StreamIt Language Specification
Version 2.0*

streamit@cag.lcs.mit.edu

October 17, 2003

Contents

*This document is the latest working version of the language specification.
`$Id: streamit-lang.tex,v 1.24 2003/10/17 11:55:23 thies Exp $`

1 Introduction

StreamIt is a language intended to simplify coding of signal-processing and other streaming computations. The programmer constructs a stream graph, containing blocks with a single input and a single output, and describes the function of atomic blocks and the structure of composite blocks. The compiler generates code for each block, and applies optimizations to the stream graph to produce efficient code for the target architecture.

The current implementation of the StreamIt compiler translates the syntax described in this document to Java code, which can then be either run against a Java library or compiled to C code and linked with a runtime library. Since the compiler can reconstruct the stream graph, it can combine adjacent filters, or split computationally intensive filters into multiple parts, or duplicate filters to have more parallel computation.

The StreamIt language is vaguely reminiscent of other imperative languages such as C or Java. In particular, the bodies of filter work functions and stream initialization code is generally legal Java code.

2 Types

This section describes the various data and object types used in StreamIt programs. Data types are passed along tapes between stream objects, and can be used as local variables. Stream objects are only created in initialization code and form a static stream graph.

2.1 Data Types

Data types are always created atomically. They are of fixed size, and are generally statically allocated.

2.1.1 Primitive Types

The following primitive types exist in StreamIt:

`boolean` Either `true` or `false`.

`bit` A one-bit unsigned integer type, containing the value `0` or `1`.

`int` A signed integer type, of unspecified length. This length will typically be the native word length on the target machine; for RAW and x86 compilation, this is usually 32 bits.

Operators	Types	
`?:`	Any	First part must be `boolean`
`||`	`int,bit,boolean`	
`&&`	`int,bit,boolean`	
`|,&,^`	`int,bit`	
`==,!=`	Any	
`<,<=,>,>=`	`int,float`	
`+,-`	`int,float,complex`	
`*,/,%`	`int,float,complex`	`%` must be `int`
`(cast)`	Any	
`!`	`boolean`	
`++,--`	`int`	

Table 1: Operators on primitive types

`float` A floating-point type, of the best precision that will give good performance. On x86 this is a double-precision float, since computation with single-precision floats is implicitly converted to double-precision; on RAW, this is single-precision, since that is all that is supported in hardware.

`complex` A floating-point complex type, of the same precision as `float`. This has real and imaginary parts, which can be directly accessed with `c.real` and `c.imag`.

A number of operations are supported on these primitive types; these are listed in Table 1. In a binary or ternary expression, if two expressions are of different primitive types, they are promoted to the lowest type on the list above. Real expressions converted to `complex` have an imaginary part of zero. `|`, `&`, and `^` are bitwise or, and, and exclusive-or operators, respectively; `!` is a boolean not. The boolean logic operators `&&` and `||` behave as in C: they are short-circuiting, integer values of exactly zero are false, and other values are true. == and other comparison operators return a boolean value.

2.1.2 Structures

Named structures of heterogeneous data types are supported. A structure must be of a fixed size. It contains a set of fields, each of which has a name and a data type. Any primitive or array type, and any structure type which has been previously declared in the program, may be used; recursive structures are not allowed. The following two structure definitions are both legal:

```
struct A {
  int a;
  int[4] b;
}

struct B {
  A a;
  A[4] b;
}
```

A structure definition contains the keyword `struct`, the name of the structure, an open brace, a listing of field (variable) declarations, and a close brace. It is not followed by a semicolon.

The only supported operations on structures are field references and copying. Field references are of the form `a.b`; `a` must be of some structure type `A`, and `b` must be a field named in the structure declaration of `A`. The type of this expression is the declared type of the field. Note that a similar syntax is used for referencing the real and imaginary parts of complex numbers; there is an implicit structure declaration

```
struct complex {
  float real, imag;
}
```

along with some syntactic sugar to make complex arithmetic work.

2.1.3 Arrays

Arrays of any data type described here are supported. Arrays must be of a fixed length. Array types are written with the base type name, followed by a single dimension in square brackets. Multi-dimensional arrays or matrices are supported as arrays of arrays.

2.1.4 Variable Declarations

Variable declarations appear in structure declarations, as well as in code blocks and `filter` declarations. A variable declaration may declare one or multiple variables, possibly with initializers; multiple variables are separated with commas. Of note, the syntax for array declarations is different from C and Java. A correct declaration is `int[4] rgbi;`, with the entire type stated before the variable name. An initializer may not be provided for array variables.

2.2 Stream Types

Computation in StreamIt is performed within stream objects. Every stream object has an input type and an output data type; the object is connected to "tapes" of a hypothetically infinite number of homogeneous data objects. *Filters* are atomic, and have initialization code and steady-state work code; *pipelines*, *split-joins*, and *feedback loops* are all composite structures that include some number of stream objects as children.

