTypedef
LimeTypedef     ::= ClassModifiers_opt 'typedef' IDENTIFIER = TypeDefType ';'
TypeDefType     ::= Type
```

Chapter 4

Type Inference

Lime performs *local* type inference for initialized field and variable declarations. Instead of writing the type, the keyword **var** is used for mutable fields or variables and **final** for immutable fields or variables. For instance,

```
var x = 3; // same as "int x = 3"
final y = 3; // same as "final int y = 3"
```

Note that type inference is performed purely locally. Thus, if the intent was for x to be a **double** value, then either of the following declarations would be needed:

```
var x = 3.0;
double x = 3;
```

Either the type of the initialization expression must be exactly the intended type, or the intended type must be declared explicitly and the initialization expression will be widened as needed, if possible.

An explicit type may still be needed if the type of the variable or field will subsequently change. For instance, the following two statements are equivalent:

```
HashSet<Foo> foos = new HashSet<Foo>();
var foos = new HashSet<Foo>();
```

however, if a different type is subsequently desired, then the initial declaration must use a more general type:

```
Set<Foo> foos = new HashSet<Foo>();
...
foos = new TreeSet<Foo>();
```

4.1 New Grammar

See section 8.3 of the Java Language Specification for `FieldDeclaration`, `VariableInitializer`, and `VariableDeclaratorId`. See section 14.4 for `LocalVariableDeclaration`, section 8.3.1 for

`FieldModifiers`, and section 8.4.1 for `VariableModifiers`. Note that just as it is semantically illegal for any modifier to appear twice, the **final** keyword cannot appear both as a modifier and as an `InferredType`.

```
FieldDeclaration                ::= LimeInferredFieldDeclaration
LocalVariableDeclaration        ::= LimeInferredVariableDeclaration
LimeInferredFieldDeclaration    ::= FieldModifiers_opt InferredType FullDeclarator
LimeInferredVariableDeclaration ::= VariableModifiers_opt InferredType FullDeclarator
InferredType                    ::= 'var' | 'final'
FullDeclarator                  ::= VariableDeclaratorId '=' VariableInitializer
```

Chapter 5

User-defined Operators

Lime allows programmers to define special meanings of certain standard lexical symbols by defining their own operations which override the automatically defined ones (often making such operations legal on types that would not otherwise permit them). For instance, we can define a unary complement operator for bit:

```
public bit ~ this { return this == one ? zero : one; }
```

A binary operator could be similarly defined. For instance, the operator & for bit can be defined as follows:

```
public bit this & (bit that) { return this == one && that == one; }
```

The unary operators that can be user-defined are

```
+++  ---  !  ~  -  +
```

The operators +++ and −−− are new prefix operators in Lime, discussed in section 6.9.

The infix operators that can be user-defined are

```
>  >=  <  <=  &&  ||  +  -  *  /  &  |  ^  %  <<  >>  >>>  ::
```

The operator "::" is the "range" operator (new with Lime) and is explained in section 8.

Note that the equality operators ("==" and "!=") may *not* be redefined in Lime. Value types (see next section) have compiler-generated equality operators that check for recursive value equality. The .equals() method should still be used when comparing for conceptual rather than literal equality (for instance, to check if two sets with different representations denote the same set of elements).

Lime also permits the array indexing operator [ ] to be redefined both for access (when not the target of an assignment) and for setting (when an indexed entity is the target of an assignment). For example,

```
public bit this [bitindex index] { return bitstore.get(index); }
public bit this [bitindex index] (bit newval) {
  bitstore.put(index, newval);
  return newval;
}
```

The semantics of all user-defined operators are very much those of the method calls that they resemble by virtue of having both formal parameters and a method body. Whenever one of the redefinable operators is encountered, the compiler first identifies the receiver. This is the sole operand of a unary operator, the left-hand operand of an infix operator, or the expression preceding the square brackets for an array operation. If the type of this expression defines a "method" for the operator using the syntax just described, then that method is assumed. The implicit arguments to that method are: none in the case of a unary operator, the right-hand operand in the case of an infix operator, the index in the case of an array access, and both the index and the new value in the case of an array store operation. The types of these must agree with the method declaration using Java's standard algorithm for type inference and type resolution for method call, else a type error is indicated.

Lime does not check the implementations of user-defined operators to ensure that they meet naive semantic expectations (thus, one could define a $-$ operator that adds and a $+$ operator that subtracts). The compiler does not even check the return type, so, for example, one could define a $+$ operator with a **void** return. However, when user-defined operators are employed in interfaces (e.g. as discussed in section 6.9) the contract will generally imply a semantics for the operators that is consistent with such expectations. The package lime.lang.tests contains testing harnesses that perform randomized testing to ensure that an implementation of an interface meets its contract, and includes tests for most of the standard interfaces in lime.lang and lime.util (see Section 21). Any implementation of those interfaces should pass the corresponding test in lime.lang.tests.

In defining the rules for values (see section 6), the index-set operator is assumed to be mutating and all others are assumed to be non-mutating.

Formal arguments of user-defined operators may not use varargs (since operators are explicitly unary or binary, it does not make sense for the declaration to specify a variable number of arguments).

5.1 Compound Operators

The definition of an operator implicitly defines the corresponding compound operator. For instance, by defining the "+" operator:

```
value class foo {
  int val;
  public this + (foo that) { return new foo(this.val+that.val); }
}
```

the "+=" operator is also implicitly defined, so that it can be used as follows:

```
foo f(foo x, foo y) {
  x += y; // same as "x = x+y"
  return x;
}
```

Note that the "+=" operator itself may *not* be defined by the programmer; its meaning is always defined in terms of the underlying operator.

In addition, the pre- and post-increment/decrement operators (eg x++) are implicitly defined by the definition of the "+++" and "−−−" operators, respectively.

Because Lime does not check the return types of user-defined operators, it is possible that a visually reasonable use of the compound version will be found to be illegal. For example, if + is defined for some type T such that its return type is inconsistent with assignment to T, then any use of the += operator with a left-hand side of type T will produce a type error related to the = part of the compound, even though the + part is correct.

5.2 Method-like Syntax

User-defined operators can be invoked using a method-like syntax, in which case the name of the method is simply the operator token. For instance

```
foo f2(foo a, foo b) {
  foo c = a.+(b) // same as "foo c = a+b"
  return c.+++() // same as "return +++c"
```

Note that since compound operators are not themselves methods, there is no syntax for invoking them as method calls.

One important motivation for the alternative syntax is that it allows you to invoke a superclass's implementation, which would not otherwise be possible. However, it is not limited to that use.

```
super.+(b)
super.+++()
```

5.3 Java Compatibility

Instances of classes whose declarations include user-defined operators may be passed to code that was not compiled with a Lime compiler. However, the user-defined operators are invisible to Java code. In fact, a Lime implementation is likely to turn these operators into ordinary methods that might be visible in, for example, a class file inspection utility.

However, even if invoking these internally-generated methods from Java code happens to work, it is not an intended feature of the language, and the internal names are subject to change in the future.

5.4 New Grammar

See section 8.4 of the Java Language Specification for `MethodHeader`. See section 15.12 for `MethodInvocation`.

```
MethodHeader                 ::= LimeOperatorHeader
MethodInvocation             ::= LimeOperatorMethodInvocation
LimeOperatorHeader           ::= MethodModifiers_opt Type LimeOperatorDeclarator
LimeOperatorDeclarator       ::= LimeUnaryOpDeclarator
LimeOperatorDeclarator       ::= LimeBinaryOpDeclarator
LimeOperatorDeclarator       ::= LimeIndexGetDeclarator
LimeOperatorDeclarator       ::= LimeIndexSetDeclarator
LimeUnaryOpDeclarator        ::= LimeUnaryOp 'this'
LimeBinaryOpDeclarator       ::= 'this' LimeBinaryOp '(' FormalParameter ')'
LimeIndexGetDeclarator       ::= 'this' '[' FormalParameter ']'
LimeIndexSetDeclarator       ::= 'this' '[' FormalParameter ']' '(' FormalParameter ')'
LimeUnaryOp                  ::= '+++' | '---' | '!'  | ' ' | '-' | '+'
LimeBinaryOp                 ::= '>' | '>=' | '<' | '<=' | '&&' | '||' | '+'
LimeBinaryOp                 ::= '-' | '*' | '/' | '&' | '|' | '^' | '%' | '<<'
LimeBinaryOp                 ::= '>>' | '>>>' | '::'
LimeOperatorMethodInvocation::= OpReceiver '.'  LimeOperator '(' ArgumentList_opt ')'
OpReceiver                   ::= Name
OpReceiver                   ::= 'super'
OpReceiver                   ::= Primary
LimeOperator                 ::= LimeUnaryOp
LimeOperator                 ::= LimeBinaryOp
```

Chapter 6

Values

Lime is designed with two goals in mind: Programmers should be able to program with high-level OO features and abstractions; These high-level programs should be amenable to bit-level analysis and should expose parallelism. To achieve these goals, Lime extends Java with value types and bounded arrays (arrays indexed by value types that have index-like behavior and limited range). In this section, we introduce value types. Subsequent sections introduce ordinal types and value enums, which are special cases of value types. Still later we consider bounded arrays. In discussing these features we will demonstrate how they can be used by the programmer. We will also highlight their implications for the compiler, particularly with respect to efficient synthesis to an FPGA.

Lime's value types share many properties with those of Kava [2]. However, they have simpler type rules and support generics, allowing them to be used to create convenient general-purpose class libraries.

6.1 Value Classes

A value class is a class whose instance fields are all implicitly **final**, and are themselves of value or primitive type. For instance:

```
value class complex {
  public double real;
  public double imag;
  public complex(double r, double i) { real = r; imag = i; }
  public complex this + (complex that) {
    return new complex(this.real+that.real, this.imag+that.imag);
  }
}
```

Instance methods of **value** classes are by default **local** (See Section 14), meaning that they can not access mutable **static** fields of their own or any other type. However, it is possible to override this default.

Value classes may have static fields. The static fields are not restricted to be values, nor are they implicitly final. They behave in the same manner as static fields of mutable classes.

In addition to being immutable, value types are different from mutable types in the following ways:

- they can not be **null**;
- synchronization operations may not be applied (ie., value types are truly stateless)
- they may not define a finalize() method;
- the hashCode() operation depends solely on the value of its fields;
- the == and != operators compare their types and the values of their fields, rather than object identity; and
- they are not allowed to provide an implementation of the index-set operator, even if that implementation is non-mutating.

Synchronization on a value is prohibited statically, unless the value is stored in a variable of Object type (or **universal** type, as explained in sections 6.2 and 6.5) through upcasting. In that case, synchronization causes an exception. Compilation of == and != operators are given value semantics or non-value semantics based on static types, except when the type of either operand is **universal** or Object, in which case different logic is selected according to the dynamic type.

These extra properties, along with immutability, cause value types to lose their reference-like behavior (even though they nominally fit into the Java object type hierarchy). It is not semantically visible whether the assignment of a value produces a reference or a copy.

6.2 Value versus Non-Value Types

The defining characteristic of value types is that they are immutable. An object of a value type, once created, never changes. Lime depends on the immutability of values in fundamental ways and so its type system must enforce the rule that no expression whose static type is a value type can have a runtime type that is not a value type. To enforce this rule, Lime must prohibit any non-value type from having a value supertype.

The opposite situation (a static non-value type with a runtime value type) is safe as long as the compiler inserts sufficient runtime checks to ensure that value semantics are preserved. An effective implementation of Lime must minimize the situations where such checks are needed and so the type system is organized to discourage value types from having non-value supertypes. On the other hand, prohibiting value types with non-value supertypes would compromise both expressivity and interoperability with Java. To achieve a balance between these goals, types are divided into three categories.

1. The *value* types include classes, enums, and interfaces that are modified by the **value** keyword. Value types are always immutable.

2. The *universal* types include interfaces and classes modified by the **universal** keyword. There are no universal enum classes. Universal types represent objects that are not themselves values but that are allowed to have value (and non-value) subtypes. The type java.lang.Object is a **universal** type.

3. The *ordinary reference* types include classes, enums, and interfaces that are not modified by either the **universal** or **value** keywords, with the exception of java.lang.Object, which is implicitly **universal**.

The interface lime.lang.Value forms the root of the subhierarchy of value types. The keyword **value** on an interface implies **extends** Value and the keyword **value** on a class or enum implies **implements** Value. Specifying these relationships explicitly is not an error but is not necessary.

As in Java, all interfaces (including **value** and **universal** ones) have Object as their implicit superclass. A **universal** type can have only **universal** supertypes. A **value** type can have only **value** and **universal** supertypes. An ordinary reference type can have only ordinary reference and **universal** supertypes.

The previous rules concern inheritance alone and should not be interpreted as meaning that other forms of interoperation between values and non-values are prohibited. Ordinary reference types may have fields of value type, and methods that accept and/or return value types. Value types may not have instance fields of non-value type, but they may have methods that accept and/or return non-value types. This is safe because of the checked property that all value fields are **final** (see section 6.4). Value types may have **static** fields of non-value type under certain restrictions (discussed in section 14).

The type parameters of generic types, whether universal, value, or ordinary, may be any type, as long as the usage of those type parameters within the generic type's definition does not cause other rules to be violated.

The elements of an array are conceptually just like instance fields for the purpose of these rules. Therefore the array v[], where v is a value type, is not itself a value type; rather, it is an ordinary reference type whose members are values. Lime has value (ie. immutable) arrays (introduced in section 12) but they require special syntax. A value array must have value elements.

In order to prevent value members of Lime classes from becoming **null** when such classes are passed to Java, some special rules prohibit the use of value types with *Java* generic types (use of value types with *Lime* generic types is generally fine; see section 2 for details).

1. A Java generic type may not be instantiated with a value type.

2. A Java generic type may only be instantiated with a Lime type variable when the bounds of that type variable preclude the possibility of substitution by any value type.

Thus, if complex is a value type and T is an unbounded type variable (substitutable by any type at all), the following are illegal and flagged by the compiler.

```
java.util.List<complex> // illegal
java.util.list<T> // illegal
```

On the other hand, if S is a type variable declared as S **extends** java.util.Map, then the following is allowed.

```
java.util.list<S> // OK; S can never be a value
```

Concrete universal types (not type variables) are also fine (including java.lang.Object).

```
java.util.list<Object> // OK
```

Values can then be stored in such a list, because, when members are subsequently accessed from the list, they must be cast back to their original **value** type, which will cause a ClassCastException if the element is **null**.

6.3 Initialization of Values

It has been mentioned that a value type cannot be **null**. This gives rise to some issues when creating either fields of value type or local variables of value type.

It is required that all value classes except ordinals (section 9) and value **enum**s (section 10) shall have a default (zero-argument) constructor whose visibility is at least that of the defining class. Lime will generate such a constructor by default if the class lacks one, even if the class has other non-default constructors. However, if the class declares a default constructor with insufficient visibility, an error is indicated.

A field of value type that lacks an explicit initialization is initialized by code generated by the compiler. The initialization code executes the default constructor of the value class. The ordinals and value **enum**s are discussed in their own chapters.

Because the initialization of fields of value type is accomplished by construction, it follows that any field of **abstract** value type (that is, whose type is a value interface or abstract value class) must have an explicit initialization (otherwise, the compiler will not be able to initialize the field). The explicit initialization must instantiate some concrete value type that is legal for assignment to the field.

Local variables in Java are not initialized by default and are subject to the definite assignment rule. Lime does not change this. Initialization of a local variable of value type is straightforward, however, given the default constructor:

```
fooValue x = new fooValue();
```

Due to the semantics of values, it is invisible whether **new** in this case really produces a new instance or not; the keyword is used to indicate a construction and not necessarily an allocation.

6.4 Type-checking Value Types

In order to ensure that the objects of value types are truly immutable, we must check certain additional properties for value types, in addition to the the rule that they may not

have ordinary reference supertypes.

Each instance field of a **value** type must be **final**, and must be of a **value** type. The keyword **final** is assumed in the definition of value types and is inserted by the Lime compiler. Compile-time checks make sure that uninitialized **final** fields of values are never observed outside the innermost constructor that is responsible for initializing those fields.

To enable compile time checking of the safe initialization of **final** fields, Lime requires the following.

1. A *safe constructor subroutine* is defined as a **private** or **final** method that does not leak **this**.
2. To *leak* **this** means any of the following
   (a) to store **this** in any field
   (b) to pass **this** explicitly to any method
   (c) to invoke an instance method of the object (implicitly passing **this**) when that method is not a safe constructor subroutine.
3. The innermost constructor of a **final** value class must not leak **this** before initializing all final fields.
4. No constructor of a non-**final** value class may leak **this** at all (allowing such leakage would require a subclass to leak **this** illegally when making the required **super** constructor call, which would no longer be a safe constructor subroutine).
5. The value of any blank final field prior to its first assignment in an innermost constructor is always the default value of the field. For ordinary reference fields, this is **null**. For value fields, this is the value produced by the default constructor of that type as described in section 6.3.

These rules are stricter than Java's standard rules (which merely say that the innermost constructor cannot return without initializing the **final** fields). In particular, these rules preclude the definition of recursive value types with cyclic instances.

Finally, methods finalize(), notify(), and notifyAll() can never be called on objects of value types. Objects of value types have no storage identity, thus these methods do not make sense for value objects.

6.5 Universal Classes and Interfaces

As previously discussed, when it is desirable to define an interface that could be implemented by either an immutable or a mutable object, the **universal** modifier is used:

```
public universal interface Gettable<T> {
    T get();
}
```

When implementing functionality that will be shared by value and reference classes, the **universal** modifier is applied to a class definition:

```
public universal abstract class Sized {
    protected int size;
    public Sized(int size) { this.size = size; }
    public int size() { return size; }
}
```

Universal classes are often abstract but this is not a hard requirement. They may only have fields that would be legal in a value, and such fields are implicitly **final**, just as in a value. Note that the fields of a **universal** class must be **value** types, not just **universal**.

On the other hand, **universal** classes have referential identity, like ordinary reference types, and expressions of **universal** type may be **null**.

Like values, **universal** classes may not define **synchronized** methods or use the **synchronized** statement in any method, nor may they have a finalize method. The java.lang.Object methods that already violate this principle (e.g. wait) are screened dynamically.

In addition to Object, a number of Java interfaces are considered a priori to be **universal** types. The pH tool (section 20.4) should be run on a given Java class library implementation to be sure that the listed types do not have non-**universal** supertypes. It is easy to establish this by inspection as well.

```
    // java.lang
Object
CharSequence
Appendable
Cloneable
Readable
Runnable

    // java.util
RandomAccess
Formattable

    // java.io
Serializable
Closeable
DataInput
java.io.Externalizable
Flushable
ObjectInput
```

```
ObjectInputValidation

    // java.net
CookiePolicy
FileNameMap

    // java.security
Guard
Key
PrivateKey
PublicKey

    // java.util.logging
Filter

    // java.util.regex
MatchResult
```

6.6 Generated methods

A **value** class will have certain standard methods generated for it to provide behavior consistent with value semantics. In no case, however, is a method generated if the method is explicitly coded, either in the class itself or a superclass (other than Object, whose methods are generally unsuitable for use in values).

If no equals() or hashCode() method has been explicitly coded, an equals() method is generated consistent with the behavior of the == operator for values. A hashCode() method is generated consistent with equals() and based only on the contents of the method. If the compiler finds that one of these two methods has been provided but the other remains to be generated, it will generate only the method that wasn't provided but will issue a warning that consistency of the result is not guaranteed.

If no toString() method (or tostring(), as described in section 11) has been explicitly coded, then toString() and tostring are generated to produce a result similar in appearance to the default toString() in java.lang.Object but without using object identity in the result (all values with the same type and contents will have the same string representation).

The methods generated for **value** types are always **local** (see section 14).

No visible methods are generated for **universal** classes, but the fields of such classes participate correctly in the semantics of any **value** subclasses that inherit from such classes; to support this, the compiler may generate invisible methods in the **universal** classes.

6.7 Special Rules for Assignment of Null

To enforce the invariant that no value may ever be **null**, Lime requires some care in the use of the **null** literal in any "assignment" context (this includes explicit assignment, initialization of fields and variables, implicit assignment when passing an argument to a method, and implicit assignment when returning a result from a method).

If the static type to which assignment is occuring is a value type, the assignment is illegal. If the assignment is to an ordinary reference type, the assignment is legal.

If the assignment is to a **universal** type, the legality depends on whether the type is explicit or is a type variable. If the type is explicit, the assignment is legal. However, if the type is a type variable, the assignment is illegal. Such assignments require an explicit cast. This is required since the type variable may later be instantiated with a value type.

```
class Foo<T> {
  T badget() { return null; } // illegal
  T goodget() { return (T) null; } // ok
}
```

The badget example must be prohibited because the type variable T (whose upper bound is the universal type Object) can be instantiated with a value type, for which assignment of **null** is illegal. The goodget example is allowed since the cast to T will produce a ClassCastException at runtime if the type of T is a value type.

6.8 Next and Previous Operators for Values

To make it easier to iterate with values and to support ranges and ordinals, Lime provides the prefix operators −−− ("previous") which returns its numeric operand minus 1, and +++ ("next") which returns its numeric operand plus 1. These operators differ from ++ and −− in that they exist only in prefix form and do not imply any mutating side-effect. The next and previous operators may be given user-defined implementations in any class, while the Java pre- and post- increment and decrement operators may not. However, for any type that implements next and previous, the Lime compiler will accept ++ and −− in either position, will use +++ or −−− to obtain the correct value, and apply the correct mutating side-effect on a variable or indexed expression.

6.9 The Top of the Value Type Hierarchy

A number of useful value interfaces are defined at the top of the value type hierarchy. Lime language features make use of these in various ways as will be discussed in individual sections of this manual.

```
public universal interface Glocal {
  // Explained later
```

```
}
public universal interface Local extends Glocal {
  // Explained later
}
public value interface Value extends Local {
  // No visible methods but value behavior is implied
}
public value interface comparable<T extends Value> extends Value {
  boolean this < (T that);
  boolean this <= (T that);
  boolean this >= (T that);
  boolean this > (T that);
  int compareTo(T that);
}
public value interface dense<T extends dense<T>> extends comparable<T> {
  T this +++;
  T this ---;
}
public value interface bounded<T extends bounded<T>> extends dense<T> {
  public T this + (T that);
  public T this - (T that);
  public int ordinal();
  public T valueOf(int from);
}
public abstract value class ordinal<T extends ordinal<T>> implements bounded<T> {
  // No new methods but all ordinal types extend this
}
public value interface numeric<T extends numeric<T>> extends comparable<T> {
  T - this;
  T this - (T that);
  T + this;
  T this + (T that);
  T +++ this;
  T --- this;
  T this * (T that);
  T this / (T that);
  T this % (T that);
}
public universal interface PseudoRing<T extends PseudoRing<T>> {
  T this & (T that);
  T this | (T that);
  T this ^ (T that);
```

```
}
public value interface logical<T extends logical<T>> extends PseudoRing<T> {
  T ~ this;
}
public value interface integral<T extends integral<T>>
    extends numeric<T>, dense<T>, logical<T>
{
  T this << (T that);
  T this >> (T that);
  T this >>> (T that);
}
```

Operators, rather than named methods, are used in the definitions of many of these types to ease the integration of user-defined value types with primitive types (see section 6.10).

The Glocal and Local interfaces are explained in section 14.5.

The comparable interface is the value version of Comparable. Its operations are assumed to be implemented consistent with some order. Note that only the inequalities must be implemented by an implementing type, since the compiler gives all values default semantics for == and !=.

The dense interface adds next and previous operations which are assumed to be implemented such that there are no values of the type between x and ++x or between x and --x.

The bounded type captures the ability of a type to be used as the indexing and bounding type of a bounded array (see section 12) and also in some other contexts (e.g. abbreviated range syntax, explained in section 8). It is sufficiently important that it is discussed in a section by itself (see section 7). The ordinal type is special to the ordinal types (see section 9).

The numeric and integral types cover the remaining numeric operators. Those operators that are typical only of integral types are segregated in a subclass so that, for example, floating point numbers would not have to implement them.

6.10 The Role of the Primitive Types

Lime considers that primitive types *are* value types. However, this leaves the question of exactly where they fit in the hierarchy of value types descended from lime.lang.Value or whether they fit there at all. For compatibility with Java, the names of the primitive types remain as keywords, and the specific semantics of the primitives (e.g. numeric promotion) are unchanged. They are still base types, and convert to and from reference types only in circumscribed ways (e.g. autoboxing, plus interoperation between **int** and the bounded types, as discussed below).

However, in order to participate as type arguments of parameterized types, the primitive types act as if they fit into the standard value type hierarchy as follows.

```
class boolean implements Value {}
class float implements numeric {}
class double implements numeric {}
class byte implements integral {}
class short implements integral {}
class char implements integral, bounded {}
class int implements integral {}
class long implements integral {}
```

As the interfaces in question are mostly defined in terms of operators and not named methods, this "metaphorical inheritance" should be easy to maintain when writing generic types that might be parameterized by primitive types.

There is an implicit conversion between all types that implement bounded and the **int** type. Lime also supports an explicit cast from **int** to any bounded type (the cast uses the bounded type's valueOf method to accomplish this). Note that this is *not* saying that the bounded types are subtypes of **int**, only that they are interconvertable.

The pseudo-hierarchy shown above is also used to decide the eligibility of primitive types to be used in certain other constructs new to Lime where types of particular capabilities are required (see, for example, "ranges" in section 8).

Currently, of the primitive types, only **char** implements bounded because only **char** has a natural first and last that corresponds to its MIN_VALUE and MAX_VALUE. In keeping with the reality that primitive types only inherit value interfaces in a metaphorical sense, the expressions **char**.first, **char**.last and **char**.ordinal() are not legal.

6.11 Java Compatibility

The ability to pass Lime value types to Java methods and constructors is gated by the ability of Java to express matching types in the declarations of those methods and constructors. Primitive types (which, conceptually, are values) can of course be passed. Any value type can be passed by casting it to Object, since that is also a Java type. Bounded types can be passed by casting them to **int**. Note also the restriction against instantiating Java generic types with value types (section 6.2).

6.12 New Grammar

See section 9.1.1 of the Java Language Specification for `InterfaceModifier` and section 8.1.1 for `ClassModifier`.

```
ClassModifier          ::= LimeValue
InterfaceModifier      ::= LimeValue
InterfaceModifier      ::= 'universal'
LimeValue              ::= 'value'
```

See section 15.15 of the Java Language Specification for `UnaryExpression`.

```
UnaryExpression         ::= LimeNextExpression
UnaryExpression         ::= LimePreviousExpression
LimeNextExpression      ::= '+++' UnaryExpression
LimePreviousExpression ::= '---' UnaryExpression
```

Chapter 7

Bounded Types

Lime defines a family of "bounded types", representing constrained integer ranges (with, perhaps, additional behaviors as well). These types are useful in Lime because they can be used to iterate over or index the bounded arrays (described in section 12).

A bounded type is indicated when a type implements the bounded interface (see section 6.9), directly or indirectly. All bounded types must adhere to additional rules in order to achieve their expected semantics.

There is an implicit conversion from all bounded types to **int** and it is always legal to explicitly cast **int** to any bounded type. The behavior of the implicit conversion to **int** is defined by the bounded type's ordinal method and the behavior of an explicit cast from **int** to a bounded type is defined by the behavior of the bounded type's valueOf method, applied to any instance.

For any bounded type X, the following invariants must hold.

```
new X().ordinal() == 0;
X x; x.valueOf(0) == new X();
X x; X y; int n; x.valueOf(n) == y.valueOf(n);
X x; x.valueOf(x.ordinal()) == x;
```

That is, the **int** equivalent of the default instance is 0, the result valueOf method is sensitive only to its argument and not to its receiving instance, and the valueOf and ordinal methods are inverses for those **int** values that ordinal can produce.

Every *non-abstract* bounded type must additionally define a **static final int** field whose name is size. This field must capture the number of distinct instances that the bounded type can have (two instances a and b are distinct iff a != b; recall that value comparison is field-wise and does not compare object identities).

The size field of a bounded type must be a compile-time constant as defined in section 15.28 of the Java Language Specification (3rd edition).

The valueOf operator of a bounded type must accept all **int** arguments and produce a *range-corrected* result. That is, for any X x and **int** n, **int** r = x.valueOf(n).ordinal() is defined so that r >= 0 && r < size. For non-negative n, r == n % size and for negative n, r == n % size + size;

The operators +, −, +++, −−− must then be defined so that they wrap at size and at zero. More precisely, the following invariants hold.

```
X x; X y; /* throughout the following */
x + y == x.valueOf(x.ordinal() + y.ordinal());
x − y == x.valueOf(x.ordinal() − y.ordinal());
+++x == x.valueOf(x.ordinal() + 1);
−−−y == x.valueOf(x.ordinal() − 1);
```

The invariants listed above are not statically checkable for all possible ways of implementing the bounded interface. However, the compiler will, in practice, find many errors in simple cases and will be able to prove, in simple cases, that the invariants are met. For ambiguous cases, the compiler may issue a suppressable warning, which places the ultimate responsibility for checking on the programmer.

If X is a non-abstract bounded type then the following special selectors, which superficially resemble static fields, are defined, as follows.

- X.first provides the instance **new** X().
- X.size provides the value of the size field of X. It also has another use in certain contexts, discussed in section 9.
- X.last provides the instance X.first.valueOf(size−1).
- X.range provides the range X.first::X.last.

The special selectors have the property of being virtually dispatched. So, if a type variable N, in a Lime generic type, becomes bound to the concrete bounded type X in some instantiation of the generic type, then the N.size will access the field X.size, and the other special selectors will be resolved according to the type X. Note that if these were real fields, this virtual dispatch would *not* happen (see section 2.6).

The special selector expression N.size, where N is a type variable, gives an accurate answer at runtime but is not considered a constant (since it depends on a variable, albeit a type variable). On the other hand, X.size, where X is a concrete bounded type, is a normal compile-time constant expression.

Chapter 8

Ranges

Lime provides programmers an easy way to iterate over subranges of many kinds of values (specifically, those that implement dense). This is done via the binary operator, "::". Given the expression x :: y, if the type of x provides an implementation of :: (see section 5), it is used. Otherwise, the result will be an object of lime.lang.range<T>. The types of both x and y must implement dense, else a type error is indicated. The type T is chosen as the least upper bound type of x and y (ie dense or a subtype thereof).

range<T> implements the lime.lang.Iteratable<T> interface, which is Lime's value-friendly variant of Java's java.lang.Iterable. Its name is distinct because both java.lang.* and lime.lang.* are implicitly imported and sharing a simple name would cause confusion. An Iteratable<T> type has an iterator() method that returns the value-friendly lime.util.Iterator<T> instead of the Java equivalent. In Lime, a "for-each" style loop (introduced in Java 5) can use either kind of Iterator. So, a range can be used as the "collection" in a for-each style loop.

For example, the following code defines a loop over the range of values greater than or equal to bit.zero, and less than or equal to bit.one:

```
for ( bit b : bit.zero :: bit.one ) { System.out.println(b); }
```

When a range<T> is created with the :: operator, it is assumed to be an ascending range. Thus "0::5" is an ascending range from 0 to 5 (inclusive), but "5::0" is an empty range. The reverse() method is used to produce a descending range, as in "(0::5).reverse()".

In Lime programs, programmers often need to iterate over the entire range of possible values of a type. A convenient shorthand is provided for this purpose, provided the type in question implements bounded. For example, **for** (bit b){...} is equivalent to **for** (bit b : bit.first :: bit.last {...}. Such a default range is always an ascending one.

Note that all of the primitive integral types implement dense and **char** implements bounded, so they can participate in this new syntax.

8.1 New Grammar

See section 15.17 of the Java Language Specification for the context of the following.

```
MultiplicativeExpression ::= LimeRange
LimeRange                ::= MultiplicativeExpression '::'  UnaryExpression
```

See section 14.14.2 of the Java Language Specification for the context of the following.

```
EnhancedForStatement     ::= LimeForStatement
LimeForStatement         ::= 'for' '(' Type LimeForVariable ')' Statement
LimeForVariable          ::= IDENTIFIER
```

Chapter 9

Ordinals

So far, the bounded types are limited to the primitive types plus types that a programmer might define. Lime provides a way to generate arbitrary bounded types whose size is any value from 1 to Integer.MAX_VALUE using the type constructor **enum**<N>. N must be a compile time constant.

The parameter expresses the number of elements (so last is $N - 1$). For instance,

```
enum<4> a;
final int size = 7 * (int) Math.PI;
enum<size> b;
```

All ordinal types implement the ordinal abstract class, which is a specialization of the bounded interface. User-written code is not permitted to extend the ordinal class, which can then be used in the declaration of type variables when an ordinal type is required.

The values of an ordinal type employ the *ordinal literals* which (in the most general syntax) take the form InS, where I is a non-negative integer in decimal notation giving the ordinal() of the ordinal and S is a positive integer in decimal notation giving the size of the ordinal type. So, for **enum**<4> the literal values are 0n4, 1n4, 2n4, and 3n4. Note that 4n4 is an illegal literal, even though the valueOf method of **enum**<4> would accept the integer 4 and produce 0n4.

The size portion of an ordinal literal can be omitted, implying a size one greater than the value. Thus, 2n means the same as 2n3. Due to the widening conversions accepted by the ordinal types, it is usually safe (although arguably unclear in some contexts) to omit the size suffix. That is, 2n in a context that expects 2n4 will be implicitly widened as described below.

In addition, assignments and initializations of ordinal-typed fields and variables permit an analog of the *narrowing primitive conversion* defined in the Java Language Specification (section 5.2). Just as you are allowed to say

```
short x = 35;
```

even though 35 has **int** type, you are also allowed to say

```
enum<4> x = 3;
```

The requirement is that the right hand side must be a compile time constant expression that is the range of the ordinal. This feature extends to array initializers.

There is a hierarchy of conversions between the ordinal types. If $M < N$ then there is an implicit widening conversion from **enum**<M> to **enum**<N> and **enum**<N> may be explicitly cast to **enum**<M>. The semantics of these conversions are just a special case of the semantics of conversions between all bounded types and **int**. The source type is first made into an **int** by calling its ordinal method, then the result is converted to the target type using valueOf. However, if the target ordinal is wider than the source ordinal this is done without requiring any explicit cast. Note that requirements of the valueOf method of a bounded type cause narrowing conversions between ordinals that are powers of two to retain the low-order bits, just as with the integral primitive types.

Ordinals can also be converted to Object like any other type. In this case, they are boxed immediately and not first converted to **int** and then autoboxed, so that when their type is interrogated the original ordinal type is reported.

Ordinals can be used in **for** statements as range generators, just like any dense type (as ordinals are also bounded, the compact **for** syntax is also available).

```
for (enum<4> i) x[i] = 666;
for (enum<4> i : 1::2) x[i] = 666;
```

Ordinal types are **final**, and do not allow user-defined methods or constructors.

Ordinal types participate in the value initialization convention described in section 6.3. Fields are initialized to the first value of the ordinal (denoted by integer literal 0) and variables can be initialized using either integer literals or first or last notation.

Since ordinal types implement bounded, the operators $++$+, $--$−, + and −, as well as all numeric comparison operators, are defined for them.

9.1 Shorthands for Ordinal Types.

When an ordinal is used as a type argument to a generic, the surrounding **enum**<> portion can be dropped. So for instance, the following

```
unsigned<enum<32>> myNumber;
```

can be written

```
unsigned<32> myNumber;
```

When a qualified name ending in .size (e.g. N.size) is used as a type argument to a generic, and the portion before .size denotes a bounded type, then, rather than interpreting the dotted name as an expression (which would not be a compile-time constant if the N in N.size were a type variable), Lime instead interprets N.size as the ordinal type whose size is the same as the bounded type. This shorthand is useful when instantiating generic types that expect ordinal type arguments, while using a type parameter that is constrained only to be bounded. For instance, the following is allowed if N is a type variable constrained to any bounded type.

unsigned<N.size> myNumber;

This second shorthand is specially useful when constructing generic types that use the Lime bounded array types (described in section 12).

9.2 Java Compatibility

Although ordinal types don't exist in Java, the implicit widening conversion to **int** means that ordinals can be passed as **int** to Java methods and constructors, and methods that return **int** can be preceded by an explicit cast for assignment to an ordinal variable. By composition with other numeric promotion rules, ordinals will interoperate with all Java numeric primitive types, either with or without an explicit cast.

9.3 New Grammar

See section 4.3 of the Java Language Specification for `ClassOrInterfaceType`.

```
ClassOrInterfaceType ::= LimeOrdinal
LimeOrdinal          ::= 'enum' '<' ConstantExpression '>'
```

See section 4.5.1 of the Java Language Specification for `ActualTypeArgument`.

```
ActualTypeArgument     ::= LimeAbbreviatedOrdinal
LimeAbbreviatedOrdinal ::= ConstantExpression
```

The ordinal literal is defined by a straightforward extension to the lexical grammar in section 3.10 of the JLS.

```
OrdinalLiteral         ::= Digits OrdinalTypeSuffix
OrdinalLiteral         ::= Digits OrdinalTypeSuffix Digits
OrdinalTypeSuffix      ::= 'n'
OrdinalTypeSuffix      ::= 'N'
```

Chapter 10

Value Enums

A value **enum** behaves like an ordinal type in which each possible instance is given an explicit name. However, the declaration mechanism is quite different and resembles the ordinary non-value **enum** types of Java. In addition, the **value enum** types do not extend the ordinal class, although they do implement the bounded interface.

The following is a user-defined representation of type bit, with two possible values, bit.zero, and bit.one:

```
public value enum bit { zero, one; }
```

Recall that Java **enum**s extend a particular generic class java.lang.Enum. In Lime, non-value **enum**s continue to do so. Value **enum**s, on the other hand, extend lime.lang.valueEnum. That class is itself a value class and implements bounded. As a consequence of implementing bounded, a value **enum** has the expected special selectors first, last, size, and range.

There is an implicit conversion from any **value enum** to the ordinal type with the same number of members. By extension, there is an implicit conversion to any ordinal type with more members or to **int**. This is just a special case of the general rule that any bounded type is interconvertable with **int** via its ordinal and valueOf methods. But, as with the ordinals, a widening conversion from a **value enum** to an ordinal of equal or greater cardinality requires no explicit cast even if conceptually it is mediated by **int** methods. There is no implicit conversion between different value **enum** types, no matter what members they may have. However, explicit casts between **value enum**s are defined and behave as if the source type was first cast to **int** and then to the target type.

As with Java enums, it is illegal for a **value enum** to have a superclass (or superenum) although it may have superinterfaces (as long as they are **value** or **universal**). When a **value enum** is cast to Object it is boxed directly and does not first convert to ordinal or **int**.

Unlike Java **enum**s, a value **enum** cannot define constructors and gets only the constructors generated for it. It must not explicitly initialize its constants (thus, it cannot implicitly subclass itself by specializing its one default constructor). All **value enum** types are therefore implicily final (regular Java enums may be non-final if they add class bodies in constant initialization, implicitly subclassing themselves). A value enum, like any value, cannot contain mutable fields.

As with Java's mutable enumeration classes, value enums can be used in switch statements and as the return types of annotation methods.

10.1 Default Values

Fields of value **enum** type are implicitly initialized to the first value. Local variables can be initialized using any of the enum's named members, or first or last. For example, in the field declaration bit b; variable b has the default value `bit.zero`.

10.2 Bit Literals

The predefined **value enum** type lime.lang.bit is given some additional syntactic sugar by the Lime language in recognition of the importance of bits in hardware. In particular, the bit literals 0b and 1b are recognized as equivalent to the standard **enum** syntax bit.zero and bit.one.

10.3 Java Compatibility

The **value enum** types have an implicit widening conversion to **int** and an explicit narrowing cast from **int**. Therefore they can be passed to Java code exactly as the ordinal types can.

10.4 New Grammar

The **value** keyword is covered in section 6.12. An **enum** in Java uses the `ClassModifier` production just like a class.

The bit literal is obtained as a straightforward extension to the lexical grammar in section 3.10 of the Java Language Specification.

```
BitLiteral      ::= Digit BitTypeSuffix
BitTypeSuffix   ::= 'b'
BitTypeSuffix   ::= 'B'
```

Note that a `Digit` includes digits besides 0 and 1 but only 0 and 1 are legal; other "bit literals" are rejected by semantic analysis.

There are no other enum-specific grammar changes.

Chapter 11

Strings

Lime has its own native string class, lime.lang.string. It is very similar to Java's native java.lang.String class, but is an instance of Value, and can therefore be contained in other value classes and passed between tasks (see Section 15).

Lime's string class also has significant additional functionality. Lime strings support the Iteratable, Indexable, and comparable interfaces, so the following are legal:

```
string s = "foo";
for (char c: s) { ... }
char f = s[0];
if (s < "food fight") { ... }
```

Lime strings also support data-parallel operations via the Collectable interface, as described in Section 18.

Instead of Java's toString() method, Lime programmers should generally implement a tostring() method which returns type string.

11.1 Java Compatibility

Lime strings and Java Strings can be used interchangeably, in a manner exactly analogous to the treatment of **int** and Integer in Java:

```
string ls = "foo";
String js = ls;
string ls2 = js;
String jn = null;
string ln = jn;
```

Note that this also means that the final statement will generate a run-time NullPointerException, since Lime strings are not allowed to be null.

Concatenating a Lime string and a Java String always produces a Lime string.

Note however that string constants are still of type java.lang.String (in order to maintain classfile compatibility). This is mostly invisible except that **var** str = "foo" will cause str to

have the inferred type String instead of string.

11.1.1 ToString Conversion

Lime programs should generally define a tostring() method returning a string. For Java compatibility, the compiler will automatically generate the following method when tostring() is defined:

```
public final String toString() { return tostring().toString(); }
```

The generated method will have the same locality modifier (see Section 14) as the explicitly defined tostring() method.

However, for compatibility, programmers may also choose to define a toString() method, in which case the following method will be automatically generated:

```
public final string tostring() { return new string(toString()); }
```

Note that it is an error to define both tostring() and toString() methods in the same class.

Finally, tostring() is treated as though it is a method of Object: it may be invoked on objects that lack a tostring() method, in which case (expr).tostring() is equivalent to **new** string((expr).toString()).

11.1.2 Equality Relationships

Lime strings follow the standard rules for "==" equality of value classes: it is based on content rather than identity. However, since String is a reference type, it will never be "==" equal to a string. Thus:

```
(new string("foo") == new string("foo")) == true
(new string("foo") == new String("foo")) == false
(new String("foo") == new String("foo")) == false
(new String("foo") == new string("foo")) == false
```

On the other hand, string's equals() method compares for content equality – but note that String's equals() method is unchanged. Thus:

```
(new string("foo").equals(new string("foo")) == true
(new string("foo").equals(new String("foo")) == true
(new String("foo").equals(new String("foo")) == true
(new String("foo").equals(new string("foo")) == false
```

In general, this should not affect programs unless string and String objects are, for example, being used interchangeably as keys in a hash table where the key type is Object.

Chapter 12

Arrays

Arrays are a critical building block in Lime, as in many programming languages. In Lime, array behavior is revised (compared to Java) with three goals in mind.

1. It must be possible to declare arrays that behave as values (are immutable and can't be null). It should already be obvious how to declare an array all of whose elements are values, but that is not the same thing.

2. It must be possible to establish static bounds on the sizes of arrays in contexts where that is important, so that data rates are statically known or the ArrayIndexOutOfBoundsException is guaranteed not to happen. This is critical to synthesizing simple and efficient code for execution in hardware.

3. Arrays in Lime must have behavior which meshes with other features, such as expanded generics (section 2), collective operations (section 18), and ranges (section 8). In Lime, arrays are also lime.lang.Iteratable types, making them more like collections.

To meet this goal, Lime provides two fundamental array types (value arrays and mutable arrays) and a static type distinction applying to either of these fundamental types which expresses either dynamic bounds checking (in which ArrayIndexOutOfBoundsException is possible) or static checking (in which the exception is precluded).

More precisely, consider the following.

```
int[] w; // A mutable unbounded array
int[[]] x; // An immutable unbounded array
int[4] y; // A mutable bounded array (only arrays of size 4 allowed)
int[[4]] z; // An immutable bounded array
```

As in Java, all arrays, once created, have a fixed size. Exploiting this fact allows us to define the unbounded array types as abstract supertypes of the corresponding bounded types. So, all of the following are accepted by the compiler and will also succeed at runtime.

```
int[] u;
int[4] b4 = new int[4];
int[5] b5 = new int[5];
u = b4;
u = b5;
```

The variable u is unbounded and will accept any array size. A cost of using u, however, is that an indexing expression like u[i] might cause an exception depending on the actual size of the array and value of i.

The variable b4 accepts only arrays of size 4. An indexing expression like b4[2] will be accepted by the compiler and cannot cause any exception. On the other hand, the expression b4[i] may be rejected by the compiler unless i is a constant expression known to be in range or falls in the category of *confined integer expressions* discussed below. Any such expression accepted by the compiler will be guaranteed not to produce an ArrayIndexOutOfBoundsException at runtime.

The following assignments would be illegal.

```
b4 = u; // illegal
b5 = u; // illegal
b5 = b4; // illegal
b4 = b5; // illegal as well
```

The compiler does not allow an array of unknown size or an array whose size is not the same as the target to be assigned without a check. It should already be clear why b5 = b4 is illegal, since it makes a later array bounds violation possible. We also make b4 = b5 illegal, even though it is safe from the narrow perspective of bounds checking, because bounded arrays allow for various size-sensitive optimizations that could fail unless the compiler can count on the exact size being known.

The compiler will accept the following, but a ClassCastException might result if the actual array has the wrong size.

```
u = new int[5];
b5 = (int[5]) u; // checked dynamically; ok
b4 = (int[4]) u; // throws ClassCastException
```

Note that bounded array types must have a positive size. That is, the following is illegal:

```
int[0] x; // illegal
```

Thus, while most unbounded arrays may be cast to a corresponding bounded type, this is never true of zero-length arrays.

All the rules discussed up to now would apply equivalently if the double brackets "[[ ]]" were used instead of single ones (that is, if all the arrays were value arrays instead of mutable ones). However, the value arrays and mutable arrays are not interconvertable with each other, either with or without a cast. Such conversions, when needed, are performed by constructors as discussed in Section 12.5.

To achieve the best level of hardware synthesis and performance, Lime bounded arrays are designed to avoid not only indexing exceptions but also ArrayStoreException. Consequently, a mutable bounded array is *not* subject to covariant subtyping through its element type. Thus, the following would be illegal.

```
Integer[] ui = new Integer[4];
Integer[4] bi4 = new Integer[4];
Number[4] bn4 = (Number[4]) ui; // rejected by compiler
bn4 = (Number[4]) bi4; // ditto
```

The other three array types accept covariant subtyping. For value arrays, such subtyping is exception-free because storing into a value is already precluded statically. Covariant subtyping of mutable unbounded arrays is accepted, with the possibility of ArrayStoreException, both because that's how Java does it and because such arrays are already subject to other runtime exceptions.

Arrays in Lime have the same size limit as in Java; the maximum size array that can be created is one whose length is Integer.MAX_VALUE.

Most generally, the construction of any dimension of an array type takes (between square brackets) either a positive integer constant expression, or a *type*, which must be an ordinal type.

That is, all three of the following are equivalent:

```
int[4] x;
int[enum<4>] x;
typedef e4 = enum<4>;
int[e4] x;
```

Also, if V is an in-scope type variable of ordinal type, then the following is also legal.

```
int[V] x;
```

In addition, due to the special shorthand described in section 9.1, if B is an in-scope type variable of any bounded type, then the following is also legal.

```
int[B.size] x;
```

It is true that the exact size of the resulting type is unknown at the point of declaration but the type is guaranteed to take on a definite size in each instantiation.

12.1 Range Indexing

In Lime, an array can be indexed by an **int** range, with a result that is another array. So, for example, x[0::3] is a syntactically legal expression and it is semantically legal as well if x is an unbounded array or a bounded array of size at least 4.

Lime also permits an array to be indexed by a bounded range (ie. range<? **extends** bounded<?>>). The semantics are defined operationally as if a bounded range r is first

converted to r.start.ordinal()::r.end.ordinal() (which has the type range<**int**>) and then indexing accordingly. The motivation for this feature is to achieve symmetry in how single indexing works and how range indexing works in the light of the *confined expression* concept described more fully in Section 12.4.

The result of indexing by a range is a bounded array if the following conditions are met.

1. The indexing expression is explicitly constructed in place with ::, not just an expression whose type turns out to be range<**int**> or range<? **extends** bounded<?>>.

2. Both the start and the end of the range are compile-time constant expressions.

3. The range is an ascending one encompassing at least one element.

Otherwise, the result is an unbounded array.

Ordinal and bit constants are treated as if they were "compile-time constant" for the purpose of this decision. This does not imply that they are compile-time constants in every respect.

Range indexing is possible on the left of an assignment as well:

```
u4[0::1] = 0;
u4[0::1] = u5[2::3];
```

In that case, when the target array is bounded then the range selector of the target must be within range, and length of the array being assigned must match the number of elements being assigned. Length errors are found (most generally) at runtime and indicated by ArrayIndexOutOfBoundsException. When the right hand side is a bounded array and the range expression on the left hand side is made up of constant expressions, yielding a manifestly known length, the error will be indicated at compile-time.

12.2 Multidimensional Arrays

Lime array syntax extends to the multidimensional case, where the kind of array at each dimension may be different. There are two special considerations.

1. value arrays (if any) must be innermost (this follows from the fact that values may not contain non-values).

2. if there is more than one value array dimension, there is only one extra set of brackets surrounding all of them.

In the following, x is a mutable unbounded array whose elements are mutable bounded arrays of size 4. The elements of those arrays are, in turn, value arrays of size 4 whose elements are unbounded value arrays of **int**.