All stream objects have initialization code that runs when the object is first created. For filters, this code may be omitted, and generally just sets up filter-local state. For composite objects, the initialization code is responsible for creating the child objects. Filters also have one or more *work functions*, which are called each time the filter executes.

I/O types. Each stream declared at the top level must also explicitly declare its input and output types. This is a declaration like `int->int` or `float->complex`; the first type is the stream's input type, the second its output type. Either type may be `void`; in this case, the stream has no input or output, as appropriate. If it is a filter, it must not declare a peek, pop, or push rate. The top-level stream in a program has type `void->void`; no other streams of this type are allowed.

2.2.1 Stream Parameters

A stream may have a set of parameters provided to it. These parameters are available to all functions within the stream type, and may not be changed. Values for these parameters must be passed in when the stream is created. Stream parameters are declared with types and names in a parenthesized list after the name of the stream; when a stream it instantiated, values for the parameters must follow the name of the stream. For example:

```
float->float filter MatrixMatrixMultiply(int A,
             int B, int C) { ... }
add MatrixMatrixMultiply(3, 5, 4);
```

Note that is is also possible to pass a parameterized-length array as a parameter to a stream, so long as the length is also passed as a parameter. For example, the following is a legal filter declaration:

```
int->int filter FIR (int N, int[N] weights) { ... }
```

Compile-time constants. Several things in StreamIt must be *compile-time constant*; for example, the I/O rates of a filter must be determined at compile-time, and the compiler must be able to statically create the expanded stream graph. An expression is compile-time constant if it is a literal, a stream parameter, or an expression whose components are entirely compile-time constant.

2.2.2 Parameterized Stream Types

Stream types may be *parameterized* on some data type or types, much like "template" types in C++. This may be used for data-reordering operations, and is used for several cases of built-in filters.

Parameterized objects cannot currently be created by the user. To instantiate a built-in parameterized type, use the name of the stream type, followed by the name of the data type in angle brackets, such as `Identity<int>`.

Note: User-instantiated parameterized types are a future StreamIt extension.

2.2.3 Filters

All computation in StreamIt takes place within filters. Filters must explicitly declare their initialization code if present, but also have the option of omitting it. A filter has a single work function. The work function must declare its *I/O rates*, the number of items it removes, examines, or places on its tape. A basic filter declaration looks like this:

```
int->int filter IntAvgFilter {
  init {
    // empty
  }
  work pop 1 peek 2 push 1 {
    push(peek(0) + peek(1));
    pop();
  }
}
```

This filter is named `IntAvgFilter`; it reads integers off of its input tape, and writes integers to its output tape. Each execution of the work function removes exactly one item from the input tape and examines at most two items; it causes exactly one item to be pushed on to the output tape.

Init functions. Init functions are declared using the keyword `init`. They take no parameters. They may initialize filter fields or do other work, but have no access to the filter's tapes.

Work functions. Work functions are declared using the keyword `work`. This keyword may be followed by rate declarations; Rate declarations consist of `push expr`, `peek expr`, and `pop expr`, in any order. If the peek rate is omitted, it is assumed to be equal to the pop rate; if the pop or push rate is omitted, it is assumed to be zero, and the corresponding input or output tape type must be `void`. The function must push and pop exactly the number of items specified, and may peek up to the number of items specified. As a special case, if there are no `pop` calls at all in the function body (all access to the input tape is by peeking), the number of items declared as the pop rate will be automatically popped at the end of the work function. All of the rate expressions must be compile-time constants.

Helper functions. A helper function performs some auxiliary bit of computation, and may be called from the init function or the work function. A helper function has a list of zero or more parameters, and returns zero or one values. A helper function declaration contains the return type, the name of the function, the parameter list, and the function body, just like a normal C or Java function.

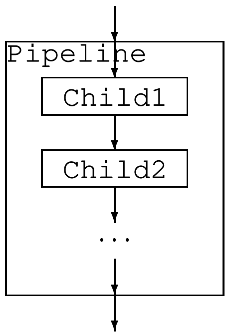
Filter state. The top level of the filter may include variable declarations. These variables are visible to every function in the filter, and may be used to carry state between iterations of the work function.

2.2.4 Pipelines

The simplest composite filter is a pipeline. A pipeline contains a number of child streams; the output of the first stream is connected to the input of the second, whose output is connected to the input of the third, and so on. A pipeline declaration looks like

```
float->complex pipeline FloatToComplexAdd {
  add ReImToComplex();
  add ComplexPairAdd();
}
```

The body of the pipeline declaration is the initialization code; no internal function declarations are allowed. In addition to the internal typing constraints mentioned previously, the input type of the pipeline must match the input type of the first filter, and the output type of the pipeline must match the output type of the last filter.




```
float->float
pipeline Pipeline {
 add Child1();

 add Child2();

 ...
}
```