```
int[] [4] [[4] []] x;
```

On the other hand, the following are illegal, y because it has value dimensions that are not innermost and z because it does not use a single pair of "value braces" around all the value dimensions.

```
int[] [[4] [4]] [] y; // wrong
int[] [4] [[4]] [[]] z; // wrong
```

Conversions between bounded and unbounded versions of multi-dimensional arrays involve some issues that are initially counterintuitive but which can be understood in terms of basic subtyping rules. First of all, casts and conversions involving just the outermost dimension are always permitted at compile time, as they follow directly from the rules for single-dimensioned arrays.

```
int[4][4] y = ...;
int[][4] x = y;
y = (int[4][4]) x;
```

The cast from x to y is checked exactly as in the single-dimensioned case. In contrast, casts involving a change in an inner dimension are permitted or not according to whether the encompassing array type does or doesn't support covariant subtyping.

```
int[][4] w = ...;
int[4][4] y = ...;
int[4][] z = y; // error
int[][] xx = y; // error
xx = w; // allowed
```

The type **int**[4][] is a mutable bounded array, which, to avoid ArrayStoreException, subtypes invariantly through its element type. The element type of this array is **int**[]; replacement by the subtype **int**[4] is therefore not allowed. This replacement is would be performed by the assignment of y to z (were it allowed) and would be implied (elementwise) in the assignment of y to xx; therefore, those assignments are not permitted. If they were to be permitted, subsequent assignment of a wrong length array to an element of the outer array would either result in loss of type safety or would have to throw an ArrayStoreException.

The type **int**[][] is an unbounded array for which ArrayStoreException is expected; therefore the assignment of w to xx is allowed; a subsequent store of an element array that isn't of length 4 would throw the exception.

If an explicit cast cannot possibly succeed, it is disallowed statically. Thus, an explicit cast from **int**[][] to **int**[4][4] is disallowed statically on the grounds that the former type cannot possibly contain any instance of the latter.

When two multi-dimensional arrays cannot be converted by assignment or casting, they can always be converted (if actually conformable) by copy-construction (see Section 12.5

12.3 Java Arrays

It is infeasible to achieve the expanded semantics of Lime arrays while using the identical representation to that used in Java, hence arrays (like generic types) are an area of tension between the expressivity requirements of Lime and its Java compatibility requirements. This requires Lime to make a type distinction between Java arrays and Lime arrays.

The tilde character (~) is used to denote a Java array, similar to how it is used to denote a Java generic type or method. That is, in the following, x is a Java (not Lime) array.

```
int~[] x;
```

Since Java arrays are less reifiable than Lime arrays (see Section 2.1), their use in a Lime program should be strictly limited to areas of a program that are interoperating intensively with Java code.

The element type of a Java array may not be any value type other than the primitive types.

```
String~[] s1; // ok, String is not a value
string~[] s2; // illegal, string is a value
string~[][] s3; // ok, string[] is not a value
string~[][[]] s4; // illegal, string[[]] is a value
```

The reason for the restriction is that Java does not understand values. If a Java array could have a value element type and such an array was passed to a Java method, it would match some supertype (e.g. string~[] might match Object~[]) and the Java method might store **null** into it, which would violate Lime type safety.

The rule prohibiting values with Java arrays is similar to the rule in section 6.2 concerning the instantiation of Java generics with value types. So, in addition to the illegal cases shown above, the use of a Lime type variable as the element type of a Java array is immediately flagged as illegal unless the type variable precludes substitution by any value type.

The Java array is not implicitly interconvertable or castable to any other kind of array. However, a Lime array can be constructed (making a copy) from a Java array as discussed in Section 12.5 and a Lime array can be converted to a legal Java array (also making a copy) using the toJavaArray method described in Section 12.6.

12.4 Confined Integer and Range Expressions

To fully understand array indexing for Lime arrays, it is necessary to define a category of *confined expressions* of **int** or range<?> type. Intuitively, such an expression is one whose exact value may or may not be known statically but where it is statically known that the value falls in a confined range. A constant expression is always confined but not vice versa. For example, the conversion of any bounded type (constant or not) to **int** yields a confined expression. Greater detail is provided in the following definition:

1. All compile-time constant expressions of integral type, implicitly or explicitly widened or narrowed to **int** are confined expressions since their exact value is known.

2. Generalizing rule 1, any **final** variable or **final** field of integral type (not necessarily static) explicitly initialized by a confined integer is considered to be a confined integer at any use site, since its value cannot change. Note that this property does not extend to blank **final** fields, even if they are everywhere initialized to a confined integer, since determining their value statically is not always possible.

3. All expressions (constant or not) of a type that implements bounded (including ordinals, value enums, and user-defined bounded types) yield a confined integer when converted to **int**. Recall that any bounded type is implicitly converted to **int** as needed. The **int** value yielded by converting any expression of type N **extends** bounded<N> is confined to lie in the range 0::N.size−1.

4. Expressions whose terms are confined integer expressions (only, with no other fields, variables, or function calls) combined by arithmetic operators are themselves confined integer expressions.

5. Other integer-valued expression are not considered to be confined. No data flow is considered in making this distinction. So, if x has ordinal type, then the **int** resulting from the implicit conversion of x in y[x] is confined. Similarly, the **final int** f in the sequence **final int** f = x; y[f] is confined. However, the non-final **int** u in the sequence **int** u = x; y[u] is not confined even though the compiler could presumeably discover its limited range of possible values.

The fact that an expression is confined does not make it legal to use as an index. The upper and lower limit of the range to which it is confined must be in range for the type it is indexing. Thus, with **enum**<8> x and **int**[6] y, the expression y[x] will be rejected by the compiler even though the x is confined because its range includes 6 and 7, which are illegal indices for **int**[6]. Note that the range of a confined expression can "widen" due to arithmetic: the expression z[x∗x∗x∗x] is illegal for indexing **int**[2000] z since the expression's upper limit is 2401.

There are also *confined range expressions* which are defined as follows.

1. If x and y are both confined integers (most generally these could be expressions, not just simple names) then both x::y and y::x are confined ranges.

2. If p and q are both expressions of (any) bounded type, then both p::q and q::p are confined ranges. The type of the range will be range<B> where B is either p or q depending on which has the greater size. As the semantics of indexing with this kind of range is defined by converting each member from start to end to an **int**, the range will function as a confined integer range for the purpose of bounds checking.

3. If a **final** variable or field of type range<**int**> or range<? **extends** bounded<?>> is explicitly initialized with a confined expression, then the variable or field is considered confined at any use site since it cannot change and ranges are values that cannot change.

12.5 Array Creation

Some examples of array creation have already been shown. Lime array creations mimic their Java counterparts. That is, either a size is given as an **int** expression (expressed perhaps as an expression of some other type implicitly convertable to **int**) or the size is implied by the number of members in an array initializer (one or the other but not both). For example, the following.

```
int[] u = new int[i];
int[4] b = new int[] {1,2,3,4};
int[[4]] vb = new int[[4]];
```

Types may also appear in array creation expressions, just as they can in declarations; they must be ordinal types.

```
typedef e4 = enum<4>;
int[[e4]] vb = new int[[e4]];
```

Similarly, type variables that are defined as being of ordinal type (or type variables of any bounded type followed by .size) can be used in creation expressions.

Whether the created array is a value or not is determined by the extra brackets. Whether it is bounded or not depends on the characteristics of the size expression.

- If the size expression is a reference to an ordinal type, the result is bounded.
- If the size expression is a compile-time constant integer expression, or a compile-time constant bounded expression that is implicitly converted to **int**, and the result is positive, then the result is a bounded array.
- If there is no size expression but instead an array initializer (whose size is manifest) is employed, and that initializer has at least one element, the result is a bounded array.
- Otherwise (including all zero-length cases) the result is an unbounded array.

A confined integer expression is not sufficient here since the exact size must be known, not just a range of sizes.

These forms extend to multi-dimensional arrays, as in Java. As in Java, the use of a multi-dimensional array form leads to all dimensions of the array being initialized, with the leaves being initialized to the default value for the leaf type.

Lime arrays accept another form of constructor, in which forms like **int**[[4]] are interpreted as types rather than expressions and are followed by constructor arguments, mimicking object constructors. Consider these examples.

```
int[[4]] vb = ...;
int[4] b = ...;
int~[] ju = ...;

int[4] mvb = new int[4](vb); /* constructs mutable array from value array */
```

```
int[[4]] vmb = new int[[4]](b); /* constructs value array from mutable array */
int[] lju = new int[](ju); /* constructs Lime array from Java array */
```

Any kind of array can be made from any other kind of array as long as no size violation occurs. This is especially useful when converting between mutable and immutable representations of an array. Conversion by construction, unlike conversion by assignment or casting, makes a copy, avoiding what would otherwise be unsafe type changes.

Although changing between bounded and unbounded can most simply be done by assignment or casting, when this change is combined with a change in mutability it may also be done by construction to save steps. All required length checks are performed by the constructor.

Copy-construction of arrays extends to multiple dimensions as long as no size violation occurs at any dimension in forming the result. This can allow conversion in cases that would be unsafe without making a copy, such as converting **int**[4][4] to **int**[][].

A Java array being converted by construction to a Lime array may have an element type that is wider than that of the array being constructed. A ClassCastException or IllegalArgumentException occurs if an element type cannot be narrowed to the target type or if a **null** appears where a value is expected. This liberality is needed to balance the fact that Java arrays may not have a value element type other than the primitive types.

12.5.1 Repeats in Array Initializers

Lime adds a small convenience function to array initialization: repeat counts, as in C.

```
int[] foo = {(100) 6, 0, 1, (2) 6};
```

The length of foo will be 104, with the first 100 elements being 6, followed by 0, 1, 6, 6. The expression between parentheses must be a compile time constant expression yielding a positive integer result, and the semantics of the initializer is precisely as if the element following the repeat count were lexically repeated the stated number of times.

12.6 Arrays as Generic Types

To understand how Lime arrays interact with other Lime features, it is useful understand that the array syntax is (in Lime) syntactic sugar for four generic types in package lime.lang.

1. Array<E> is the unbounded mutable array type, usually represented as E[].
2. valueArray<V **extends** Value> is the unbounded value array type, usually represented as V[[]].
3. BoundedArray<E,B **extends** ordinal<B>> **extends** Array<E> is the bounded mutable array type, usually represented as E[B], with the ordinal type usually abbreviated as an integer constant expression (see Section 9.1).

4. boundedValueArray<V **extends** Value,B **extends** ordinal<B>> **extends** valueArray<V> is the bounded value array type, usually represented as V[[B]], with the ordinal type abbreviated as above.

It is possible to declare arrays using the generic syntax, although usually this is unnecessary and just makes the program harder to read. However, the fact that Lime arrays are also generic types makes it possible to define a variety of additional behaviors which are implemented as methods of the array classes. The array classes also have a series of super-interfaces which abstract various aspects of their behavior. Among those are Iteratable and Collectable which give arrays common behavior with other types in Lime.

It is not possible to create arrays using the generic syntax: the array types have no public constructors. Rather, the array creation syntax shown previously must be used.

Since Lime arrays are instances of a generic type, many operations are possible on them by invoking instance methods.

Lime arrays implement Indexable and Collectable (value arrays implement ValueCollectable) and so can participate in collective operations as discussed in chapter 18).

All Lime arrays provide size() and iterator() methods, making them more like collections (these methods are contributed by the interfaces Sizeable and Iteratable, respectively). Lime arrays also have a .length field for compatibility with Java.

The toJavaArray method has the following signature.

```
public local <T> T toJavaArray(T into);
```

It is a generic method that infers the type of its result from the type of its argument; it always copies into an existing Java array provided as an argument and then returns the result. Just like the similar toArray() method of Java collections, it will reallocate the array if it is not of the right size. For example, given a Lime array x of type **int**[] one produces the **int**~[] equivalent using

```
int~[] y = x.toJavaArray(new int~[x.length]);
// or
int~[] y = x.toJavaArray(new int~[0]);
```

Note that Java arrays are constrained from having a value element type or a type variable element type that could be later substituted by a value type. Thus, the previous example would have failed if you used bit instead of **int**. The Java array must be one capable of receiving the contents of the Lime array without an ArrayStoreException.

Lime arrays can be turned into streams as discussed in chapter 15.7.

The java.util.Arrays class provides useful utilities for arrays, and the System.arraycopy() method is heavily used. These do not work with Lime arrays. Instead, Lime arrays provide the same functionality as instance methods arraycopy, binarySearch, sort, and fill.

There are some other methods that are applicable to arrays that are more usefully discussed in their own sections.

12.7 Bit Array Literals

The type lime.lang.bit is given extra syntactic sugar in Lime due to the importance of bits to hardware programming. In particular, Lime provides *bit* literals (described in section 10.2) and *bit array* literals (described here). The general form of a bit array literal is IbS where I is a sequence of binary digits 0 and/or 1, and S, if present, is a positive integer in decimal form.

A bit array literal denotes a bounded value array of lime.lang.bit. If S is present, the bounding type is the ordinal type of size S. If S is absent, the bounding type is the ordinal type whose size is the length of the literal.

A small irregularity exists because 0b and 1b are bit literals, not bit array literals. Thus, to express an unbounded value array of bits whose length is 1 you need the array initializer syntax ({0b} or {1b}). Otherwise, for example, 10b is equivalent to **new** bit[[2]]({1b,0b}) and 10b6 is equivalent to **new** bit[[6]]({(4)0b, 1b, 0b}) (zero extension on the left). It is illegal for the number of binary digits to exceed the size bound, even if the leftmost digits include enough zeros to avoid loss of significance.

12.8 Java Compatibility

Java does not understand Lime arrays and Lime arrays have no useful Java supertypes other than Object. Therefore, Lime arrays can only be passed to Java as Object, where they remain opaque. To interoperate with Java, one can use Java arrays in lieu of Lime arrays (only recommended for sections of code that interoperate with Java at high frequency). Alternatively, use the toJavaArray method to convert Lime arrays to Java ones and the appropriate array copy constructor to construct a Lime array from a Java one. Both of these transformations copy the contents of the array (deeply, in the case of a multi-dimensional array).

12.9 New Grammar

The `ArrayType` production is defined in section 4.3 of the Java Language Specification.

```
ArrayType        ::= Type '[' ']'
```

In that definition, `Type` includes `ArrayType`, making the definition recursive. To define the Lime extensions, we first alter the production to the equivalent, more convenient, form.

```
ArrayType        ::= ArrayTypeBase Dims
ArrayTypeBase    ::= PrimitiveType
ArrayTypeBase    ::= ClassOrInterfaceType
```

The `ClassOrInterfaceType` production is also defined in section 4.3 of the JLS and `PrimitiveType` in section 4.2. The `Dims` production, although new in this context, also replaces the `Dims` defined in the JLS, section 15.10 (array creation expressions). The `LimeAbbreviatedOrdinal` is defined in this manual in section 9.3.

```
Dims                ::= LimeValueDims
Dims                ::= LimeMixedDims
LimeValueDims       ::= '[' NestableDims ']'
LimeMixedDims       ::= AnyDims LimeValueDims_opt
NestableDims        ::= NestableDim
NestableDims        ::= NestableDims NestableDim
AnyDims             ::= AnyDim
AnyDims             ::= AnyDims AnyDim
AnyDim              ::= NestableDim
AnyDim              ::= JavaDim
JavaDim             ::= '~' '[' ']'
NestableDim         ::= UnboundedDim
NestableDim         ::= ExplicitBoundedDim
NestableDim         ::= ImplicitBoundedDim
UnboundedDim        ::= '[' ']'
ExplicitBoundedDim  ::= '[' BoundType ']'
BoundType           ::= PrimitiveType
BoundType           ::= ClassOrInterfaceType
ImplicitBoundedDim  ::= '[' LimeAbbreviatedOrdinal ']'
```

The `ArrayCreationExpression` of section 15.10 of the JLS is replaced as follows. The `ArrayInitializer` production is from section 10.6 of the JLS, as amended below.

```
ArrayCreationExpression    ::= 'new' LimeArrayDescription
LimeArrayDescription       ::= LimeArrayWithInitializer
LimeArrayDescription       ::= LimeArrayWithNewStyleArgs
LimeArrayDescription       ::= LimeSimpleArray
LimeArrayWithInitializer   ::= LimeSimpleArray ArrayInitializer
LimeArrayWithNewStyleArgs  ::= SimpleArray '(' ArgumentList_opt ')'
LimeSimpleArray            ::= ArrayTypeBase CreateDims
CreateDims                 ::= CreateMixedDims
CreateDims                 ::= CreateValueDims
CreateMixedDims            ::= CreateAnyDims CreateValueDims_opt
CreateValueDims            ::= '[' CreateNestableDims ']'
CreateNestableDims         ::= CreateNestableDim
CreateNestableDims         ::= CreateNestableDims CreateNestableDim
CreateAnyDims              ::= CreateAnyDim
CreateAnyDims              ::= CreateAnyDims CreateAnyDim
```

```
CreateAnyDim               ::= CreateNestableDim
CreateAnyDim               ::= CreateJavaDim
CreateJavaDim              ::= '~' '[' DimExpr_opt ']'
CreateNestableDim          ::= '[' DimExpr_opt ']'
DimExpr                    ::= Expression
DimExpr                    ::= LimeOrdinal
```

Note the need to distinguish between a `Dim` and a `CreateDim`. This is due to the different set of constructs that are legal between the square brackets.

The `ArrayInitializer` in section 10.6 of the JLS is not changed but the list of contained elements, known as `VariableInitializers` is amended as follows.

```
VariableInitializers       ::= LimeVariableInitializer
VariableInitializers       VariableInitializers ',' LimeVariableInitializer
LimeVariableInitializer    ::= Expression
LimeVariableInitializer    ::= ArrayInitializer
LimeVariableInitializer    ::= '(' LimeRepeat ')' SimpleVariableInitializer
SimpleVariableInitializer  ::= Expression
SimpleVariableInitializer  ::= ArrayInitializer
LimeRepeat                 ::= ConstantExpression
```

Chapter 13

Tuples

Tuples are immutable collections of heterogeneous objects. They are written with a "`" (backquote) followed by a parenthesized set of values, such that the backquote and the opening parenthesis are contiguous (no intervening white space). Examples of tuples are

```
`(1, "foo")
`(666)
`(2, 1.14159, new Blah())
```

Tuples consisting entirely of value types are themselves value types; tuples containing non-value types are not values. Two tuple type declarations that employ the same types in the same positions are the same type and will interoperate (structural equivalence).

The scope within which two independent tuple declarations are recognized as structurally equivalent is global (if all member types are **public**), else the package plus subclasses (if all member types are either **public** or **protected**), else the package only (if no member types are **private** or locally declared within a block), else the class (if no member types are locally declared within a block), else the block (if any member type is locally declared).

Tuple type definitions can be used as follows:

```
`(int, String) pair = `(1, "foo");

public `(int, double, Blah) bar() { return `(2, 1.14159, new Blah()); }
```

Tuples must have at least two fields. The singleton tuple and the empty tuple are both disallowed since having them would cause ambiguities when mapping between tuple types and method signatures.

A tuple can also be passed as the sole apparent argument to a method or constructor if the members of the tuple, matched successively against the successive formal parameters of the method or constructor, would be legal arguments. A tuple passed as a sole apparent argument is expanded eagerly prior to method lookup.

```
String f(int a, String b) { return b+a; }
x = `(1, "foo");
f(x)
```

By saying that the tuple must be the sole apparent argument we mean to rule out invocations like g(x, y) where g is a method of three arguments, x is a tuple of two members, and y is some other expression. Lime would flag such invocations as type mismatches. Methods *may* have signatures that use tuples explicitly (in that case, for a tuple to match the signature it would need to have an embedded tuple).

However, a method or constructor with exactly one argument, where that argument has a tuple type, is disallowed by Lime. Such a method, were it to exist, would be inaccessible due to the rule that tuples are expanded eagerly before lookup. On the other hand, the eager expansion means that tuples can match variable arity ("varargs") methods as well.

13.1 Tuple Element Access

The elements of a tuple can be accessed using a dot (".") following by a 0-based integer literal indicating its position in the tuple. Thus:

```
double pi = `(1, 3.14159, "blah").1;
String s = pair.1;
```

Note that constant expressions are *not* allowed for accessing tuple elements. That is, x.(1+2) is illegal.

13.2 Tuple Element Binding

The elements of a tuple may be bound to variables as follows:

```
`(int a, String b) = `(1, "foo");
int x;
String y;
`(x, y) = `(2, "bar");
```

Regardless of whether types are declared inside the tuple-form left-hand-side or outside, all assignments with tuples on the left are statements, not expressions.

Element-wise type compatibility and widening is provided for tuples. Thus the following assignments are legal:

```
`(double, String) x = `(1, "foo");
`(long, Object) y = `(1, "foo");
```

When only some of the elements of the tuple are needed, the anonymous variable "_" (a single underscore character) can be used in order to discard the value. For instance,

```
`(_, String b) = `(1, "foo");
int x;
`(x, _) = `(2, "bar");
```

The "_" (single underscore) is a reserved identifier in Lime.

13.3 Java Compatibility

Tuples may not be passed to Java code. Instances of classes with non-private methods that accept explicit tuple arguments or return tuple results or with non-private fields of tuple type are also not Java compatible. However, classes with private fields of tuple type are Java compatible if they do not violate any other rules. The use of a tuple as the sole argument to a method invocation is allowed for invoking Java methods, since the tuples are expanded at the call site.

13.4 New Grammar

See section 4.3 of the Java Language Specification for `ClassOrInterfaceType`, section 4.1 for `Type`, section 15.27 for `Expression`, section 15.8 for `PrimaryNoNewArray`, section 15.14 for `PostfixExpression`, section 15.11 for `FieldAccess`, 3.10.1 for `IntegerLiteral` and 3.10.2 for `DecimalFloatingPointLiteral`.

```
ClassOrInterfaceType        ::= LimeTupleType
LimeTupleType               ::= ’`(’ LimeTupleTypeList ’)’
LimeTupleTypeList           ::= Type
LimeTupleTypeList           ::= LimeTupleTypeList ’,’ Type
PrimaryNoNewArray           ::= LimeTupleExpression
LimeTupleExpression         ::= ’`(’ LimeExpressionList ’)’
LimeExpressionList          ::= Expression
LimeExpressionList          ::= LimeExpressionList ’,’ Expression
PostfixExpression           ::= LimeTupleAssignmentReceiver
LimeTupleAssignmentReceiver ::= ’`(’ LimeTupleAssigneeList ’)’
LimeTupleAssigneeList       ::= LimeTupleAssignee
LimeTupleAssigneeList       ::= LimeTupleAssigneeList ’,’ LimeTupleAssignee
LimeTupleAssignee           ::= Type IDENTIFIER
LimeTupleAssignee           ::= PostfixExpression
FieldAccess                 ::= LimeTupleFieldAccess
LimeTupleFieldAccess        ::= TupleReceiver TupleProjection
TupleProjection             ::= ’.’  IntegerLiteral
TupleProjection             ::= DecimalFloatingPointLiteral
TupleReceiver               ::= Name
TupleReceiver               ::= Primary
```

Note that the `DecimalFloatingPointLiteral` production is required as part of the definition of `TupleProjection` since a sequence like .2 is recognized lexically to be such a literal. However,

to be legal, a `TupleProjection` must consist of a period followed only by digits that make up a non-negative decimal integer.

The ability to use tuples as the sole apparent argument of a method or constructor invocation does not require new syntax; it is merely a change to the semantics of matching arguments against parameters.

Chapter 14

Local and Global Methods

A *local* method is one that does not write any static fields and does not read any static fields except *repeatable static fields* as defined in Section 14.2. A *global* method is one that may freely access all static fields.

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.

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 **local** and **global** keywords. As the defaults apply even to system-generated default constructors, it may be necessary to write out such declarations in order to change the defaults.

Bear in mind that, unless a constructor invokes another constructor, it implicitly includes the initialization of all instance fields that have initializers. Thus, labeling any such constructor as **local** implies that the initializers of instance fields would be legal inside a **local** method.

Although **local** and **global** are implementation properties and not interface properties, these modifiers are permitted on abstract methods, including the methods of interfaces, since that facilitates separate checking. The methods of a value interface and the abstract methods of a value class are considered **local** by default. All other methods (including the methods of a **universal** classes and interfaces) are **global** by default. In order to facilitate extension of **universal** types by value classes, attention should be given to correct labeling of those methods intended to be **local**.

```
class Foo {
  static int x = 7;
  static final int z = 9;
  int getX() { return x; }
  local int localGetX() { return x; } // error
  local int getZ() { return z; }
}
```

```
value class bar {
  static int y = 9;
  int localGetY() { return y; } // error
  global int getY() { return y; }
  int getZ(Foo f) { return f.getZ(); }
  int getX(Foo f) { return f.getX(); } // error
}
```

Lime permits and encourages the placement of **local** and **global** keywords on **native** methods as well as on those written in Lime. Native methods in Lime may be implemented by IP blocks in hardware or some other back-end-specific mechanism, in addition to the normal JNI implementation for use within a JVM. A different native implementation may be required for each distinct backend. The Lime compiler does not check native code, so it relies on the programmer to correctly distinguish native methods that mutate global state from those that do not. The default for native methods is the same as for methods in general, so native methods of value classes are considered **local** by default.

14.1 Other Restrictions

Thread scheduling is part of the global static state of the system. Therefore, synchronization is not allowed in **local** methods. More particularly, if the **synchronized** modifier is applied to a method, of if a method contains a **synchronized** block, then the method must be **global**.

In addition, finalize() methods must be **global**. Since finalize() methods are called on objects once they have become unreachable, the only way for a finalize() method to do anything useful is to access a **static** variable.

14.2 Repeatable Static Fields

A static field is *repeatable* if it meets all of the following criteria.

1. It must be **final** with an explicit initializer (not "blank final" with separate initialization in a **static** clause).

2. It must be of value type.

3. If its initializer were prepended with the **return** keyword and terminated with a semicolon, the result could be (without error) the sole statement in a **local static** method with no arguments.

4. It may not refer directly or indirectly (via other static variables) to the variable it initializes (this would make its value dependent on the order in which classes are initialized).

For example, x defined as "**static final** bit x = **new** bit();" is a repeatable static field because it is final, of value type, and the following would be legal.

```
static local bit foo() {
  return new bit();
}
```

However, y defined as "**static final long** y = System.nanoTime();" is not a repeatable static field because, although it is final and of value type, the following is illegal.

```
static local long foo() {
  return System.nanoTime(); // error: not local
}
```

14.3 Local/Global Polymorphism

Whether a method accesses global state is often a function of its input arguments. For instance, most equals() methods do not access global variables, and in most cases it is not good programming practice to do so. On the other hand it is not disallowed.

Glocal methods provide a mechanism for defining methods which may be local or global depending on their argument types. For instance

```
glocal boolean thatEquals(Object that) { return that.equals(this); }

global boolean fooEquals(Foo that) { return thatEquals(that); }
local boolean barEquals(bar that) { return thatEquals(that); }
```

Because thatEquals() calls the equals() method of a parameter of type Object, which can be a **global** method, thatEquals() can not be declared as a **local** method. And indeed, when called with a parameter of type Foo in the fooEquals() method, it behaves like a **global** method.

However, since bar is a value type, its equals() method *is* local. When such a parameter is passed to a **glocal** method, the called method becomes **local** in that calling context. Thus it is legal to call thatEquals() from the **local** method barEquals().

The rules for **glocal** methods are as follows:

- access to static fields is prohibited unless they are repeatable;
- calling **final global** methods is prohibited;
- calling **global** methods is prohibited except when the receiver is a parameter of the method being defined, or if its type is a generic type parameter;
- 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* as described below.

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 **local** in the static type of the actual parameter.

The overriding rule follows a simple pattern: **local** > **glocal** > **global**. A method may not override one that is "greater than" it in this relation.

14.3.1 Generics and Glocal Methods

When used in combination with generics, a **glocal** method can only be localized when all of the generic parameters (as well as all of the method parameters) are localizing. For example,

```
class Holder<T> {
  T item;
  local Holder(T item) { this.item = item; }
  local T get() { return item; }
  glocal int hashCode() { return item.hashCode(); }
}

global int fooHash(Foo that) { return new Holder<Foo>(that).hashCode(); }
local int barHash(bar that) { return new Holder<bar>(that).hashCode(); }
```

Because the generic type parameter T of class Holder is unconstrained, its hashCode() method can't be defined to be **local**. However, it *can* be defined to be **glocal**.

Since a value type causes all of the non-final Object methods to become **local**, all value types localize parameters of type Object. Thus when the value type bar is used to instantiate the Holder class, the hashCode() method is localized, and can be called from the **local** method barHash().

14.4 Multiple Method Definitions

When a class inherits multiple definitions of a method with identical type signatures but different locality qualifiers, the following cases govern the resulting locality qualification and the need to implement a new method.

1. The current type declares the method and there is more than one occurance in a supertype. The new declaration must be at least as restrictive as the most restrictive of the supertypes.

2. The current type does not declare the method and there is more than one occurance in a supertype, all of them abstract. The more restrictive one dominates, because, in any lookup or subsequent override the more restrictive one will dominate.

3. The current type does not declare the method and there is more than one occurance in a supertype, one of which is concrete, and concrete one is the most restrictive. The

concrete method will dominate in any subsequent use (both because it is concrete *and* because it is the most restrictive).

4. The current type does not declare the method and there is more than one occurance in a supertype, one of which is concrete, but the concrete one is less restrictive than some abstract one. There are two subcases.

   (a) The concrete method is **glocal** but some abstract method is **local**. In this case, the compiler will generate a simple delegator which is **local** but invokes the **glocal** concrete method passing through any arguments. The resulting delegator may be legal (if the delegating call is localizing), in which case there is no error. Otherwise, the appropriate error for a non-localizing call to a glocal method will be displayed. To correct the error, the method *must* be overridden as in case (1).

   (b) For any conflict not covered by the previous clause, an error is indicated: the method *must* be overridden as in case (1).

14.5 The Mutable Class and the Local Interface

In order to maximize reuse, it is highly desirable that classes be usable in a **local** method, and as localizing parameters. Most fundamentally, as we have just seen, is that the three overridable public methods of Object – equals(), hashCode(), and toString() – should be **local** whenever possible.

For values, the localness of these methods is provided and enforced by the compiler. For non-values, the programmer must take care to ensure this property. The easiest way to do so is to have a non-value class extend lime.lang.Mutable, rather than Object.

The Mutable class 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 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.

In general, it is good Lime programming practice to make all non-value classes extend Mutable unless there is compelling reason against it.

Both Mutable and Value implement the Local interface:

```
public universal interface Local extends Glocal {
    local boolean equals(Object obj);
    local int hashCode();
    local String toString();
}
```

And for completeness Lime also provides:

```
public universal interface Glocal {
    glocal boolean equals(Object obj);
    glocal int hashCode();
    glocal String toString();
}
```

14.6 Debugging

Because **local** methods may not access any global state, they can not, for instance, call System.out.println() for debugging purposes. Lime provides a general facility for performing debugging actions. Debugging actions are omitted in normal execution and any compiler is free to ignore them (for instance, the hardware compiler might not execute debugging operations designed to produce terminal output). In the latter case the compiler emits a warning message identifying which debugging activities will be ignored even though debugging is enabled.

The class lime.lang.Debug provides capabilities for such debugging actions. The core functionality is a pair of static methods called debugRun(), which are declared to be **local** methods but which take a special kind of runnable object and run its (non-local) method runWhenDebugOn(). This allows the implementation to pass arbitrary closures that perform potentially side-effecting operations, but because all global debugging operations pass through these two methods, they can easily be controlled and disabled.

While the Debug class is only needed inside of **local** methods, it should be used consistently across all code when performing debugging-related output.

The two runnable interfaces are:

```
public interface DebugRunnable {
    public void runWhenDebugOn(Object... args);
}

public interface DebugAccessorRunnable {
    public Object runWhenDebugOn(Object... args);
    public local Object runWhenDebugOff(Object... args);
}
```

These interfaces are supported by the following methods of Debug:

```
public static local void debugRun(DebugRunnable toRun, Object... args);
public static local Object debugRun(DebugAccessorRunnable toRun, Object... args);
```

A very simple debugging facility which just prints a list of objects, one per line, can be built as follows:

```
static void debugprint(Object... objs) {
    Debug.debugRun(new DebugRunnable() {
        public void runWhenDebugOn(Object... args) {
            for (Object o: args) { System.out.println(o); }
        }
    }, objs);
}
```

Using this paradigm, more complex facilities can be built up, including generation of performance traces, log files, and so on.

The Debug class provides convenient methods for textual output:

```
public static local void print(Object... objs);
public static local void println(Object... objs);
public static local void printf(String format, Object... args);
public static local void printf(Locale locale, String format, Object... args);
```

By default, output from these Debug methods is directed to System.out. This behavior can be changed with setDebugOutput(). The static methods out() and err() also return objects which can be used to print explicitly on System.out and System.err, respectively. However, when debugging is off, these printing operations will be ignored.

The latter functionality is supported with the second style of runnable, the DebugAccessorRunnable. Unlike the first form, it allows an arbitrary object to be returned. However, when debugging is off or not supported for the compilation target, the runWhenDebugOff() method will be invoked instead. Since this method is constrained to be **local**, no global side-effects may be performed. In the case of Debug.out() when debugging is off it returns a DebugNullPrinter object, whose implementation of all the output methods is a no-op.

14.7 Java Compatibility

The presence of **local**, **glocal** or **global** modifiers on the methods of a Lime class does not cause it to become incompatible with Java, as long as the class is otherwise Java-compatible. Such modifiers are invisible to the Java code.

The static and instance methods of Java classes are normally considered **global** and so cannot normally be used in many contexts where Lime requires them to be **local** or **glocal**.

It is possible to use pH to label any method of any final Java class as **local** or **glocal** if it can pass the necessary static checks. The class has to be **final** because otherwise subsequent Java overriding can violate the implied contract.

Java class libraries used by the Lime class libraries are assumed to have labelled all constructors of the following classes as **local**.

```
    // java.lang
Object
Byte
Character
```

```
Double
Float
Integer
Long
Short
Boolean
Void
Throwable (and all subclasses defined by the Java spec)

    // java.util
all subclasses of Throwable in the package

    // javax.sound.sampled
AudioFormat
AudioFormat.Encoding
```

Java class libraries used by the Lime class libraries are assumed to have labelled the following specific methods and constructors as **local**.

```
  // java.lang.Object methods
getClass()
  // java.lang.Class methods
getName()
getFields()
  // hashCode() methods in java.lang
String.hashCode()
Boolean.hashCode()
Long.hashCode()
Integer.hashCode()
Short.hashCode()
Byte.hashCode()
Character.hashCode()
Float.hashCode()I
Double.hashCode()I
  // toString() methods in java.lang
String.toString()
Boolean.toString()
Long.toString()
Integer.toString()
Short.toString()
Byte.toString()
Character.toString()
Float.toString()
Double.toString()
```

```
  // static toString() methods in java.lang
Boolean.toString(boolean)
Long.toString(long)
Integer.toString(int)
Short.toString(short)
Byte.toString(byte)
Character.toString(char)
Float.toString(float)
Double.toString(double)
  // java.lang.String constructors
String(String)
String(char[])
String(char[],int,int)
String(StringBuffer)
String(StringBuilder)
String(int[],int,int)
String(byte[])
String(byte[],int,int)
  // java.lang.String methods
length()
charAt(int)
concat(String)
compareTo(String)
substring(int,int)
subSequence(int,int)
getChars(int,int,char[],int)
equalsIgnoreCase(String)
regionMatches(int, String, int, int)
regionMatches(boolean,int,String,int,int)
startsWith(String,int)
startsWith(String)
endsWith(String)
indexOf(int)
indexOf(int,int)
lastIndexOf(int)
lastIndexOf(int,int)
indexOf(String)
indexOf(String,int)
lastIndexOf(String)
lastIndexOf(String,int)
toCharArray()
substring(int)
```

```
replace(char,char)
toLowerCase()
toUpperCase()
trim()
split(String,int)
copyValueOf(char[])
copyValueOf(char[],int,int)
valueOf(boolean)
valueOf(byte)
valueOf(char)
valueOf(float)
valueOf(double)
valueOf(int)
valueOf(long)
valueOf(short)
valueOf(char[])
valueOf(char[],int,int)
  // java.lang.StringBuilder constructors
StringBuilder()
StringBuilder(int)
StringBuilder(String)
  // java.lang.StringBuilder methods
toString()
append(String)
append(StringBuffer)
append(boolean)
append(byte)
append(char)
append(float)
append(double)
append(int)
append(long)
append(short)
append(char[])
append(char[],int,int)
subSequence(int,int)
charAt(int)
length()
  // Number methods in each of the final subclasses
  // of java.lang.Number
floatValue()
doubleValue()
```

```
byteValue()
shortValue()
intValue()
longValue()
  // Misc methods of specific Number classes
Double.doubleToLongBits(double)
Float.floatToIntBits(float)
Integer.toHexString(int)
Integer.rotateLeft(int,int)
Integer.rotateRight(int,int)
Long.toHexString(int)
  // java.lang.Character
toLowerCase(char)
toTitleCase(char)
toUpperCase(char)
isDefined(char)
isLetter(char)
isDigit(char)
isLetterOrDigit(char)
isLowerCase(char)
isUpperCase(char)
isTitleCase(char)
isJavaIdentifierStart(char)
isJavaIdentifierPart(char)
isUnicodeIdentifierStart(char)
isUnicodeIdentifierPart(char)
isIdentifierIgnorable(char)
isDefined(int)
isLetter(int)
isDigit(int)
isLetterOrDigit(int)
isLowerCase(int)
isUpperCase(int)
isTitleCase(int)
isJavaIdentifierStart(int)
isJavaIdentifierPart(int)
isUnicodeIdentifierStart(int)
isUnicodeIdentifierPart(int)
isIdentifierIgnorable(int)
  // java.lang.System
arraycopy(Object,int,Object,int,int)
identityHashCode() // but can't be invoked on values
```

```
  // java.lang.Math
abs(int)
max(int,int)
min(int,int)
abs(long)
max(long,long)
min(long,long)
abs(double)
acos(double)
asin(double)
atan(double)
cbrt(double)
ceil(double)
cos(double)
cosh(double)
exp(double)
expm1(double)
floor(double)
log(double)
log10(double)
log1p(double)
max(double,double)
min(double,double)
nextUp(double)
rint(double)
signum(double)
sin(double)
sinh(double)
sqrt(double)
tan(double)
tanh(double)
toDegrees(double)
toRadians(double)
ulp(double)
atan2(double,double)
copySign(double,double)
hypot(double,double)
IEEEremainder(double,double)
nextAfter(double,double)
pow(double,double)
scalb(double,int)
round(double)
```

```
getExponent(double)
abs(float)
max(float,float)
min(float,float)
nextUp(float)
signum(float)
ulp(float)
copySign(float,float)
nextAfter(float,double)
scalb(float,int)
round(float)
getExponent(float)
  // java.lang.reflect
Array.newInstance(Class,int)
  // java.util
Arrays.hashCode(long[])
Arrays.toString(long[])
Arrays.equals(long[],long[])
Arrays.binarySearch(long[],long)
Arrays.binarySearch(long[],int,int,long)
Arrays.fill(long[],long)
Arrays.sort(long[])
Arrays.copyOf(long[],int)
Arrays.copyOfRange(long[],int,int)
... and similarly for other primitive arrays
```

Java class libraries used by the Lime class libraries are assumed to have labelled the following specific methods and constructors as **glocal**.

```
StringBuilder(CharSequence)
StringBuilder.append(CharSequence)
StringBuilder.append(CharSequence,int,int)
StringBuilder.append(Object)
String.valueOf(Object)
```

.

14.8 New Grammar

In Lime, **local**, **global** and **glocal** are lexically reserved keywords and are considered to be method modifiers as defined in section 8.4.3 of the Java Language Specification and also abstract method modifiers are defined in section 9.4.

```
MethodModifier ::= 'local'
```

```
MethodModifier ::= 'global'
MethodModifier ::= 'glocal'
AbstractMethodModifier ::= 'local'
AbstractMethodModifier ::= 'global'
AbstractMethodModifier ::= 'glocal'
```

Chapter 15

Stream Computation

Lime offers language features for programming in the large. They are based on the creation of data flow 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 stream graph, consists of nodes that perform computation and edges that imply an exchange of data between connected nodes. In Lime, nodes are *tasks* (see 15.3) 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 can be applied repeatedly as long as there are input data available on the port.

A simple task is illustrated in Figure 15.1. It is known as a *Filter* (see 15.4). Filters may be stateless or stateful. Stateless filters do not maintain persistent state between applications of the worker method, whereas stateful filters do maintain history. Tasks are created using the task operator applied to a method.

Tasks may be connected to each other (see 15.6) so that the stream of one task is connected to a port of another. The connections between tasks can be thought of as FIFO buffers with one task writing at one end and another task reading at the other end. A sequence of connected tasks is known as compound filter or *pipeline*. An example is illustrated in Figure 15.2: the output stream of `worker1` is the input to `worker2`, and the output of `worker2` is the input to `worker3`.

Methods in Lime are mode-less. In other words, a method declaration is agnostic to its eventual use: it may be used as a worker method in a task, or as a method invoked from an instance object. Conceptually, one can think of the worker methods in tasks as wrapper methods that are 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 then writes the returned results to the output stream. For a pair of connected tasks, the return type of one method should match the input type of the other. If the types do not quite match, but are compatible, Lime provides a matching operator to disaggregate and reaggregate data (see 16). In example shown in Figure 15.3, the return type of `work1` is a bounded array of `int`s, whereas the `work2` method expects a single scalar parameter of type `int`. The types are considered compatible since they match with respect to the base type

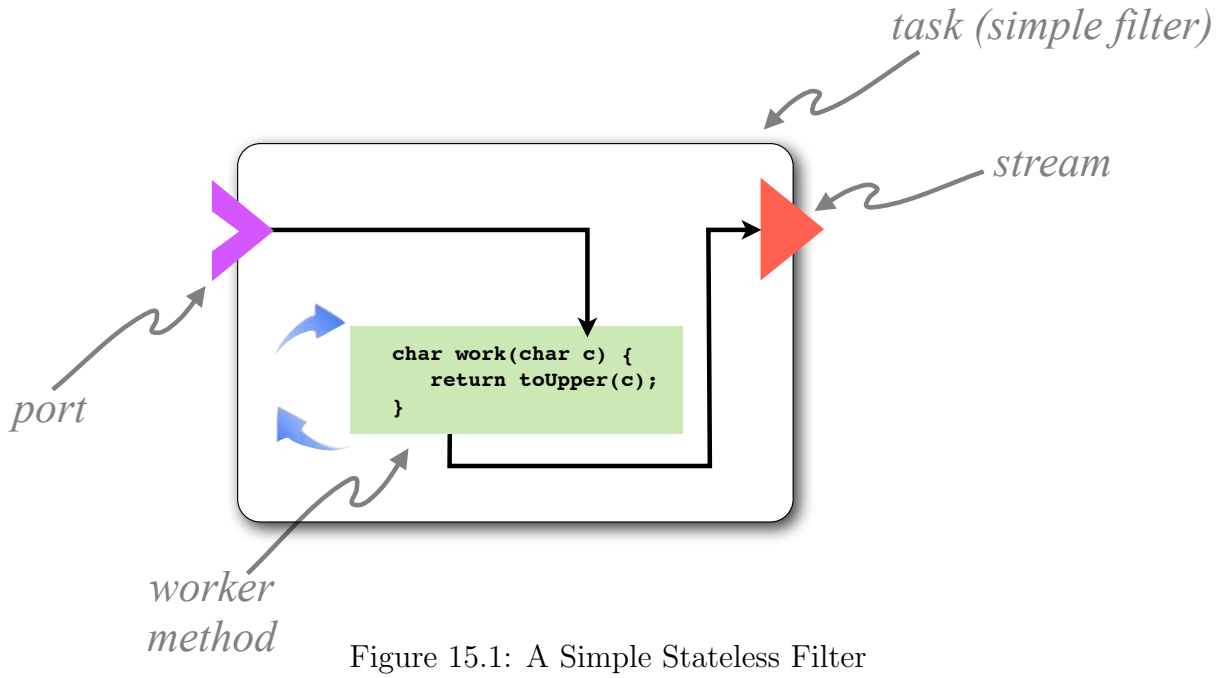

Figure 15.1: A Simple Stateless Filter

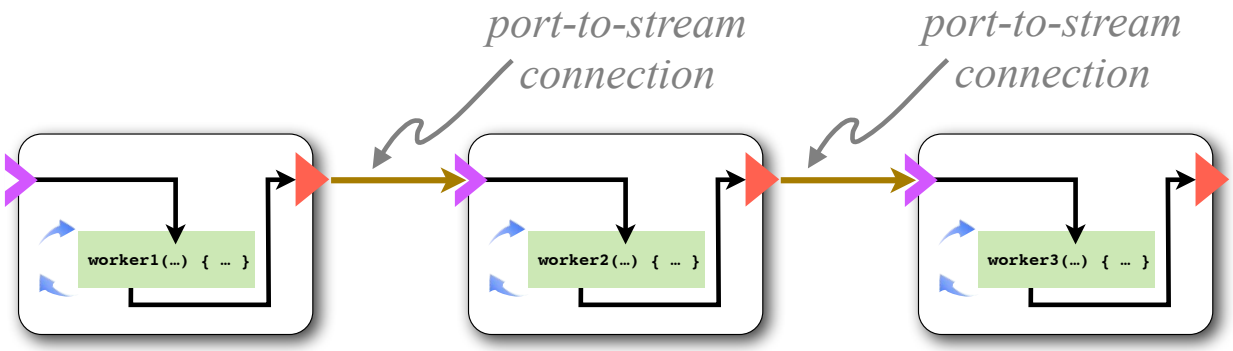

Figure 15.2: A Compound Filter Composed from Three Filters

(`int`) and hence the *rate matcher* can disaggregate the bounded arrays returned by the first task and produce scalar values that are consumed by the second.

In addition to filters, there are *splitter* and *joiner* tasks (see 15.10). A splitter conceptually distributes its input to multiple tasks, and a joiner aggregates output streams from multiple tasks into a single new stream. The example in Figure 15.4 shows a pipeline consisting of a splitter, two filters, and a joiner. Lime's language feature for stream computing expose pipeline parallelism using compound filters, data parallelism using stateless filters, and task parallelism using splitters and joiners.

Filters, and tasks in general are strictly isolated so that only a task can mutate its own fields. In other words, a task may not contain references to globally visible objects that are mutable. This is an important property that affords the compiler a lot of flexibility in navigating the implementation-space for stream graphs. However, the strict isolation is relaxed for two types of filters known as *sources* and *sinks* (see 15.7). As shown in Figure 15.5,

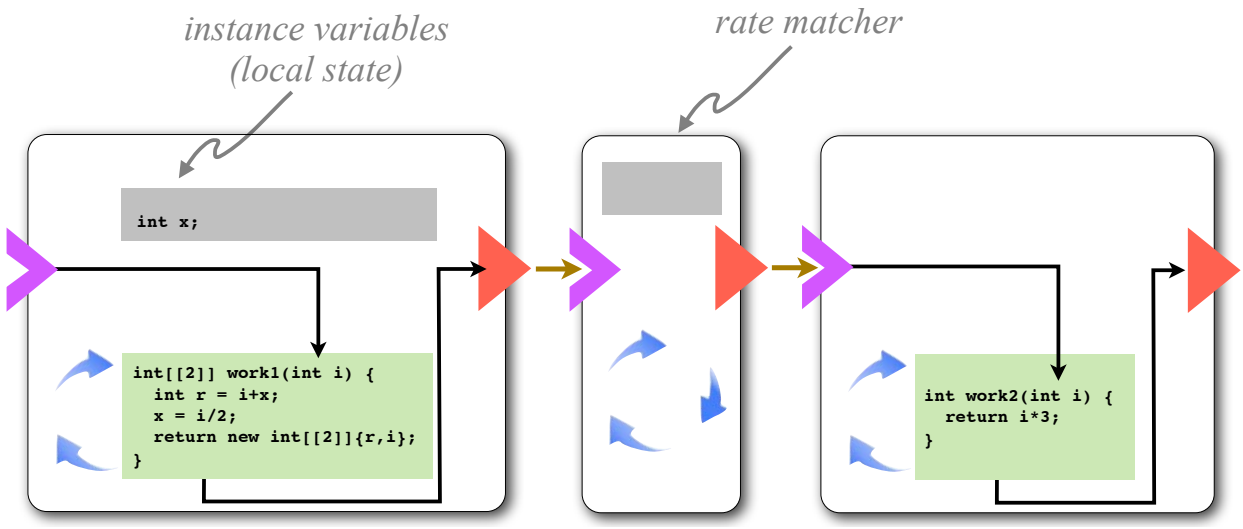

Figure 15.3: A Stateful Filter connected to a Rate Matcher that converts a stream of int[[2]] into a stream of int.

sources and sinks may perform side-effecting actions that other tasks are not permitted to perform. They may interact with the program heap, file system, and in general perform input and output.

15.1 Stream Types

A stream is a possibly infinite sequence of values (that is, instances of the value types, including primitive types, ordinal types, **value enum** types, value arrays, value classes, and tuples made up entirely of values). The values of a stream can be retrieved in order. Usually, once a value is retrieved from the stream it is no longer part of the stream. A "peek" operation is provided, but efficient use of streams requires that access is largely sequential.

The simplest looking form of stream is a stream literal:

```
Stream<int> is = { 1, 1, 2, 3, 5, 8 };
```

In fact, a stream literal is actually a special case (with some syntactic sugar) of the source() method of Lime arrays. The same affect can be achieved, more verbosely but ultimately more generally, with the following.

```
Stream<int> is = (new int[[]]{ 1, 1, 2, 3, 5, 8}).source().out();
```

Stream types are *not* value types. Rather, they are reference types whose behavior is defined by the lime.lang.Stream interface.

```
public interface Stream<T extends Value> extends Iteratable<T> {
  public T get();
  public T[[]] get(int n);
  public T[[]] getAll();
```

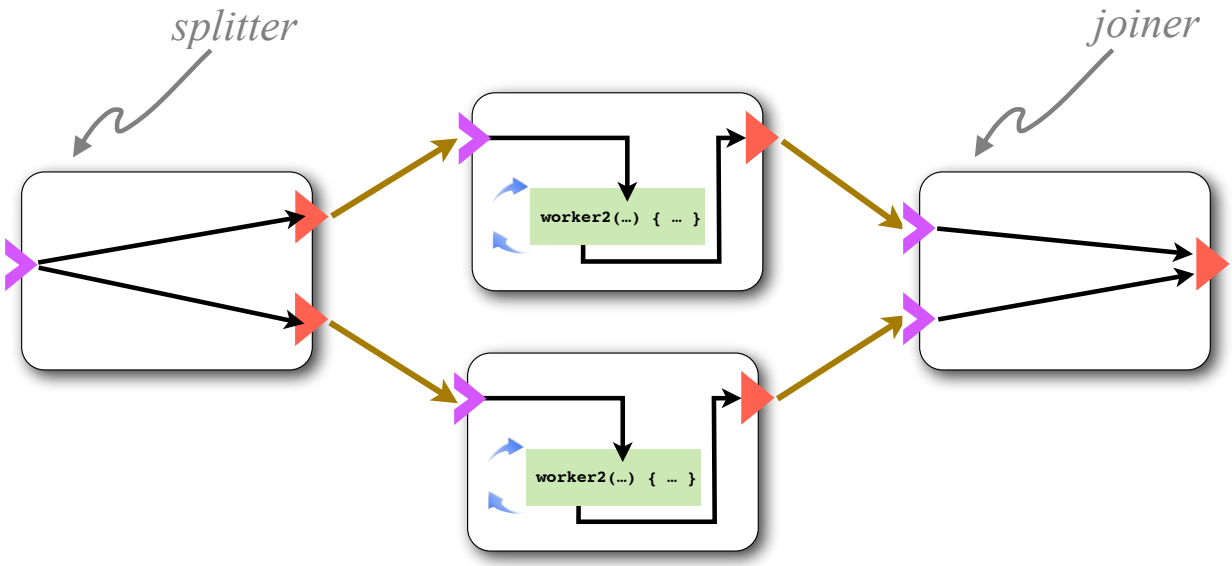

Figure 15.4: A Compound Filter containing a Split and a Join

```
  public T peek();
  public T[[]] peek(int n);
  public T[[]] peek(int skip, int n);
  public boolean empty();
  public Iterator<T> iterator();
}
```

While it is possible for a user-written class to implement this interface, only instances produced by the operations of this section will have the behavior ascribed by this section to streams.

Streams include operations to get the next element and find out if there are more elements:

```
while (! is.empty()) {
  System.out.println(is.get() + " ");
}
```

will print out "1 1 2 3 5 8". An attempt to get an element from an empty stream will result in a Stream−Underflow being thrown. Thus the code above could be written as

```
try {
  while (true) {
    System.out.println(is.get() + " ");
  }
} catch (StreamUnderflow e) {
}
```

As will be seen, streams that arise in practice are often logically infinite, in which case empty() always returns false. Thus, the second form is often preferable to the first.

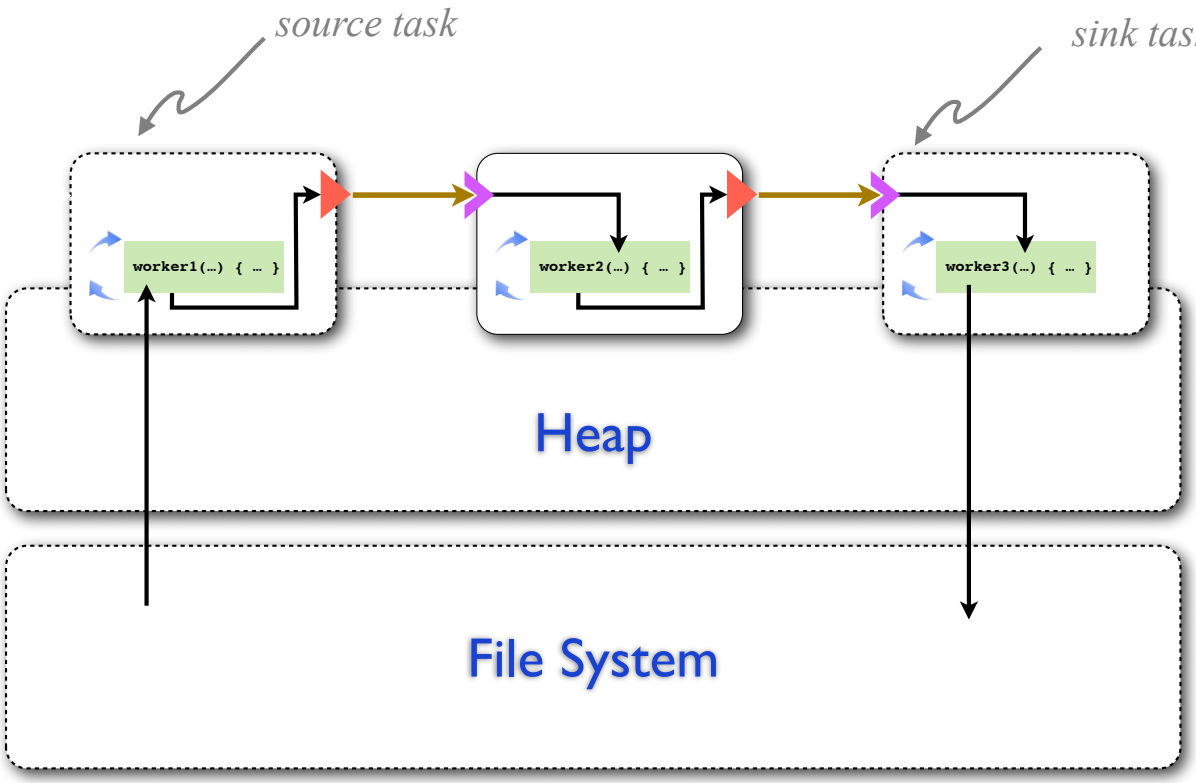

Figure 15.5: A Complete Stream Graph with a Source, Filter, and Sink. Sources and Sinks need not be isolated and may therefore access the heap and other forms of global state.

15.1.1 Inspection and Iteration

The get(n) method when supplied with a positive integer returns an array containing that many elements, or throws a StreamUnderflow. The getAll() operation on a stream gets all of the elements of a stream and returns them as an array. If the stream is unbounded, this may involve a large delay or an OutOfMemoryException.

The peek operation is similar but leaves the values in the stream. peek() returns the value at the head of the stream, and peek(n) returns an array containing the n values at the head of the stream. The operation peek(s,n) returns an n-element array consisting of the n values in the stream that follow the first s values in the stream. In all cases, StreamUnderflow is thrown if there is insufficient data in the stream.

Both get and peek are only applicable to streams that have not been connected to other tasks (see below). An attempt to use them on a connected stream will cause an AlreadyConnectedException to be thrown.

Note that Stream implements Iteratable (see section 8); hence iteration over streams may be performed with a **for** loop, as in

```
for (int i: is) {
  System.out.println("Value: "+i);
}
```

As was the case with the empty() method, many streams will return iterators whose hasNext() method always returns **true** and so loops such as the above may terminate by throwing StreamUnderflow.

15.2 Ports

Streams are used to retrieve data; ports are used to supply data.

Port types are declared using the Port interface.

```
Port<int> p;
```

The definition of Port is:

```
public interface Port<T extends Value> {
  public void put(T val);
  public void put(T[[]] vals);
  public void close();
}
```

Just as streams provide a get() operation, ports provide a put() operation, which sends a value to the port:

```
p.put(7);
```

There is an array generalization of put but there is no way to write values out of order or to change a value that is already written. The close operation is used to terminate a task by producing a StreamUnderflow indication on its port; there is more on this in section 15.8.

The put() and close() operations are only legal if the port has not yet been connected to a stream; otherwise, an AlreadyConnectedException is thrown.

If repeated put() operations are performed without consuming values at the other end of the task graph, the number of values stored in the queue associated with the port may exceed its capacity. In this case a StreamOverflowError is thrown by the put() operation.

In the following section we will see how ports are created and associated with tasks.

15.3 Tasks

In general, streams of data are produced by *tasks*. Task outputs are called *streams*, and task inputs are called *ports*. Tasks can be classified along two dimensions, as follows.

Tasks are either *closed*, *simple* or *compound*. A closed task has no ports or streams. A simple task has at most one port and at most one stream but is not closed. A compound task has more than one port or more than one stream or both.

Tasks are either *user*, *system*, or *aggregate* tasks. The first two of these are called *primitive* tasks because they were not constructed from smaller tasks. A user task runs user-written code. A system task runs system-provided code. An aggregate task is constructed by assembling other tasks.

A user task is always a simple task. System and aggregate tasks may be simple or compound and aggregate tasks can be closed.

Tasks with exactly one port and exactly one stream are called *filters*. User filters are constrained to produce one physical output value for each physical input value. Other rates are expressed by using aggregate types (consuming an array or tuple of values and producing an array or tuple of values). System filters called *matchers* (described in section 16) can be interposed between user filters to aggregate, disaggregate, or reaggregate these arrays or tuples.

Filters (in fact, most but not all tasks) have the property that they are *isolated* from the rest of the system in that they only observe data via their ports and only generate data via their streams. These data are constrained to be values. 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 15.7).

15.3.1 Isolation

User filters can be created from constants, static methods, instance methods, or operators. However, they can only be created from entities that satisfy one of two *isolation* properties.

1. Some entities are *inherently isolated*, meaning that it does not matter what references to the entity may exist.

2. Others are *sole-reference isolated*, meaning that an appropriately isolated task can be created from them by (and only by) ensuring that the task holds the sole reference to the object.

Inherent isolation is defined as follows.

- An instance of a value (value class, value array, ordinal, value enum, or primitive type) is inherently isolated when used as a constant.

- An instance method of a value or a static method is inherently isolated if it is **local** and has only value arguments and a value (or void) return.

- A user-defined operator of a value is inherently isolated if it meets the same criteria as the previous. The language-defined operators on the primitive types meet these criteria *a priori*.

Sole-reference isolation is defined for pairs consisting of a constructor and an instance method from the same class or one of its superclasses. The property holds for the instance method iff the constructor is used to create the instance and the task holds the sole reference to that instance.

- A constructor is *isolating* if it is **local** and has only value arguments.

- An instance method is sole-reference isolated with respect to an isolating constructor of the same class or a subclass if it is **local**, and it has only value arguments and a value (or void) return.

Sole-reference isolation is used to create stateful (history-sensitive) tasks, which is not possible using values or static methods. The instance method selected to define the task may mutate the instance to keep state. Sole-reference isolation works even if the class also defines **global** instance methods, because the local method will (by definition) not call these and isolation assures that no other code will either. Sole-reference isolation works even if the class has mutable **public** fields because, even if the chosen instance method accesses these, no other code will.

Both kinds of isolation are checked statically by the compiler. This can be done without interprocedural analysis because the definitions depend only on staticness, valueness, and localness, which are present in declarations.

Task creation operations correspond to these different kinds of isolated entities, and are described below.

15.3.2 Task Types

Tasks are created with the **task** operator, which returns a "handle" for the task. This handle is always a subclass of the type lime.lang.Task. The Task interface is not generic and has only methods related to control, which are discussed later.

```
public interface Task {
  public void start();
  public Throwable rendezvous();
  public boolean isRunning();
}
```

While the Task interface may be implemented by user code, the result is not a task as defined in this section; true tasks are *always* created by the system.

User tasks are always one of the following subclasses. Only the Filter, Source, and Sink types can be created directly; the SimpleTask and TerminalTask interfaces further refine what operations the resulting tasks can perform.

```
public interface Filter<Tin extends Value, Tout extends Value> extends SimpleTask {
  public Port<Tin> in();
  public Stream<Tout> out();
}
public interface Source<Tout extends Value> extends TerminalTask {
  public Stream<Tout> out();
}
public interface Sink<Tin extends Value> extends TerminalTask {
  public Port<Tin> in();
}
```

```
public interface TerminalTask extends SimpleTask {
  public boolean isIsolated();
  public Object getWorker();
  // Explained later
}
public interface SimpleTask extends Task {
  // Covers all Tasks that have at most one port and at most one stream
}
```

In other words, user tasks are limited to filters (single input, single output), sources (single output) and sinks (single input).

The entities (methods, constants or operators) used to create filters must be isolated. As will be seen, sources and sinks (which extend TerminalTask) are exempt from this rule.

15.3.3 Logical Rates

A Filter has a *rate* property which is the ratio between the number of input values and the number of output values. The rate for user filters is always 1:1. The use of rate matching (see Section 16) may create system filters with non-unit rates.

The input and output counts that comprise a ratio are actually ranges. The lower bound of either range can be zero. The upper bound of either range can be infinite. A rate in which both input and output can be anything from zero to infinity basically means that nothing is known about the rate. The most useful rates are those in which both input and output are narrowed to a single integer (e.g. 1:5).

15.4 Filter Creation

We now describe the various means for creating user filters. The creation of user sources and sinks is covered later.

For present purposes we will use these example type definitions:

```
value class foo {
  static final double p = Math.random();
  static double addP(int a) { return p+a; }

  int x;
  foo(int n) { this.x = n; }
  int add(int a) { return x+a; }
  int this + (int a) { return add(a); }
  int getX() { return x; }
}
```

15.4.1 Filters from Static Methods

Isolated static methods with at least one parameter and a non-void return type can be used to create filters as follows:

```
Filter<int, double> addrandom = task foo.addP(int);
```

More generally, the **task** keyword is followed by name of a static method, qualified by its signature to make the reference unambiguous. The type of this kind of **task** expression is a Filter whose port type is derived from the method's parameter list (which must not be empty if a filter is intended) and whose stream type is the return type of the method (which must not be **void** when a filter is intended). Methods with empty parameter lists or void returns can be used to create sources and sinks, as described later.

The port type of a Filter is a tuple type if the number of arguments is greater than one. Otherwise, the port type of a Filter is the type of the method's single argument.

15.4.2 Filters from Value Instance Methods and Operators

Any inherently isolated instance method of a value class can be used to create a task. This includes user-defined operators of user-defined value classes, and extends to the primitive types, treated as values, for which the operators may also considered to be instance methods. These cases are shown in the following examples:

```
foo f = new foo(7);
Filter<int, int> add7 = task f.add(int);
Filter<int, int> add7a = task {f + int};
Filter<int, int> add7b = task {7 + int};
```

The general rules are the same as for a **static** method except that the method or operator is associated with specific value instance.

The set of operators that can be used in this way are exactly the set of normally immutable operators that may be user defined (all the ones discussed in section 5 except indexed assignment).

Note that curly braces are used when making a task from an operator. This syntax is needed to disambiguate operator references from simple expressions (tasks produced from expressions are discussed in section 15.7). The operator must be one that is legal for the value, either because the value is of primitive type, or because it it is a user-defined value type that implements the operator.

15.4.3 Filters from Non-value Instance Methods

So far, all of the tasks we have introduced have no mutable state. Suppose we wish to have a stream operator that produces the average of its inputs. As previously discussed, this requires the use of a non-value class, an isolating constructor, and an instance method that is sole-reference isolated with respect to that constructor. We could define the following class:

```
class Averager {
  private double total;
  private long count;

  public local Averager() {}

  public local double runningAverage(double x) {
    total += x;
    count++;
    return total / count;
  }
}
```

The constructor was explicitly declared **local** since methods and constructors of non-value classes are **global** by default.

Then this class could be used to create an averaging task as follows:

```
Filter<double,double> avg = task Averager().runningAverage(double);
```

The **task** operator in this case performs the equivalent of a **new** operation on the class using the isolating constructor, and then instantiates a stream task that executes the runningAverage method for each value supplied on its input port. The task holds the sole reference to the object.

To understand the general principle being illustrated, it is useful to rewrite the earlier value instance method example so that it uses a single line of code.

```
Filter<int, int> add7 = task new foo(7).add(int);
```

That works because the add methods of all instances of foo (a value) are intrinsically isolated. If we used the same expression with **new** Averager().runningAverage(**double**), however, the compiler would complain that this instance method is not intrinsically isolated (the definitions deliberately do *not* assume that the compiler applies escape analysis to answer isolation questions, which would probably work in this particular case, but would lead to fragile and surprising results in general). The task creation expression, when only sole-reference isolation exists, must name both the isolating constructor, its arguments, if any, and the instance method to be used, all in a single expression. Thus, **task** Averager(). runningAverage(**double**) must be thought of as a single grammatic production that cannot be broken up. To avoid a number of syntactic ambiguities, stateful filters cannot be constructed from user-defined operators but require a named instance method.

For generality, we accept the form of **task** expression that is required by sole-reference isolated methods even for intrinsically isolated methods. That is, the compiler would have accepted **task** foo(7).add(**int**).

15.4.4 Abbreviation of Worker Methods

The method or operator used in a task creation operation is called the *worker method* of the task.

When the method is unambiguous (not overloaded), the signature may be omitted in the task creation expression, as in

```
Filter<int, double> addrandom = task foo.addP;
Filter<double,double> avg = task Averager().runningAverage;
```

15.5 Direct Use of Filters

When a task is first created, its port and stream are not connected to anything. Therefore, they may be accessed directly with the put() and get() operators. For instance,

```
avg.in.put(2);
avg.in.put(6);
double a1 = avg.out.get(); // a1 == 2
double a2 = avg.out.get(); // a2 == 4
```

This capability should be used sparingly as correct use requires knowing the filter's rate. Connecting filters into graphs provides a less error prone way of accomplishing the scheduling.

15.6 Connecting Ports and Streams

The *connect* operator "=>" connects a stream to a port. So, using the example tasks created above:

```
add7.out() => avg.in();
```

Note that add7.out() is of type Stream<**int**> while avg.in() is of type Port<**double**>. The connection is legal because there is a widening conversion from **int** to **double**. A connection requires that the receiving port have the same or wider type than the providing stream.

If either the port or the stream are already connected, an AlreadyConnectedException is thrown.

It should be noted that any not-yet-connected stream can be connected to any not-yet-connected port but the result is not necessarily legal and not necessarily schedulable. A graph with cycles is illegal. However, cycles are not detected until the graph is started (see section 15.8). Even a legal graph may not be statically schedulable due to unbalanced rates; such a graph is dynamically scheduled, which may be inefficient and may embody deadlock or unbounded queue growth. Better schedulability guarantees are obtained by following the practices recommended in section 15.11.

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 techniques of messaging, as described in section 17.

Once a port is connected, the put() operation is no longer allowed; if it is invoked, an AlreadyConnectedException is thrown. Similarly, if a stream is connected, an attempt to use get or peek operations will cause a AlreadyConnectedException to be thrown.

The connect operator can also be used on filters, in which case it returns a new filter whose input type is that of the leftmost filter and whose output type is that of the rightmost filter:

```
Filter<int, double> threestages = add7 => addrandom => avg;
```

Note that the types of the three filters are Filter<**int**,**int**>, Filter<**int**,**double**>, and Filter <**double**,**double**>, making the composition legal.

Similarly, connecting streams to filters or filters to ports is allowed, and the result is the end of the filter that was not just connected:

```
Filter<int, double> f = ...;
Stream<int> s1 = ...;
Stream<double> fstream = s1 => f;

Filter<double,string> g = ...;
Port<string> p1 = ...;
Port<string> gport = g => p1;
```

15.7 Sources and Sinks

15.7.1 Sources

The simplest form of task is created from a simple value, and produces a task which produces an infinite stream which repeats that value. The **task** operator followed by the **value** keyword and an expression that evaluates to a value creates a task, and returns an object of parameterized type Source:

```
Source<int> threes = task value 3;
Source<bit> zeroes = task value bit.zero;
Source<uint> rand = task value (uint) Bar.p;
```

A source is also created if a parameterless method is used in the task creation. For instance:

```
foo f = new foo(9);
Source<int> nines = task f.getX(void);

class Countup {
  local Countup() {}
  int x;
  local int next() { return x++; }
}
```

```
Source<int> intsequence = task Countup().next(void);
```

The **void** type is used to remove any ambiguity about whether you are referring to the method or invoking the method. In the absence of method overloading, both the parentheses and the **void** may be omitted. That is, assuming there were no other getX or next methods, then **task** f.getX and **task** Countup().next could have been used.

If you recall that binary operators (system-defined and user-defined) on values could be used to create filters, you might wonder if unary operators can be used to create sources. The answer is no. Since values aren't stateful, something like **task** { +++0 } would be constrained always to return the same answer and is not usefully better than **task value** 1.

The result of connecting a Source to a Filter with the => operator is a Source whose out type is that of the Filter.

The source() and infinite() methods of Lime's array types will produce sources that either provide the elements of the array once, followed by a stream underflow (source), or provide the elements repeatedly, producing an infinite stream (infinite).

The stream literal syntax shown at the start of this chapter is just syntactic sugar over the source() method. But, source() and infinite() are general ways of turning arrays into sources.

Unlike filters, sources and sinks are not required to be isolated. Any method with the appropriate signature can be chosen. The isolation properties are checked by the compiler just as for a filter. If the source or sink is, in fact, isolated, the compiler may make use of that fact in generating more optimal code (therefore, it is good practice to accurately label source and sink methods as local or global).

In addition, non-isolated sources and sinks are guaranteed to run on a Thread unique to each such non-isolated source or sink. This thread identity is used, for instance, when a source or sink performs synchronization operations, and is returned if the Thread.currentThread() method is called.

In contrast, threads used to run isolated tasks are controlled by the system, which does not guarantee a unique thread for each task. Methods of Thread that could reveal the underlying thread or allow it to be manipulated are **global** and hence not accessible to isolated tasks.

15.7.2 Sinks

A sink is created when a task is instantiated with a **void** function or method. For instance

```
class DoSomething { native void act(int x); }
Sink<int> sink = task DoSomething.act(int);
```

The example shows a non-isolated sink (the native method act is global, since it is in a non-value class and not declared local).

A second possible usage pattern is enabled by using an isolated sink in conjunction with the getWorker method which is described more thoroughly in section 15.8. One possible example is as follows.

```
public class MySink<V extends value> {
  public local MySink() {
    // make some collection
  }
  public local void consume(V item) {
    // append to the collection
  }
  public V[] getArray() {
   // return collection contents as an array
  }
}
```

A Sink task created using the example class behaves like a Sink as long as the task is running (which will as long as the stream feeding it data has not underflowed).

```
Sink<int> sink = task MySink<int>().consume;
someStream => sink;
sink.start();
sink.rendezvous();
int[] result = ((MySink<int>) sink.getWorker()).getArray();
```

The getWorker function returns the Object whose instance method is the worker method of the task. It is only permitted on TerminalTasks. If such tasks are isolated and the task is running, getWorker will throw IllegalStateException (it is impossible to retrieve the worker of a running isolated task). If the worker method is static, getWorker returns **null**.

The result of connecting a Filter to a Sink with the => operator is a Sink whose in type is that of the Filter. The result of connecting a Source to a Sink with the => operator is an expression of type lime.lang.ClosedGraph.

```
public interface ClosedGraph extends Task {
}
```

A ClosedGraph has all the control operations of Task but no exposed port or stream on which either get or put operations can be performed.

15.8 Task States

When a task is created, it is in the *initial* state. It does not pull data from its ports nor push data to its streams. Simply connecting the ports or streams of a task to other tasks does *not* immediately cause it to leave the initial state unless those connections are to already-running tasks.

A task enters the *running* state when any of the following things happen.

1. A get or peek or getAll method is called on one of its unconnected streams.
2. A put method is called on one of its unconnected ports.

3. A task to which one of its streams or ports is connected moves to the running state or the task is connected to another task that is already running.

4. The start method of the task is executed.

It follows that calling the start method on any task in a fully connected graph of tasks is sufficient to start a chain reaction resulting in all the tasks of the graph running. Connecting an unstarted graph segment to a started one will cause the started state to propagate into the formerly unstarted region. A graph offering an unconnected stream can also be activated by calling get or getAll and one with an unconnected port can be activated by calling put. A graph that is moved to the running state because of put or get activity stays in that state even though it is currently blocked waiting for more puts or gets to occur.

Note that the sole reference to the worker object in the case of sole-reference isolation is created eagerly at the time the **task** expression is evaluated. Such creation is not delayed until start time.

The start operation can throw a CyclicGraphException if the graph is found to be cyclic at the time it is started. If a graph is already started at the point when a new connection causes it to become cyclic, the exception may be thrown at that time. The start operation may throw an UnschedulableGraphException if areas of the graph have provably divergent static rates such that deadlock or resource exhaustion is inevitable. However, failure to throw that exception does *not* mean that deadlock or resource exhaustion won't happen. Variable rates within the graph cause dynamic scheduling to be used such that, in general, it is impossible to determine whether the result will be viable. The scheduling proceeds optimistically in that case.

Starting an already-started graph is silently accepted without producing any exception. Connecting two graph segments that were previously started is allowed (as long as no cycle is created). This action can cause both graphs to temporarily pause while a new schedule is calculated and then the resulting fused graph is restarted. Exceptions associated with start can therefore by thrown at such a time.

The isRunning method of a task returns **true** if and only if the task is in the running state.

A task enters the *terminated* state when an uncaught exception of any kind occurs during the execution of the task's worker method. By convention, sources are expected to indicate "normal" termination by throwing lime.lang.Completion or one of its subclasses. Completion is a subclass of RuntimeException which has the specific meaning of "normal termination" in Lime. StreamUnderflow is a subclass of Completion which should be used when the proximate cause is activity on streams; Completion or some more detailed subclass should be used when the worker is functionally finished.

A task also enters the terminated state when it has at least one unconnected port and the close() method of some port is called.

If a single task in a graph terminates for one of the preceding reasons, the termination propagates in either an orderly or less orderly fashion depending on the reason for termination.

If the task terminates due an explicit close() on one of its ports or due to throwing Completion or a subclass thereof, then termination is "orderly" as follows.

- Tasks whose ports are fed by the streams of the terminated task continue to execute until all values produced by the terminated task have been consumed. Those tasks are then considered to have undergone orderly termination and propagation downstream continues.
- Tasks whose streams feed the ports of the terminated task are stopped as soon as feasible. This is not precisely determined since they may be executing asynchronously. As soon as a task is stopped as part of this rule, it is considered to have undergone orderly termination and propagation upstream continues.
- a get operation on an unconnected stream of the terminated task will return normally as long as there is data in the stream buffer. On the first execution in which there is no more data, get throws StreamUnderflow.
- a getAll operation on an unconnected stream of the terminated task will return (normally) all of the data in the stream buffer, it being clear that no more is forthcoming.

It follows that a single Completion indication on a connected graph of tasks will cause all the tasks to terminate eventually with queues drained to the extent possible. If the graph has only one source and that source is the first to terminate, then the graph will terminate precisely at the point when it would have stalled due to lack of data from the terminated source. Otherwise, termination of the non-underflowed portions of the graph will be as soon as feasible.

If the cause of termination is 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 some data on queues. Any outstanding get or getAll operation will return with just the data available to it.

If more than one task moves from the running to the terminated state independently, before the propagation of termination from one reaches the other, or if an exception happens while task is "draining" its port queues, the result is non-deterministic and generally follows the weaker guarantees of "abrupt" termination.

The rendezvous method will block the caller until the task is in the terminated state. The return value of rendezvous is always the Throwable that caused termination, whether normal or abrupt.

When a task is in the terminated state (only), the object providing its worker method implementation may be retrieved (even if sole-reference isolation was used, it being now safe to break the isolation since the task has terminated). The getWorker method, available only on TerminalTasks will do one of these three things.

1. If the task is isolated and is not in the terminated state, an IllegalStateException is thrown.

2. If the task is in the terminated state or is not isolated but represents the composition of more than one primitive task or was created from a static method (ie it does not have a single object containing its implementation) then **null** is returned.

3. If the task is in the terminated state and represents a primitive task created from an entity other than a static method, then that entity is returned. As the entity might be a value, such an entity will be returned in boxed form.

15.9 Constant Task Parameters

It is possible to supply constant values for some of the parameters of the worker method of a task. In this case, the missing parameters are specified by their type, and the supplied parameters are specified as expressions. The resulting input type of the Filter is then the concatenation of the missing parameters. For instance, if the signature of the worker method of a class Worker is

```
local int work(int a, double b, bit c);
```

then the following filters can be created:

```
Filter<`(int, double, bit), int> f0 = task Worker().work;
Filter<int,int> f1 = task Worker().work(int, 1.0, zero);
Filter<`(int,bit), int> f2 = task Worker().work(int, 1.0, bit);
```

15.10 Splitting and Joining

15.10.1 Joining Streams

The **join** operator takes a tuple or bounded array of streams and produces a single stream of tuples or bounded value arrays consisting of the element-wise merge of the input streams:

```
Stream<int> s1;
Stream<double> s2;
Stream<`(int, double)> s3 = join `(s1, s2);
Stream<int>[2] t;
Stream<int[[2]]> tt = join t;
```

When multiple streams are joined, if one stream ends before the others, Stream−Underflow is thrown.

15.10.2 Splitting a Stream

A stream of tuples or bounded value arrays can be split into multiple streams containing the tuple elements or array elements by using the **split** operator.

```
Stream<`(int, double, string)> s;
`(Stream<int> is, Stream<double> ds, Stream<string> ss) = split s;
Stream<int[[2]]> tt;
Stream<int>[2] t2 = split tt;
```

The first assignment splits the stream into three streams. Note that when a stream is split, if any one of the split streams is not consumed, then the other split streams may block.

Note also that when bounded arrays are employed these are necessarily bounded value arrays when the array is the element type of a stream (these must be values) and bounded non-value arrays when the element type *is* a stream (streams are not values and hence cannot be the element type of a value array).

15.10.3 Splitters and Joiners

The **split** and **join** keywords apply only to streams, which means that using *only* these operators to build a pipeline that has fan-out and fan-in can be inconvenient. Such a pipeline would have to be connected incrementally, starting with the source, and finishing with the sink, so that unconnected streams are always available to be split and joined. This makes it is impossible to create and save intermediate pieces of the pipeline or write utility methods to generate such pieces. To obviate this inconvenience, a more general mechanism is provided. We start by noting that the actual splitting and joining is performed by system tasks, reified by the following types.

```
public interface CompoundTask extends Task {
  // Covers all Tasks that have more than one port or more than one stream
}
public interface Splitter<Tin extends Value,Touts> extends CompoundTask {
  public Port<Tin> in();
  public Touts outs(); // A stream 'cluster'
}
public interface Joiner<Tins,Tout extends Value> extends CompoundTask {
  public Tins ins(); // A port 'cluster'
  public Stream<Tout> out();
}
```

The Lime generic type system cannot express exactly the constraints unifying the type parameters Tin and Touts or Tins and Tout. Rather, Lime treats most compound tasks specially for type-checking purposes. To understand how this works, we need a definition.

A stream or port *cluster* is one of the following.

1. A tuple all of whose members are streams (for a stream cluster) or all of whose members are ports (for a port cluster).

2. A bounded array whose element type is a stream or port.

The Tin type of a splitter or the Tout type of a joiner must be either tuple types or bounded value array types. We call these types the "splittable" types. For each splittable type there is a corresponding port cluster or stream cluster defined as follows.

1. If the splittable type is a tuple, the corresponding stream and port clusters are tuples as well.
2. If the splittable type is a bounded value array, the corresponding stream and port clusters are bounded non-value arrays.
3. The element type(s) of the streams (ports) that make up the corresponding stream and port clusters are the same as the element type(s) of the original splittable type.

The Touts type of a splitter must be the stream cluster corresponding to the Tin type. The Tins type of a joiner must be the port cluster corresponding to the Tout type. That is

```
Splitter<`(int, double),`(Stream<int>, Stream<double>)>
```

and

```
Joiner<Port<int>[2], int[[2]]>
```

are valid compound task types.

An instance of a splitter or joiner can be created with the operators **task split** *Type* and **task join** *Type*. Although made up of two keywords, separated by arbitrary amounts of white space, these should be thought of as single operations. The type must be a splittable type as defined above.

```
Splitter<`(int, double),`(Stream<int>,Stream<double>)> aSplitter
    = task split `(int, double);
Joiner<Port<int>[2], int[[2]]> aJoiner = task join int[[2]];
```

Note that Splitter and Joiner types have a very verbose syntax. For readability, it is recommended to use Lime's local type inference facility described in section 4 whenever creating and initializing a variable of Splitter or Joiner type.

```
var aSplitter = task split `(int, double);
var aJoiner = task join int[[2]];
```

15.10.4 Other CompoundTask Types

The Splitter and Joiner are special cases of the CompoundTask with one side that is a cluster and one side that is a simple port or stream. To complete the picture and permit Splitters and Joiners to be used in meaningful combination, we define additional types as follows.

```
public interface MultiSource<Touts> extends CompoundTask {
  public Touts outs(); // A stream 'cluster'
}
public interface MultiSink<Tins> extends CompoundTask {
  public Tins ins(); // A port 'cluster'
}
public interface MultiFilter<Tins, Touts> extends CompoundTask {
  public Tins ins(); // A port 'cluster'
  public Touts outs(); // A stream 'cluster'
}
```

These task types always denote aggregate tasks: there are no user or system tasks (ie. primitive tasks) that have these types.

The *compound task creation* operator **task** [...] is one way to create these task types.

```
var compoundFilter = task [ (3) task foo ];

// more flexibly, with N an ordinal type variable
var filters = new Filter<int, `(int,int)>[N];
for (N i) {
  filters[i] = task foo;
}
var compoundFilter = [ filters ];

// more complex example
Source<double> doubleSource = task ... ;
Filter<double,boolean> booleanFilter = task ... ;
Sink<int> intSink = task ... ;
var multi = task [ doublesource, booleanFilter, intSink ];
```

Between the square brackets of a compound task creation are either

1. A comma-separated list of one or more task expressions optionally prefixed with repeat counts (see section 12.5.1). As described earlier, repeat counts must be positive integral compile-time constants.

2. A single expression without a repeat count denoting a bounded array of tasks.

If the expression contains commas or repeat counts it denotes a potentially heterogeneous compound task. There must be at least two member tasks in that case. The elements must be SimpleTask types (compound tasks and ClosedGraphs can't be direct members of another compound task). The member tasks may be a mixture of sources, sinks, and filters. The resulting type is determined by aggregating the ports (if more than one) into a tuple and aggregating the streams (if more than one) into a tuple.

If the expression does not contain commas or repeat counts it must be a bounded array of some *specific* simple task type (Filter, Source or Sink, not the more general Task or SimpleTask). The resulting type is will have a port cluster (if any) that is a bounded array of the same

length as the compound, and a stream cluster (if any) that is a bounded array of the same length as the compound.

Because the actual size of a bounded array might depend on a type parameter, there is no restriction that the single array in a compound task expression must have at least two elements. However, the following should be born in mind.

- If the bounded array of tasks is restricted to always have a constant length of one, the compiler may issue a warning.
- The degenerate case (a compound task with only one member) is not the same as a simple filter. A "split-join" constructed using such a task may not perform as well as a simple pipeline. That being said, the behavior should correctly generalize to the width-one case.

The resulting type of a compound is determined as follows.

- If there is both a port cluster and a stream cluster, the type is a MultiFilter.
- If there is a port cluster and a single stream, the type is a Joiner.
- If there is a port cluster and no stream, the type is a MultiSink.
- If there is a single port and a stream cluster, the type is a Splitter.
- If there is no port and a stream cluster, the type is a MultiSource.

The four other combinations that involve no clusters in the result are impossible for the following reasons.

- No ports and no streams: could only arise if the compound task could consist of a single ClosedGraph but ClosedGraph members are not permitted.
- Single port or single stream: can technically happen if a compound task is made from a length one array of Source or Sink; however, that is treated as a degenerate cluster (an array of one element).
- Single port, single stream: similar to the previous in the case where it arises from a length one array of Filter. Otherwise, it is ruled out by the following restriction.

The simple tasks making up a compound task must either be all Sources, or all Sinks, or include at least one Filter. That is, it is illegal to build a compound task from a mixture of both Sources and Sinks not including any Filter. Such a compound task, were it possible, would have no meaningful dataflow; it it would simply be an arbitrary pairing of a (Multi) Source with a (Multi)Sink. This restriction rules out the possiblity that a compound task of more than one member could have just a single port and a single stream since, in that case, both would have to come from its Filter and its other simple tasks must contribute at least one more port or stream.

The rate of a MultiFilter or of a Splitter or Joiner constructed in this way is determined by the rates of their actual filters. If the sources and sinks are isolated, they will be "passive" and are either "pushed" or "pulled" at a rate commensurate with the Filters' rates. However, a non-isolated source or sink in this situation will have its own thread, just as in the simple case.

15.10.5 Compound Connect Operations

Having created a splitter or joiner task using the **task split** or **task join** operators, and having created compound tasks using the **task** [...] syntax, one might now legitimately ask how one connects the resulting compound tasks. One can, of course, "pick apart" the stream and port clusters and connect things individually. Using aSplitter and aJoiner from section 15.10.3, consider the following.

```
Filter<double,int> aFilter = task Some.staticMethod;
aSplitter.outs().0 => aFilter.in();
aFilter.out => aJoiner.ins()[0];
aSplitter.outs().1 => aJoiner.ins()[1];
```

What you end up with is a set of task objects. The port aSplitter.in() and the stream aJoiner.out() are unconnected, so, conceptually, what you have is a pipeline, which, for purposes of modular composition, you'd like to represent as an object of type Filter<`(**int**, **double**), **int**[[2]]>. But, Lime *deliberately* does not permit you to just create a Filter object from an arbitrary port and stream. The motivation for this restriction is schedulability, which is discussed in section 15.11.

There is a way out of this dilemma, but it (again, deliberately) does not work for the irregular problem posed above. For illustration, suppose we have the following splitters and joiners.

```
// aSplitter as defined above (splits `(int, double))
// aJoiner as defined above (joins (int[[2]]))
var splitter2 = task split double[[2]];
var joiner2 = task join `(double,boolean);
```

Then we have these methods.

```
int doubleToInt(double);
boolean doubleToBoolean(double);
double intToDouble(int)
```

We can now make these new filters.

```
var filter1 = splitter2 => task [ (2) task doubleToInt ] => aJoiner;
var filter2 = aSplitter => task [ task intToDouble, task doubleToBoolean ] => joiner2;
```

We were able to do this by exploiting the compound meanings of the => operator. The => operator can connect a wide variety of different task types. First, consider connections involving the "simple" side of a Splitter or Joiner.

1. Filter => Splitter yields Splitter.
2. Source => Splitter yields MultiSource.
3. Joiner => Filter yields Joiner.
4. Joiner => Sink yields MultiSink.
5. Joiner => Splitter yields MultiFilter.

For the preceding connection forms to succeed the type argument of the stream of the left-hand side must be convertable to the type argument of the port of the right-hand side.

Next, those CompoundTasks that have outs members can be connected to those with ins members.

1. Splitter => Joiner yields Filter.
2. Splitter => MultiFilter yields Splitter.
3. Splitter => MultiSink yields Sink.
4. MultiSource => Joiner yields Source.
5. MultiSource => MultiFilter yields MultiSource.
6. MultiSource => MultiSink yields ClosedGraph.
7. MultiFilter => Joiner yields Joiner.
8. MultiFilter => MultiFilter yields MultiFilter
9. MultiFilter => MultiSink yields MultiSink

For a compound connection to succeed, the stream cluster represented by the outs member of the left-hand side must have the same number of elements as the ins member of the right-hand side. They need not be the same kind of aggregate (one may be a bounded array and the other a homogeneous tuple). The elements must match one-for-one such that the type argument of the stream is convertable to the type argument of the port.

Thus, there are certain constrained sequences that ultimately result in making new filters, as well as many sequences that do not have that result. The ones that produce Filter results require that splitters, joiners, and filter aggregates be assembled directly without picking apart any split or joinable types of any splitters or joiners.

15.11 Schedulability of Task Graphs

The ability to split and join streams (or create splitter or joiner tasks) and to arbitrarily connect the split-off streams in any fashion that you might desire (other than cyclically) means that the sorts of graphs that might be produced are essentially arbitrary and many such graphs are not viable (they will deadlock or stall due to unbalanced rates). When lack of graph viability is detected during calculation of a static schedule, an exception might be thrown, but, in many cases, the scheduler must optimistically assume that a graph is viable unless proven otherwise. Such graphs often require dynamic dispatch mechanisms to sort out all the possible execution sequences.

Efficient synthesis in hardware requires that the graphs be schedulable in as static fashion as possible and it is also useful to know that a graph won't deadlock and won't stall until it is ready to terminate. Consequently, Lime takes steps to "encourage" the creation of schedulable graphs.

In particular,

1. Any Filter object, regardless of whether it is a primitive filter created with **task** or a complex pipeline or "split-join" created with the simple or aggregate connect operator, is individually schedulable.

2. Any graph consisting of a Filter object optionally connected to a single Source and optionally connected to a single Sink is schedulable.

Other graphs certainly *may* be schedulable, and the Lime compiler and runtime will make a best effort to discover whether an arbitrary graph, no matter how it was *actually* constructed, is one that *could have been* constructed in accordance with the preceding rules. However, for portions of a program that are intended to run with high efficiency in highly parallel hardware, it is good practice to construct graphs explicitly in a way that guarantees schedulability.

Note that, based on what we have said so far, not only will all Filter objects be schedulable, but they will, in fact, always have a rate ratio of 1:1. The latter property will not hold in general once we introduce rate matching in section 16.

15.12 New Grammar

See section 15.15 of the Java Language Specification for `UnaryExpression` and `UnaryExpressionNotPlusMinus`. See also section 15.8 for `Primary`, 6.5 for `AmbiguousName`, section 4.1 for `Type`, 4.3 for `ClassType`, 15.9 for `ArgumentList`, and 15.17 for `Expression`. `LimeBinaryOp` is defined in section 5 of this document. `LimeExpressionList` is defined in section 13 of this document.

```
UnaryExpressionNotPlusMinus     ::= LimeTaskExpression
LimeTaskExpression              ::= 'task' StreamMethodDescription
StreamMethodDescription         ::= 'value' UnaryExpression
```

```
StreamMethodDescription         ::= TaskFromOperator
StreamMethodDescription         ::= Isolate_opt StreamMethodId StreamMethodArgs_opt
StreamMethodDescription         ::= '[' LimeExpressionList ']'
Isolate                         ::= IsolateDescriptor '.'
IsolateDescriptor               ::= Primary
IsolateDescriptor               ::= AmbiguousName
IsolateDescriptor               ::= IsolatingConstruction
IsolatingConstruction           ::= ClassType '(' ArgumentList_opt ')'
StreamMethodId                  ::= IDENTIFIER
StreamMethodArgs                ::= '(' StreamMethodArgList ')'
StreamMethodArgList             ::= TypeOrArg
StreamMethodArgList             ::= StreamMethodArgList ',' TypeOrArg
TypeOrArg                       ::= Type
TypeOrArg                       ::= Expression
TaskFromOperator                ::= '{' OperatorAsMethod '}'
OperatorAsMethod                ::= BinaryInstanceOpMethod
OperatorAsMethod                ::= BinaryStaticOpMethod
OperatorAsMethod                ::= UnaryStaticOpMethod
BinaryInstanceOpMethod          ::= UnaryExpression LimeBinaryOp Type
BinaryInstanceOpMethod          ::= UnaryExpression '[' Type ']'
BinaryStaticOpMethod            ::= Type LimeBinaryOp Type
BinaryStaticOpMethod            ::= Type '[' Type ']'
UnaryStaticOpMethod             ::= LimeUnaryOp Type
```

See section 15.26 of the Java Language Specification. The following is replaced.

```
AssignmentExpression        ::= ConditionalExpression
AssignmentExpression        ::= Assignment
```

What replaces it is the following.

```
AssignmentExpression        ::= LimeConnectExpression
AssignmentExpression        ::= Assignment
LimeConnectExpression       ::= ConditionalExpression
LimeConnectExpression       ::= LimeConnectExpression '=>' ConditionalExpression
```

See section 14.8 of the Java Language Specification for `StatementExpression`. A `LimeConnectExpression` is also a `StatementExpression`.

```
StatementExpression         ::= LimeConnectExpression
```

The `UnaryExpressionNotPlusMinus` is further extended as follows.

```
UnaryExpressionNotPlusMinus     ::= LimeSplitExpression
UnaryExpressionNotPlusMinus     ::= LimeJoinExpression
UnaryExpressionNotPlusMinus     ::= LimeSplitterCreation
UnaryExpressionNotPlusMinus     ::= LimeJoinerCreation
LimeSplitExpression             ::= 'split' UnaryExpression
LimeJoinExpression              ::= 'join' UnaryExpression
LimeSplitterCreation            ::= 'task' 'split' Type
LimeJoinerCreation              ::= 'task' 'join' Type
```

Chapter 16

Rate Matching

So far we have only discussed how to create filters with a *rate* of 1:1. Indeed, all user-written filters have a unit (1:1) rate. Non-unit-rate filters are created using *rate matchers*.

The key is that Lime permits homogeneous aggregates to be used as both the arguments and return types of worker methods, so rate matching comes down to a problem of aggregation, disaggregation and reaggregation. Consider the following two tasks:

```
static local bit bitMaker() {...}
static local void bitConsumer(bit[[64]] block) {...}
...
Source<bit> bitMaker = task bitMaker;
Sink<bit[[64]]> bitConsumer = task bitConsumer;
bitMaker.out() => bitConsumer.in(); // illegal: bit != bit[[64]]
```

We can't connect these tasks directly even though they both deal in bits, because one produces a single bit at a time while the other wants to consume 64 bits at a time. The solution in Lime is simple:

```
bitMaker.out() => # => bitConsumer.in(); // works!
```

The match operator "#" creates a system filter (called a *matcher*) that only does aggregation, disaggregation or reaggregation; composing this filter with other filters that do computational work can then produce a rate-matched pipeline. Conceptually, the matcher task (in this case) requires 64 put operations on its port before it will satisfy one get operation on its stream (its rate is 64:1).

In the example shown, the "#" operator required no type information because it was between two connect operators, providing unambiguous information about the input and output types. More generally, the "#" operator is enclosed in parentheses with an optional port type parameter written before it and an optional stream type parameter written after it:

```
Filter<bit,bit[[64]]> bitRateMatcher = (bit # bit[[64]]);
```

Types may be elided only if the matcher is being immediately connected on the side where the type is elided. If both types are elided, then the parentheses may be elided.

Because a matcher is only allowed to affect aggregation and has no basis for doing any "deeper" conversions, there are some specific type rules relating a matcher's port type parameter to its stream type parameter:

1. Any value type in Lime has a corresponding *base value type*.
   (a) If a value type is not an array or a tuple, its base type is itself.
   (b) The base type of an array is the base type of the array's element type.
   (c) The base type of a tuple is determined as follows.
      i. If the base type of all the tuple's members is the same type, then the tuple is considered *homogeneous* and its base type is the common base type of its members.
      ii. Otherwise, the tuple is not homogeneous and its base type is itself.
2. All value types can be categorized as either *fixed size*, *simple variable size*, or *complex variable size*.
   (a) A homogeneous tuple or a bounded array whose elements are not fixed size is complex variable size.
   (b) An unbounded array whose elements are of the base type (no nested aggregation) is simple variable size. Otherwise, the array is complex variable size.
   (c) All other types are fixed size.
3. The port and stream type parameters of a matcher must have the same base type. Note that if neither of these type parameters is an aggregate, then this means the types must be the same, which is just a special case of the general behavior.
4. The stream type parameter of a matcher must not be complex variable size except in the special case where the port and stream type parameters are identical.

While the tuple is mainly a heterogeneous construct, we support homogeneous tuples for matching as a convenience since it is often clearer to write stream functions with a small number of arguments of the same type, rather than using a bounded array argument.

The behavior of the matcher is as follows.

1. If the port and stream type parameters are the same and the size directive is not used (see Section 16.1), the matcher behaves as an identity filter with a statically known rate of 1:1.
2. Otherwise, each (possibly aggregate) value that is received on the matcher's port is treated as a sequence of one or more values of the base type for the purpose of "filling" values to be provided on the matcher's stream. All levels of aggregation are "flattened."

(a) If the stream type parameter is simple variable size, then, there are two subcases.

i. If there is no size directive (see Section 16.1), then, for each value read from the port, the entire sequence of base type values is provided. The rate is still statically known to be 1:1.

ii. If there is a size directive, then, the matcher behaves as described in Section 16.1) and the rate is not known statically.

(b) If the stream type parameter is fixed size, then the matcher buffers the sequence of base type values internally and "doles them out" to the output values, releasing output values only when full.

3. When the port type parameter is not fixed size but the stream type parameter is fixed size or the size directive is used, then the rate of the matcher is not statically known.

4. When both the port type parameter and the stream type parameter are fixed size, then the rate of the matcher is statically known although, in general, it is not 1:1.

16.1 Size Directives

It is possible to add a directive to a match expression which imposes a fixed output size when (and only when) the output type is simple varying size. This size can be given by an arbitrary positive integral expression (not necessarily compile-time constant). This is useful when there will be a definite size at runtime but it is not known statically. The general syntax for a size directive uses the contextual keyword size (which is not a reserved word outside this one context).

```
var sizedMatcher = (float[[]] # float[[]], size 2*k);
```

The comma, the word size, and the expression following it appear within the parentheses after the output type, or immediately after the "#" symbol if the output type is elided.

Since the size expression need not be constant, it follows that checking for illegal values is dynamic and so there can be an IllegalArgumentException when the matcher is instantiated. The example above would throw this exception if k<1.

This is only one of several optional comma-separated directives that are supported (in addition to fill and shift as described in the following sections). When there is more than one directive, they can be specified in any order. The presence of any directive causes the outer parentheses to be required, even if both input and output type are elided.

16.2 Underflow Handling

When a matcher is buffering (ie, when the stream type is fixed size and the port type is not the same size), then there is an ambiguity about what happens when an underflow is detected on the matcher's port when its buffer is not empty. As discussed in section 15.8, Lime provides "orderly" drain semantics if underflow is detected "upstream" of a task before

any downstream tasks have terminated (buffered values should first be consumed before underflow is propagated). To meet these semantics, a matcher must decide whether to discard the partial buffer and immediately propagate underflow, or whether to provide one last value before propagating underflow. However, the matcher can only provide one more value if it has a way to fill out its partial buffer with pad values of some sort.

By default, the matcher will discard the partial buffer and immediately propagate underflow. This behavior can be changed by using a *fill directive* in the "#" expression.

```
typedef bit3 = `(bit, bit, bit);
typedef bit5 = `(bit, bit, bit, bit, bit);
Filter<bit3,bit5> matcher1 = (bit3 # bit5);
Filter<bit3,bit5> matcher2 = (bit3 # bit5, fill);
Filter<bit3,bit5> matcher3 = (bit3 # bit5, fill 1b);
```

A fill directive with no following expression means that the buffer will be filled with the default value of the base type. The optional expression following fill causes a different value of the base type to be used. The expression need not be a constant expression. Note that fill is not a reserved keyword in Lime but is reserved only in this context.

When the port of matcher1 underflows, it propagates the underflow immediately. In contrast, matcher2 will first fill any partial buffer out with 0b values send it and matcher3 will fill with 1b values.

16.3 Shifting

Normally, a port consumes values from its connected stream using get semantics, not peek semantics. Once a stream is connected, the explicit use of get and peek is no longer available. However, in some cases it is desirable to examine more values than are consumed. A limited peek operation on connected streams is optionally provided by a matcher along with (or instead of) its aggregate-management responsibility. The relevant directive is called shift. An expression is required following shift (the "shift count") and it determines the number of values consumed on each logical iteration, defined as follows.

1. Note that, due to aggregation, there is a difference between a physical get from the supplying stream and a logical get from the internal buffer of the matcher; similarly, there is a difference between a physical peek and a logical peek. The semantics of shifting is defined in terms of logical get and peek operations; the actual getting and peeking from the source stream is done conservatively and consistently with the logical requirement.

2. Shifting is only available in matchers for which the stream type parameter is fixed size or a size directive is used to impose a fixed output size. In either of those cases, the matcher behaves as if it executes an indefinite number of logical iterations. In each iteration, it does a number of logical get operations equal to the shift count, followed by a number of logical peek operations equal to the output size minus the shift count.

The sum of these provides enough values to satisfy one output on the matcher's output stream.

3. Of course, due to data availability on the port, the logical iterations just described may block somewhere in the middle. Lime does not specify exactly how the logical iterations are scheduled (by "pushing from the port", "pulling from the stream", dedicating a thread, etc).

For example, in the following code:

```
static local int f(int x, int y) { return x + y; }
void foo() {
  Stream<int> s1 = { 1, 2, 10, 20 };
  Stream<int> s2 = s1 => (#, shift 1) => task f;
}
```

the resulting stream s2 contains 3, 12, 30, whereas without the shift operator the result would simply be 3, 30.

It is an error to shift by fewer than one or by more than the output size. It is vacuous (though permitted) to shift by exactly the output size since, in that case, there will be no logical peeking (permitting this is useful since, in general, both the shift and the size might be dynamically determined). Since these restrictions can't always be statically checked, violations may result in IllegalArgumentException when the matcher is instantiated.

16.4 New Grammar

See section 15.12 of this manual for `LimeConnectExpression`. See section 4.1 of the Java Language Specification for `Type`.

```
LimeConnectExpression     ::= MatchExpression
LimeConnectExpression     ::= LimeConnectExpression '=>' MatchExpression
MatchExpression           ::= '#'
MatchExpression           ::= '(' PortItemType_opt '#' StreamItemType_opt MatchDirectives_opt ')'
PortItemType              ::= Type
StreamItemType            ::= Type
MatchDirectives           ::= MatchDirective
MatchDirectives           ::= MatchDirectives MatchDirective
MatchDirective            ::= ',' IDENTIFIER Expression_opt
```

The grammar for directives is supplemented by a check that only the prescribed directive names size, fill, and shift are used in match expressions.

Chapter 17

Messaging

Sometimes the regular (and typically high rate) flow of information through a graph of tasks must be supplemented by additional information whose characteristics are different. The additional information appears irregularly or at a lower rate. The sender and receiver might appear at very different parts of the graph and might even be upstream from the normal data-flow. The messaging feature of Lime is designed to handle these cases.

Messaging can be thought of as a broadcast mechanism that allows two actors to communicate even though they have no direct data connection, as long as both are in the graph (and meet some other restrictions to be described). The central mediating entity is a *message type* which is declared using an **interface** with the **message** modifier. For example,

```
public message interface Change {
  public void Change(int newState);
}
```

All of the methods of a message interface must return **void**, and all of the parameters must be of **value** type. Typically, such an interface will have only one method but that is not a requirement. Conceptually, a message interface represents a variant type whose cases are its different methods. Each case consists of the tuple made up of that method's arguments. In the above example, the message interface is declaring one message type Change which contains only a single integer. The methods of a message interface are considered to be **local**; the **local** keyword is permitted but not required.

User tasks can specify that they can receive this message by implementing the message interface in the class from which the task is created.

```
public class Receiver implements Change {
    private int state = 1;

    public local int receiver(int a, int b, int c) {
      return (a + b + c) * state;
    }

    public local void Change(int newState) {
```

```
      state = newState;
    }
  }
  ...
  => task Receiver().receiver =>
  ...
```

Methods used to implement message interfaces are **local** by default and will be checked accordingly. The **local** keyword is permitted for clarity but not required. As we will see later, the worker method and a message-receiving method will never be invoked concurrently so there is no race condition possible.

Note that to make meaningful use of a message, the receiving task must be stateful, meaning that it is sole-reference isolated and created using the form of **task** that specifies construction explicitly. If the task is stateless, then running a message-receiving method will not result in any discernible effect on any subsequent calls to the worker method. Lime does not enforce the rule that a receiving task must be stateful, since isolation properties are analyzed only when a **task** is created, not when a class is compiled. A stateless task implementing a message interface is safe, albeit probably useless. Lime may issue a suppressable warning if it can determine that this is happening.

Unlike the mechanism for indicating receiving of messages, a message sender is identified at the method level rather than at the class level. Specifically, a message is sent by

1. indicating that a method is a sender for a message interface
2. calling a method of the message interface in the body of the method
3. making that method into a task.

The form of task isolation used for the sender is not important; stateless tasks can be useful senders. Continuing with the above example, below is a sender that would send a message that would be picked up by the above task. The **sends** clause indicates that the method sender *may* send one or more messages from the Change interface with a delay of 2. If a zero delay is desired, one can either write @ 0 or omit that part of the clause. The meaning of delays will be discussed in the next section.

```
    public int sender(int a) sends Change @ 2 {
      count++;
      if (count == 4) Change(10);
      return count;
    }
  ...
  => task (...).sender =>
  ...
```

The **sends** clause on method declaration, if present, must come before the **throws** clause, if present. One or more message interfaces, separated by commas may be listed. The invocations of methods of a message interface are ignored and have no effect unless the method is made into a task (even then, they may be ignored, as will be explained).

A method with a **sends** clause is not obligated to invoke a message method on every invocation. In fact, if it does, it is probably abusing the message facility, which is designed for occasional traffic, not high volume traffic. Lime may issue a warning if a method that indicates use of a message interface in fact *never* uses it. The method is also not required to invoke each of its message interfaces at most once per activation. However, if it invokes a particular message interface more than once, only the last invocation will have any effect.

It is an error if more than one interface mentioned in a **sends** clause defines the identical method (as denoted by its name and argument types). This restriction sidesteps the issue of overlapping message interfaces.

Because the **sends** clause imposes such a weak obligation, and only needs to be considered when a method becomes the worker method of a task, the **sends** clause is only accepted on concrete method declarations, not abstract methods. This differs from **throws**, which requires strict checking.

17.1 Timing of Message Delivery

The timing of message delivery is probably the trickiest part of the semantics. Since the task graph is acyclic, for any two nodes in the task graph, there may exist directed paths from one to the other. If there are no such paths, we disallow messaging and the exception IllegalMessageException will be thrown at run-time if such a messaging is attempted. In the other case, the receiver will be either downstream or upstream from the sender and we then correspondingly call such a message downstream or upstream. Because the graph is acyclic, multiple task graph paths will be in the same direction though possibly of differing lengths.

17.1.1 Downstream Messaging

If a receiver is downstream from a sender, then there is one or more dataflow paths from the sender to the receiver. Each dataflow path can be thought of as potentially providing a sort of timing mechanism by which we can synchronize the timing of message delivery. Multiple dataflow paths can be reconciled by preferentially selecting the shortest path.

We can give a precise operational semantics on the timing through a tagging mechanism. On the iteration that a sender messages one or more downstream receivers, we tag the output that is generated by the sender with a newly generated tag corresponding to the message. Thereafter, that tag will remain on that data item and stick to any computation and result on which that data item participates in. The stickiness is conservative and applies regardless of whether the filter actually uses a data value by treating all tasks as black boxes. Thus the tag will flow through the task graph in a downstream manner and will eventually reach the receiver. Because there are potentially multiple paths connecting the sender and receiver,

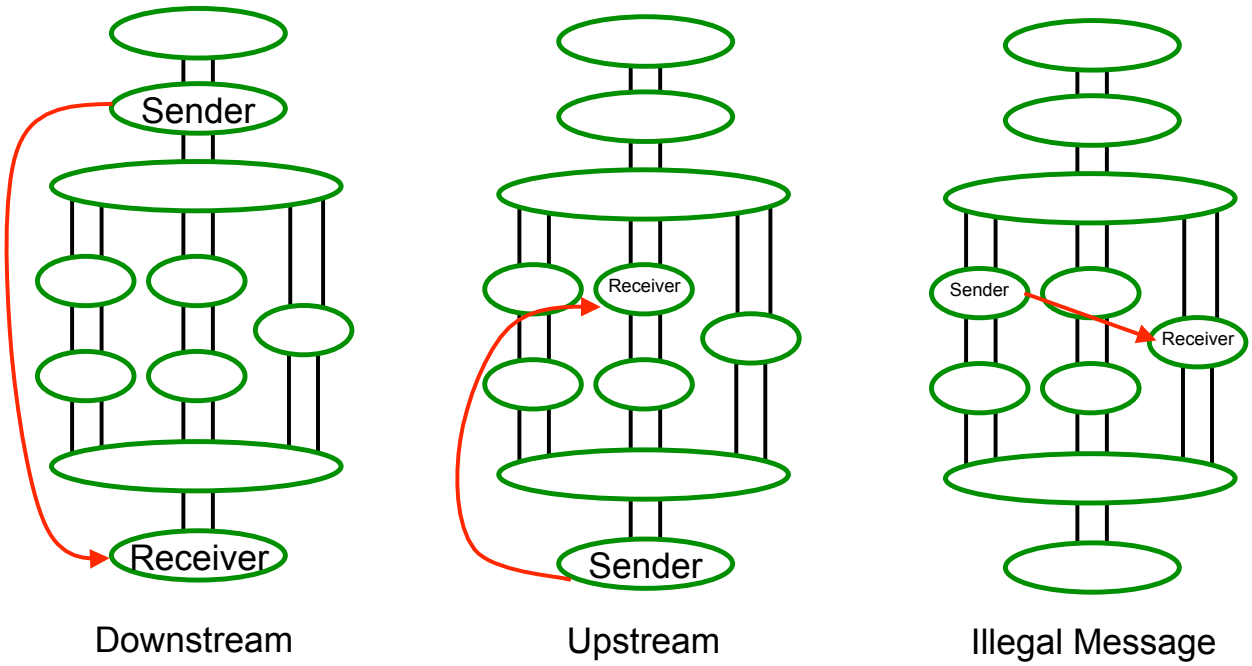

Figure 17.1: Different Messsage Types

the receiver may observe this tag multiple times. We choose the first time that the receiver observes this tag as the moment that corresponds to the tagging at the sender side. When a tag is first observed by a receiver of that message type, it will react to the message *prior* to running its worker method.

In the example in figure 17.2, we have put the sender and receiver examples from above and placed it into the depicted graph. We have modified the latency on the sender to a latency of zero for this example. That is, the clause `@ 3` should be replaced by the clause `@ 0`. In this example, we see that there are multiple paths from the sender to the receiver but the tagging mechanism will naturally discover the dynamically shortest path. We say "dynamically" because there there may be variable rate filters intervening and so it is not possible to always state which path will be the one taken. The tag flows through the intervening filters in a straightforward way where each output has the same tag as the input (regardless of the internal logic of the filter). When a splitter encounters a tagged data item, the tag is replicated and each output item is tagged by the tag on the incoming data item. A joiner behaves similarly and puts *all* the tags of its inputs onto its sole data output which can end up with multiple tags. In this example, because the same tag is showing up in the inputs, the duplicate tags are collapsed. Finally, note that the receiver will only respond to the tag only on the initial receipt and so the same tag on the final iteration is ignored and not even propagated. In general, propagation other than of the initial receipt is not necessary since the earlier tag must have already been propagated based on the treatment of all tasks as black boxes.

In general, a message can be sent with any non-negative constant integral delay. Consider

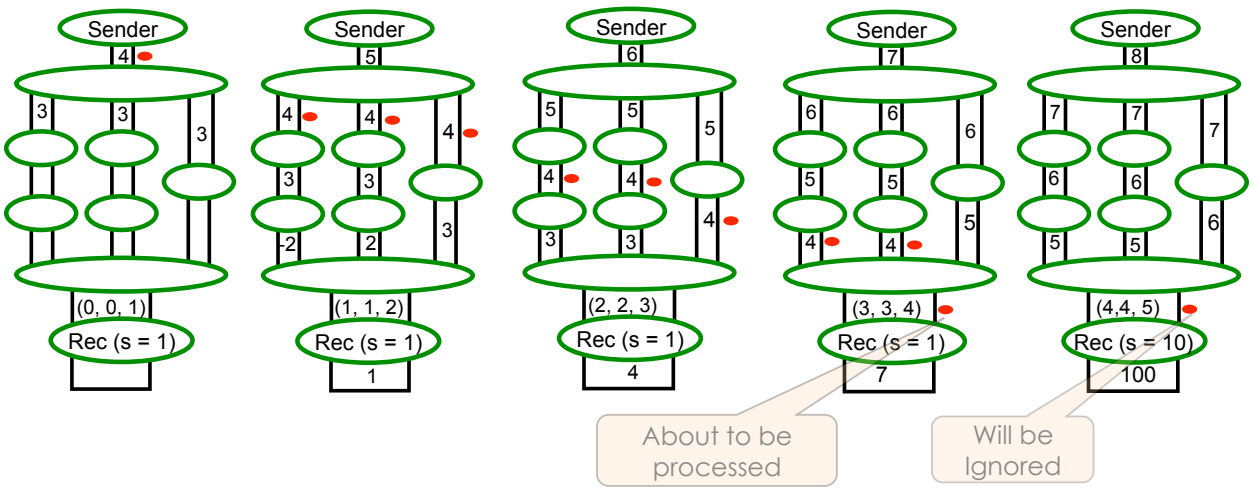

Figure 17.2: Downstream Messaging Example

the original example where the message was sent with a delay of 2. The timing is determined as before through a tagging scheme except that the tagging of the data item at the sender is delayed by 2 iterations. This feature is provided as a convenience as the delay can be entirely implemented by buffering in user code from the sender's work method. Figure 17.3 shows the same example with the delay.

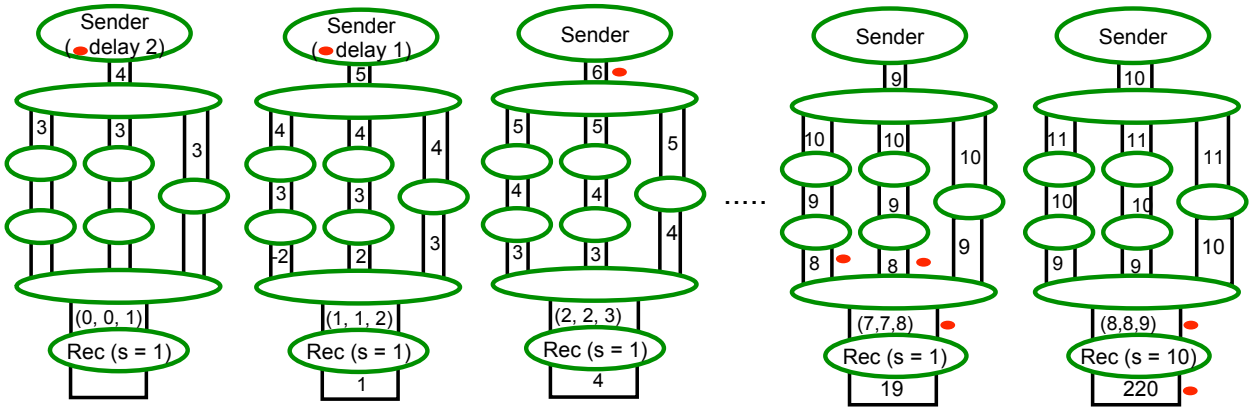

Figure 17.3: Downstream Messaging Example with Delay

17.1.2 Upstream Messaging

Because data flows downstream, the tags that represent messages also flow downstream and cannot be directly used to synchronize a sender to a receiver as before. Nevertheless, the path(s) connecting the upstream receiver to the downstream sender can establish synchronization by having the upstream receiver pre-emptively and continuously tag all data items that it 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. (Note that a sender can observe multiple tags from the same upstream receiver per iteration.) Assuming a delay of 0, the message should be delivered immediately before the sender invocation that would generate the next tag. We consider this case to be a delay of 0 since this is the soonest that a message can be delivered. Whenever there is a potential that a zero delay upstream message might be sent, the timing semantics prevents the upstream receiver from running ahead of the downstream portions of the graph at all since the potential message send must be processed first. This constrains scheduling but may be required to implement a tight feedback loop such as an IIR filter. We need a slightly different sender and receiver to give an example of upstream messaging using the same task graph we have been using.

```
public class Receiver implements Change {
    private int count = 1;
    public local int receiver() {
      state++;
      return state;
    }
    public local void Change(int newState) {
      count = newState;
    }
}

public class Sender {
    private int count = 0;
    public int sender(int a, int b, int c) sends Change @ 0 {
      count++;
      if (count == 1) Change(10);
      return a+b+c;
    }
}
```

In the earlier figure showing downstream messaging, we run every task as we go from one program state to the next since it is intuitive for high performance to extract maximal parallelism from the task graph. However, if we use this schedule, as shown in Figure 17.4, proper delivery of the message will be impossible. By the time the sender has sent the message, it is trying to deliver the message such that the receiver generates the data item tagged with the blue square (since the blue square immediately follows the red circle). However, the receiver has already run several iterations beyond this point.

In order for the upstream message to be delivered with zero delay, all upstream components between the sender and receiver cannot have run. This results in running things only when needed to provide data to downstream tasks. The resulting schedule is shown in Figure 17.5 where each time step corresponds to running only one stage of the task graph. When the message is being sent at the penultimate time step, the receiver has not run be-

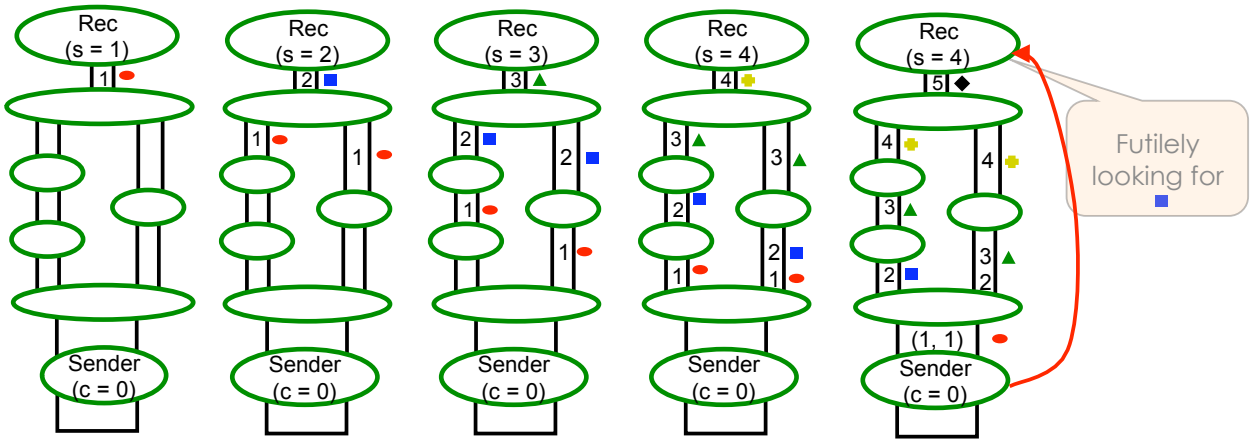

Figure 17.4: Upstream Messaging Example with overly eager schedule

yond the iteration that generated the tag (red circle) associated with the message and so it is able to receive the message just before the iteration that would generate the next tag (blue square).

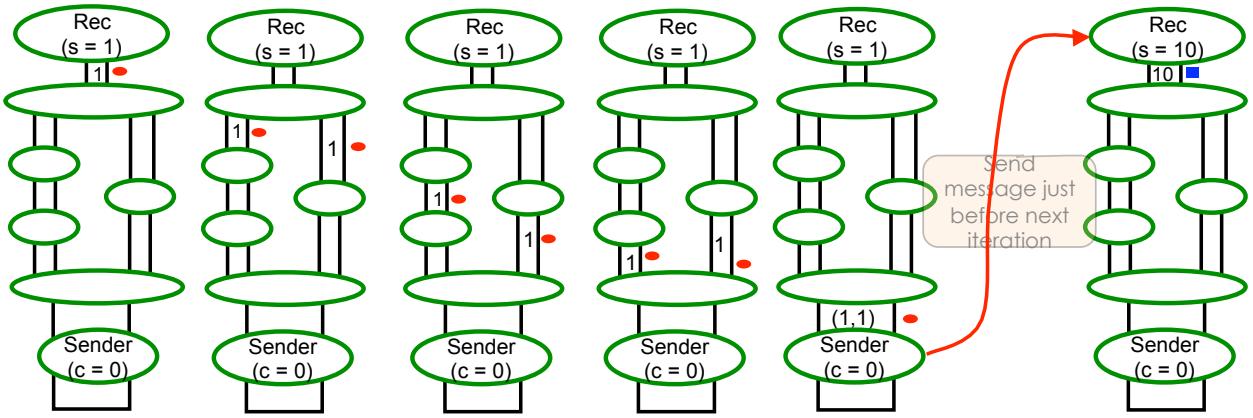

Figure 17.5: Upstream Messaging Example with lazy schedule can handle zero delay

Upstream messages can be also sent with a positive delay d. Like downstream messages, the delay has the effect of delaying the message send at the sender side by d iterations. For example, if we had sent the upstream message with a delay of 4, then the schedule in Figure 17.4 would have worked because we would be targeting the tag just after the black diamond tag. Since the receiver had just generated the black diamond tag, we are just in time to deliver the message. The delay of 4 corresponds to the length of the longest path connecting the receiver and sender.

17.1.3 Unspecified Delay

As just shown, messaging with specified delays corresponds to message deliveries at a precise iteration. This means that the regular dataflow must be tightly coupled with message flow so the runtime implementation of the latter is somewhat hampered by whatever choices may have been made in the former. In addition, upstream messaging can restrict the scheduling of the tasks because the upstream message can put all connecting paths into lockstep iteration. However, there are situations in which the application logic makes the timing irrelevant to the correctness of the algorithm. Typically, the algorithm only requires that the message is delivered reasonably soon. For example, algorithms in which a work queue is initially populated but can be further populated by internal computations would fit this pattern as long as the work items are independent. Such an application might be modeled by a graph in which some designated task representing the work queue can receive messages (work items) from any other task. The independence of the work items means that it does not matter when the work queue task receives work item messages as long as they are received at some point. Eventual receipt is necessary to ensure that all work items are completed before termination.

Using the example from above, the sender would indicate an unspecified delay with a question mark as follows

```
public class Sender {
    ...
    public int sender(int a, int b, int c) sends Change @ ? {
                ...
    }
}
```

The use of an unspecified delay is encouraged as it frees the runtime to use a more efficient implementation. For example, messages with unspecified delays destined for a particular receiver can be added to a buffer whose contents can be delivered asynchronously and in bulk to the receiver. We can even have multiple buffers for the same receiver as there is not a guarantee of the message ordering when the delay is unspecified.

17.1.4 Matchers

In our examples, each tag was associated with a single value and there was no ambiguity about when a particular tag was being propagated. However, matchers can consume and generate multiple values per iteration and so there is potential confusion about how tags are treated as they pass through matchers. For simplicity, we use the rule that a tag flows out of the matcher on the same iteration as when it flowed into the matcher. That is, we treat matchers no differently than if it was a regular task. This means that although matchers do not change base values, it may move the tag away from the output value containing the base value originally held the tag upon input.

17.1.5 Discussion

The example of upstream messaging in Figure 17.5 shows that a delay of zero corresponds to the earliest possible delivery as constrained by causality. That is, regardless of implementation details like scheduling of tasks, the delivery of the message cannot happen any sooner that the iteration corresponding to zero delay. However, for the downstream case, we can buffer results to an arbitrary degree so that the receiver can be running an iteration that is arbitrarily earlier than the one by the receiver. Thus, the sender can seemingly send a message back in time leading to a possible extension of delays to negative values. However, we do not allow this case because of the complexity in the programming model and implementation details. It is also not obvious what the use case would be that would motivate this addition.

Although we restrict delays to be constant, nothing in the definition of message timing requires compile-time constancy. In practice, knowing the actual delay can greatly facilitate optimal scheduling. In particular, if there is a possibility of an upstream message with variable delay, then a portion of the graph must be run in lockstep which precludes parallel execution. This is a feature we may add in the future.

In general, better performance is obtained by using the weakest constraint. Whenever possible, use an unspecified delay. Otherwise, larger delays are better than smaller delays. If we add variable delays, it is still preferable to use constant delays where possible.

17.2 Java Compatibility

The semantics of the **message** interface type is not understood by Java. However, classes that implement such interfaces can be passed to Java. The interface is seen by Java as an ordinary interface and the receiving methods as ordinary methods.

Classes with methods that have **sends** clauses can be passed to Java. Both the **sends** clauses and all invocations of sending methods are compiled by Lime in a way that causes these invocations to have no effect outside of a task graph. Thus, Java invocations of methods with **sends** clauses are well-defined, it is just that the sending will have no effect.

17.3 New Grammar

See section 9.1.1 of the Java Language Specification for `InterfaceModifier`.

```
InterfaceModifier      ::= 'message'
```

See section 8.4.1 of the Java Language Specification for `MethodHeader`. The original definition is as follows.

```
MethodHeader ::= MethodModifiers_opt TypeParameters_opt ResultType MethodDeclarator
```

Throws$_{opt}$

The definition is altered as follows. See section 4.3 of the Java language specification for InterfaceType.

```
MethodHeader     ::= MethodModifiers_opt TypeParameters_opt ResultType MethodDeclarator
                 LimeSends_opt Throws_opt
LimeSends        ::= 'sends' MsgInterfaceList
MsgInterfaceList ::= MsgInterface
MsgInterfaceList ::= MsgInterfaceList ',' MsgInterface
MsgInterface     ::= InterfaceType
MsgInterface     ::= InterfaceType '@' DelaySpecification
DelaySpecification::= ConstantExpression
DelaySpecification::= '?'
```

Chapter 18

Collective Operations and Reductions

18.1 Collective Operations

Collective operations are the application of an operation across all elements of a collection, yielding a new collection with the resulting values. Arrays and the standard lime.util collections support collective operations, as well as other types like string. User-defined classes may also support collective operations by implementing the Indexable interface and one or more of the Collectable, ValueCollectable, or ReflexiveCollectable interfaces.

The "@" character indicates a collective operation. Collective operations 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.

Collective operations may be used with methods of any number of parameters, and the parameters may be a combination of either Indexable 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 = { "foo", "bar" };
string[] v2 = v1@toUpperCase(); // v2=={"FOO", "BAR"}

int[] a1 = { 1, 2, 3 };
int[] a2 = { 0, 0, 0 };
int[] a3 = a2 @+ 1; // a3=={ 1, 1, 1 }
int[] a4 = a3 @+ a1; // a4=={ 2, 3, 4 }
```

If the collections are not of the same size, a DomainConformanceException is thrown.

The ability to mix Indexable and scalar parameters interacts with method resolution in the presence of multiple methods of the same name. As in standard Java method resolution, a method whose arguments match exactly is preferred over one that requires an upcast or widening conversion. If there is more that one inexact match, the expression is ambiguous and a compile-time error is indicated. Sometimes, a particular argument might be interpreted

as both a scalar and an indexable; for example, string implements Indexable but may be a scalar participant in an operation involving (for example) an array of strings. An Indexable match is considered less exact than a scalar one (the less surprising choice in common cases like those involving strings), so there are generally more ways that a method call can be ambiguous when it is collectivized.

In order to be eligible for collective operations, the collection must implement the following interfaces:

```
// Required
public universal interface Indexable<I extends Value, E> {
    local E this [I index];
    local Iteratable<I> domain();
}

// Required unless one of the extended forms used
public universal interface Collectable<I extends Value> {
    local <E> Collector<I, E> collector();
}

// Optional (use if value results are to be handled distinctly)
public universal interface ValueCollectable<I extends Value>
        extends Collectable<I> {
    local <E extends Value> Collector<I, E> valueCollector();
}

// Optional (use if results that match the Collectable collection's
// element type are to be handled distinctly)
public universal interface ReflexiveCollectable<I extends Value, E>
        extends Collectable<I> {
    local Collector<I, E> reflexiveCollector();
}
```

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

The Collectable interface provides 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. There are up to four ways to design a collection to handle collective operations.

1. If the same kind of result collection is to be used regardless of the result type of the per-element operation, then the simple Collectable interface is sufficient. Note that this result type bears no necessary resemblance to the element type of the collection as it depends on the operation that is collectivized and all of the operands.

2. To use a different result collection type for **value** per-element results and non-**value** per-element results (so that the entire collection can be made into a value in the case of value results) use the ValueCollectable interface. This option adds a distinct method, valueCollector() which is called if and only if the result of the per-element operation is a **value** type. Lime value arrays implement ValueCollectable so that they can return instances of themselves parameterized by the result value type when the result is a value type.

3. To use a distinct collection type for per-element results that exactly match the type of the original collection, use the ReflexiveCollectable interface. This option adds a distinct method reflexiveCollector() which is called if and only if the result of the per-element operation exactly matches the collection's element type. The string class implements ReflexiveCollectable so that it can return a string rather than the more general **char**[] when the per-element operation produces a **char** result.

4. It is possible to implement both ValueCollectable and ReflexiveCollectable to obtain up to a three-way distinction in the kinds of result collections produced.

The I (index) parameter of both the Collectable and Indexable interfaces implemented by a collection must be the same type. In addition, if ReflexiveCollector is implemented, the E (element) parameter of both the ReflexiveCollector and Indexable interfaces must be the same type.

The collector provided by any of the three kinds of Collectable collections must implement the Collector interface:

```
public universal interface Collector<I extends Value, E> {
    void this [I index](E val);
    Indexable<I,E> result();
}
```

The indexed store operation allows the individual results to be collected, and the result() method returns a new collection which must also be Indexable. The precise implementation of Indexable is jointly determined as follows.

- The generic type of the Indexable is determined by the Collector implementation, which is, in turn, determined by the collector() or valueCollector() or reflexiveCollector() implementation.

- The parameter I (the index type) is the same as that of the original collection since it is passed through at every step.

- The element type E is the same as that with which the Collector was instantiated, which was in turn set by the compiler to be equal to the result type of the element-wise operation.

The semantics of a collective operation dst = src1@foo(src2) (of type C, element type E and index type I) are given by the following expansion:

```
if (! src1.domain().equals(src2.domain())) { throw new
    DomainConformanceException(); }
Collector<I,E> tmp = src1.collector(); // or valueCollector() or reflexiveCollector()
for (I index: src1.domain()) { tmp[index] = src1[index].foo(src2[index]); }
C dst = tmp.result();
```

The single evaluation of any scalar arguments occurs conceptually before any of the code shown above, so side-effects during those evaluations can affect the creation of the collector but not vice versa.

The other collective parameters of a collective operation need not implement Collectable, but only Indexable.

For the most part, Lime's provided classes are implemented and checked exactly as so-far described, but there are additional issues when bounded arrays are involved.

1. If any participant in a collective operation is a bounded array, then all such participants must be scalars or else bounded arrays of the same size (thus, the DomainConformanceException can't arise in the bounded case).

2. In the bounded case as just described, the result will also be a bounded array of the same size.

Static methods can also be used as collective operations. In this case, the @ character is placed before the Collectable parameter whose Collector will be used to gather the result. For example:

```
int a5[] = Math.max(@a1, { 99, 1, 1 }); // a5=={99,3,4}
int a6[] = Math.max(9, @a5); // a6={99,9,9}
string s1 = "fooBAR";
static char flipCase(char c) {
    return Character.isLowerCase(c) ? Character.toUpperCase(c) : Character.
        toLowerCase(c);
}
string s2 = flipCase(@s1); // s2=="FOObar"
```

More generally, collective operations can be applied, for example, to hash tables with the same key sets (that is, the results of their domain() methods will compare to be .equals()). Thus if two hash tables x and y contain {("foo", 1), ("bar", 2)} and {("foo", 10), ("bar", 10)} then the result of x @+ y is {("foo", 11), ("bar", 12)}.

18.1.1 Data Parallelism

When supported by the underlying target architecture, and when it can be done without changing the program semantics, collective operations will under certain circumstances be implemented in a manner that takes advantage of parallel operation.

The parallel behavior is expected to obtain with a high degree of predictability (assuming support from the underlying platform) when all of the following are true. Note that any implementation is free to perform a larger set of operations in parallel.

- The element types of the collections are values
- The iterated operation is either one between primitive types or else the method that implements it is inherently isolated as defined in section 15.3.1.
- The Collector implementation does not depend on the order in which calls to its indexed set operations are made, and it is safe to make those calls concurrently.

The first two properties are readily verified both by the programmer and by the compiler. The third property is readily understood but the compiler's check is necessarily conservative. A Collector that

1. uses a Lime array to store values
2. directly translates its own index type (a bounded type or **int**) into an array index without arithmetic and
3. performs no other actions in its indexed set method

will be recognized as order-independent. The provided Collector implementations for the arrays themselves have this property.

18.2 Reduction

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

Reduction can be applied to any Iterable class, although it is recommended that classes instead implement Iteratable; otherwise the reductions will not be available in **local** methods.

Either instance or static methods can be used for reduction. In either case, they must apply to 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 @@. For example:

```
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
```

If a reduction is applied to zero items, an EmptyReductionException is thrown. That is, the semantics of @@+ a above is ((1 + 2)+ 3) and not (((0 + 1)+ 2)+ 3).

18.2.1 Optimization

If the class being reduced also implements Indexable, the compiler may optimize the operation using various parallel reduction techniques. However, such optimizations will only be applied if the operator in question is annotated with @Axioms({associative}). Specifying commutative as well will expose further optimization opportunities.

In a **local** method, the compiler is restricted to use a deterministic parallel evaluation strategy.

.

18.3 Java Compatibility

There are no new types introduced in this section and hence no implications for Java Compatibility. Collective operations don't arise in Java code, but there is no restriction on their use in Lime methods that might be called from Java code.

18.4 New Grammar

We first revise the definition of `MethodInvocation` in section 15.12 of the Java Language Specification to the following semantically equivalent definition.

```
MethodInvocation                    ::= SimpleMethodInvocation
MethodInvocation                    ::= NamedParameterizedMethodInvocation
MethodInvocation                    ::= DottedMethodInvocation
SimpleMethodInvocation              ::= Name '(' ArgumentList_opt ')'
NamedParameterizedMethodInvocation  ::= Name '.'  TypeArguments
      IDENTIFIER '(' ArgumentList_opt ')'
DottedMethodInvocation              ::= MethodReceiverNotName '.'  TypeArguments_opt
      IDENTIFIER '(' ArgumentList_opt ')'
MethodReceiver                      ::= Name
MethodReceiver                      ::= MethodReceiverNotName
MethodReceiverNotName               ::= Primary
MethodReceiverNotName               ::= 'super'
```

We define `ArgumentList` using the revision shown in section 12.9 of this manual, which defines the `ArgumentExpression` production.

We then define the collective operations and reductions as follows. See the Java Language Specification for `TypeArguments` (4.5.1), `Expression` (15.27), `UnaryExpression` (15.15),

and `MultiplicativeExpression` (15.17). See section 5.4 of this manual for `LimeUnaryOp` and `LimeBinaryOp`.

```
MethodInvocation                ::= LimeCollectiveMethodInvocation
MethodInvocation                ::= LimeReducingMethodInvocation
ArgumentExpression              ::= LimeCollectorArgument
ArgumentExpression              ::= LimeReducerArgument
LimeCollectiveMethodInvocation  ::= MethodReceiver '@' TypeArguments_opt
      IDENTIFIER '(' ArgumentList_opt ')'
LimeReducingMethodInvocation    ::= MethodReceiver '@@' TypeArguments_opt IDENTIFIER
LimeCollectorArgument           ::= '@' Expression
LimeReducerArgument             ::= '@@' Expression
LimeUnaryCollectiveOperation    ::= '@' LimeUnaryOp UnaryExpression
LimeReduction                   ::= '@@' ReductionOp UnaryExpression
ReductionOp                     ::= LimeBinaryOp
LimeBinaryCollectiveOperation   ::= MultiplicativeExpression '@' LimeBinaryOp UnaryExpression
```

Chapter 19

The "Closed World" Model

The Lime language, like Java, supports separate compilation and dynamic class loading: no closed world is assumed in the language as a whole. However, a stated purpose of Lime is to support synthesis of code into hardware. In hardware, at least in the near term, there is no dynamic class loading and limited capacity to consume arbitrary class metadata. This can make it problematic to use non-final types in hardware, when the set of possibly extenders or implementers of the type is open-ended. In the absence of any hints from programmers, many programs that would otherwise be synthesizable in hardware may end up running in software.

Thus, Lime supports "closing the world" around parts of a class hierarchy to ensure that hardware synthesis will be possible. The support takes the form of an **extendedby** keyword, which follows after 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.
3. An **extendedby** clause may not require a cycle in the type hierarchy (such cycles are already illegal in Java and Lime).

It is the 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 suppressable warning that the type extension subhierarchy is not closed.

We can now state the closed world model for hardware synthesis a little more precisely.

1. A *closed type* is a type that is either **final** or has an **extendedby** clause such that all of its possible extenders are closed types.

2. The *initial types* of a task are the type declaring its worker method plus the type of any curried arguments to that worker method.

3. A *synthesizable realm* consists of a set of Lime tasks that are connected to form a graph. The graph may be a subgraph of a larger one.

4. The *entry types* of a synthesizable realm are the types of any ports of tasks of the realm that are not fed by other tasks of the realm, plus the initial types of all the tasks in the realm. Note that types created inside the realm (via the **new** operator) are not considered entry types: their exact types are always statically known. Note also that reflection (e.g. Class.forName) is not available in hardware.

5. A synthesizable realm is a closed world if all its entry types are closed types and there are no reflective object creations.

It is never an error to write a Lime program that does not use **extendedby**. Once **extendedby** is used, it must be used correctly as defined above. The closed world model will then affect what is synthesized to hardware.

19.1 New Grammar

See section 8.1 of the Java specification for `NormalClassDeclaration`, section 9.1 for `InterfaceDeclaration`, and section 4.3 for `ClassOrInterfaceType`. The original productions as modified in section 2 of this document are further modified as follows.

```
NormalClassDeclaration   ::= ...  'class' ...  Interfaces_opt ExtendedBy_opt
InterfaceDeclaration     ::= ...  'interface' ...  ExtendsInterfaces_opt ExtendedBy_opt
ExtendedBy               ::= 'extendedby' ClassOrInterfaceTypeList
ClassOrInterfaceTypeList ::= ClassOrInterfaceType
ClassOrInterfaceTypeList ::= TypeList ',' ClassOrInterfaceType
```

Chapter 20

Java Compatibility

20.1 Source Compatibility with Java

Except for certain exceptions, a legal Java program is a legal Lime program. The deviations fall into two categories: lexical and semantic.

20.1.1 Lexical Issues: Keywords

A legal Java program might not be a legal Lime program for purely lexical reasons, because Lime reserves many words that are not reserved in Java. For example, **task** is a reserved word in Lime but might be an identifier in some Java programs. All such problems can be fixed by renaming variables, without loss of meaning from the original Java.

When importing Java code into Lime, the back-tick character "`" can be used to prefix a Lime keyword that should be treated as a Java identifier. For instance, **split** is a keyword in Lime. However, it is used in Java as the name of a method of the String class. To invoke this method from Lime:

```
String s = ...;
String[] pieces = s.`split("/");
```

The identifier "_" (that is, a single underscore), is a legal Java identifier but is reserved in Lime (see Section 13.2). It can also be named using "`_".

20.1.2 Semantic Issues

Three constructs of Java (generic types, generic methods, and arrays) are given expanded semantics in Lime, such that the same syntax has an expanded meaning. Because the Lime definitions are a supertype of the Java ones, it often just works to take Java code as is and compile it as Lime code. However, this is not guaranteed for at least two reasons. First, Java generics permit some type correctness "cheating" through the use of raw types and unchecked conversions. If a program depends on these, it may fail in the stricter checking

environment of Lime. Second, there may be untranslated Java code with which a Lime program must interoperate.

The tilde escape character "~" immediately before the declaration of generic type parameters or generic method parameters, or before the opening square bracket of an array declarator, indicates a Java compatible generic type, generic method, or array, respectively. Examples are shown in sections 2 and 12. Note that code that is separately compiled with a Java compiler is automatically recognized as Java and not Lime; it is not necessary to recompile such code with a Lime compiler using tilde escapes.

20.2 Binary Compatibility with Java

Previously we considered the case where a Java source program was compiled using a Lime compiler. Here we consider the case where the Java source files were previously compiled by a Java compiler and exist only as class files.

Lime is binary compatible with Java in the sense that a Lime main program can directly or indirectly load and instantiate Java classes and invoke their methods. The opposite behavior is not guaranteed. That is, a Java main program cannot load and instantiate Lime classes.

When a Lime program passes reference arguments to a Java class, it is possible that the methods of some Java class will end up calling the methods of Lime objects. Correct behavior is only guaranteed when those Lime objects fall into a certain compatible subset of Lime. This compatible subset is defined in terms of types (only the type of the object and the types of its visible fields and methods matter in deciding whether it is safe to pass to a Java method, not what its methods may happen to do internally).

Throughout this manual we identify, for each Lime innovation that affects the type system, whether or not the resulting type is Java-compatible.

20.3 Gotchas

There are a few corner cases regarding Java compatibility that cause surprising behavior, some of it not realized until run-time. Here we try to enumerate these situations so they can be avoided.

20.3.1 String Equality

A problem arises when string and String types are mixed and treated as Object types. As described in Section 11.1.2, a Java String will not compare as .equals() to the corresponding Lime string. In addition to the potential problems with individual variables, if a collection is used with Object element type, unexected behavior may occur. In particular,

```
var set = new HashSet<Object>();
string foo1 = "foo";
set.add(foo1); // succeeds
set.add("foo"); // literal has type java.lang.String; succeeds
```

will produce the (probably) unexpected result of a set with two copies of "foo".

20.4 The pH Tool

There are certain cases where binary compatibility with Java can be substantially enhanced by selectively adding certain Lime modifiers to existing Java classes, interfaces, enums, and methods. The pH tool (so-called because it "adds Lime") will do this, and will perform static checks to ensure that type safety is maintained. Specifically,

- Any Java interface may be labelled **universal**, as long as all of its superinterfaces are so-labelled either previously or at the same time.
- Any Java class may be labelled **universal**, as long as all of its supertypes are labelled universal either previously or at the same time and its existing fields and methods meet the requirements for universal types. If such a class has fields and is not **final** (meaning that it may subsequently have **value** subtypes), code is generated to support contents-based equality testing.
- Any **final** or **static** Java method and any Java constructor may be labelled **local** or **glocal** if it meets the definition required for these properties. Note that it would not be safe to so-label a non-final instance method because subsequent subclassing could produce an unsafe state. Also, the rules are necessarily stricter for Java methods because the only "repeatable" **static** fields in Java classes are compile time constants. To get a set of Java methods to pass the **local** or **glocal** restrictions it may be necessary to label them as a group.

20.5 New Grammar

The use of the backtick ("`") as an escape for identifiers is a change to the lexical grammar for identifiers (section 3.8 of the Java Language Specification). The following production is added.

```
Identifier ::= '`' IdentifierChars
```

Although the formal statement does not capture the following subtlety, it is nevertheless true. The backtick escape is always elided and results in a seeming identifier that might otherwise match a keyword. If the resulting identifier has the same characters as a keyword newly introduced by the Lime language (a word that is not reserved in Java) it will be a

legal identifier. However, if the resulting character sequence matches a word that was already reserved in the Java language, the result is still erroneous. In part, this makes a virtue of a limitation in our current implementation (which relies on being able to translate Lime to syntactically legal Java). However, it is a reasonable limitation, since the backtick is just an escape mechanism to support Java compatibility, and having this restriction enables similar implementations that may be undertaken by others.

The grammar surrounding the tilde escapes for generics and arrays are documented in those sections.

Chapter 21

Testing

21.1 Random Values

In order to promote systematic testing, most of the classes in lime.lang and lime.util implement the RandomlyGenerable interface, which provides one method, randomValue(), that takes a random generator and uses it to produce a random value of the class:

```
public universal interface RandomlyGenerable<T> {
    glocal T randomValue(IRandom generator);
}
```

This creates a random value but requires an instance of the object to do so; by convention, such classes also include a static method with the same functionality:

```
public static glocal T random(IRandom generator);
```

For example:

```
IRandom generator = ...;
uint randomUnsigned = uint.random(generator);
BoundedMap<color> randomMap = BoundedMap<color>.random(generator);
Set<int> s = ...;
Set<int> randomIntSet = s.randomValue(generator);
```

There are several random generators implementing IRandom in lime.util. Some of them use external (global) random sources, like the nanosecond clock, and will therefore cause invocations of randomValue() to be **global** method calls; others use a deterministic pseudo-random algorithm and are therefore **local**.

For some classes, it may not be statically determinable whether they can generate random instances of themselves because it depends on the types of their contained values. For instance, ArrayVSet implements RandomlyGenerable, but if it contains an element type that is not itself RandomlyGenerable, then it may throw a Contained−Value−Not−Randomly−Generable−Exception; if for some other reason a class can not generate a random value, it can throw a Random−Value−Generation−Exception.

21.2 Interface Testing

With a disciplined programming practice, the operations provided by an interface often obey certain invariants, pre-conditions, and post-conditions. The random value generation methods described above can be used to test whether an implementation of an interface obeys these properties.

The package lime.test contains such tests for all of the the key interfaces in lime.lang as well as a testing framework that can be used to build ones' own tests. Programmers choosing to implement these interfaces from lime.lang should use these tests to ensure that they honor the associated contracts.

Programmers are also encouraged to think carefully before overriding the operators used in these interfaces when their overridden operators have a semantics different from that specified in the core Lime interfaces. Disciplined use of operator overloading makes programs more maintainable and easier to understand.

The key interfaces are:

- comparable
- dense
- bounded
- numeric
- logical
- integral
- IBinaryWord
- IBinaryNumber
- IBinaryInteger
- Indexable
- Concatenable

The testing framework will repeatedly generate random values of the class being tested, and invoke the programmer-supplied test() method. A section of the testing method for the logical class is shown below:

```
public class LogicalTest<T extends logical<T> & RandomlyGenerable<T>>
    extends TestBase<T>
{
  public void test(T t, T u, T v) {
    verify("involution", eq(t, ~ ~ t));
    verify("and idempotency", eq(t & t, t));
    verify("and commutativity", eq(t & u, u & t));
    verify("and associativity", eq(t & (u & v), (t & u) & v));
    ...
```

```
    }

    ...
}
```

Acknowledgments

Shan Shan Huang (Georgia Tech) made substantial contributions to an earlier version of the Lime language during her internship at IBM in 2007.

Feedback from the entire Liquid Metal team has improved the language. In addition to the manual's authors and Shan Shan, the team includes Amir Hormati (University of Michigan), an intern in 2007 and 2008, Andre Hagiescu (National University of Singapore), an intern in 2008, Myron King (MIT), an intern in 2009, Christophe Dubach (University of Edinburgh), visiting scientist in 2010, and Sunil Shukla, a post-doctoral fellow in 2010-2011.

We also thank Doug Lea and David Ungar for their valuable comments on earlier versions of this document.

Bibliography

[1] J. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2010.

[2] D. F. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency—Practice and Experience*, 15(3–5):185–206, Mar.–Apr. 2003.

[3] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer, 2008.

[4] J. Sasitorn and R. Cartwright. Efficient first-class generics on stock java virtual machines. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1621–1628, New York, NY, USA, 2006. ACM.

[5] M. Viroli and A. Natali. Parametric polymorphism in java: an approach to translation based on reflective features. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 146–165, New York, NY, USA, 2000. ACM.