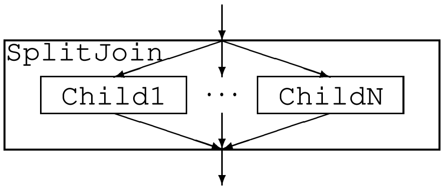
Figure 1: Pipelines

2.2.5 Split-Joins

A split-join allows computation to be run in parallel, possibly using different parts of the input stream. It is so named because incoming data passes through a *splitter*, is redistributed to the child streams for processing, and then is fed through a *joiner* to be recombined into a single output stream; see section 4.3.

```
int->int splitjoin AddAndSub {
  split duplicate;
  add IntAdder();
  add IntSubtractor();
  join roundrobin;
}
```

This split-join has two children. Incoming data is duplicated on to both streams. The first branch consumes two elements and adds them; the second consumes two elements and subtracts them. The resulting values are then pushed on to the output stream in a round-robin fashion, with values from alternating streams.




```
int->int splitjoin SplitJoin {
 split roundrobin(2);
 add Child1(); ... add ChildN();
 join roundrobin;
}
```

Figure 2: Split-Joins

2.2.6 Feedback Loops

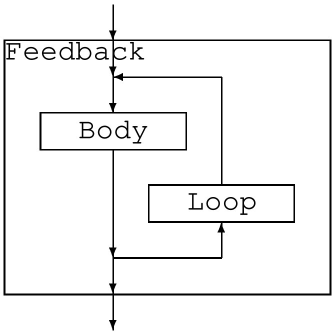
A feedback loop has a body stream. Its output passes through a splitter; one branch of the splitter leaves the loop, and the other goes to the loop stream. The output of the loop stream and the loop input then go through a joiner to the body's input.

```
float->float feedbackloop AddFeedback(float scaling) {
  join roundrobin;
  body FloatAdder();
  loop FloatScaler(scaling);
  split duplicate;
  enqueue 0.0;
}
```

The `body` and `loop` declarations may be omitted; if so, an implicit `Identity` filter of the appropriate type is inserted.

Semantics of splitters and joiners. Splitters and joiners both treat the stream input as the first child, and the loop edge as the second child. Splitters and joiners that do not allow exactly two children are disallowed.

`enqueue` statement. The `enqueue` statement takes a value and places it in FIFO order on the output tape from the loop stream. You will generally need to enqueue enough items that the input joiner will be able to fire once; enqueuing more items causes a delay in the feedback loop.




```
float->float
feedbackloop Feedback {
 join roundrobin;
 body Body();

 loop Loop();
 split roundrobin(4,1);
 enqueue(0.0);
}
```

Figure 3: Feedback Loop

2.2.7 Anonymous Streams

It is possible to declare a stream object without a name, if it is used in exactly one place in the program. In general anonymous stream declarations look exactly like the corresponding normal stream declaration, except that the stream type is optional for non-filter streams and the name is omitted. The semicolon that normally ends the `add`, `body`, or `loop` statement is optional in this case.

```
complex->float feedbackloop FeedbackMagnitude
    (float scaling) {
  join roundrobin;
  body complex->float filter {
    work pop 2 push 1 {
      push(abs(peek(0) * peek(1)));
      pop(); pop();
    }
  };
  loop pipeline {
    add float->float filter {
      work pop 1 push 1 { push(pop() * scaling); }
    };
    add FloatToComplex();
  };
  split duplicate;
}
```

The input and output types of anonymous streams are determined by the compiler, if these types are not explicitly specified in the code.

Anonymous streams do not have stream parameters. In certain cases, though, code within anonymous streams can access variables and stream parameters from the containing stream. Values so referenced must be compile-time constant. Fields of the containing stream cannot be accessed, and local variables and stream parameters in the containing code cannot be modified.

2.3 Execution Model

StreamIt does not explicitly assume sequential or parallel hardware. A component of the stream graph may *fire*, possibly in parallel with other firings, if its conditions to execute are met. The scheduler in the compiler chooses an order to execute filters such that the firing conditions are met. A particular back-end may place additional restrictions on firings; a uniprocessor back-end may require that only one stream object executes at a time, or a parallel back-end might add the constraint that two objects may not both be executing if the output of one is an input of another. These constraints are transparent to the programmer, however.

- A filter may fire when its input has at least as many items as its peek rate. It pops its pop rate from the input, and pushes its push rate on to its output.

- A joiner in a feedback loop or split-join may fire when the specified number of items are available on each of its inputs. It pushes the sum of the input weights on to its output.

- A round-robin splitter may fire when the sum of the number of output items is available on its input, and produces the specified number of items on each output.

- A duplicate splitter may fire when at least one item is available on its input. It pops one item from its input and pushes one item to each output.

3 Statements

This section describes statements legal in StreamIt code. For the most part, these are identical to legal statements in C or Java. Some statements, particularly those that set up the stream graph, are only legal in initialization code; other statements are legal anywhere.

3.1 Initialization-Only Statements

3.1.1 `add` Statement

The `add` statement adds a child stream to the current stream object. It is valid in split-joins and pipelines only. In a split-join, it adds a new stream to the end of the current list of children. In a pipeline, it adds a new stream to the end of the chain of children.

This statement takes a *stream constructor*; see section 4.2.

3.1.2 `body` Statement

The `body` statement adds a child stream as the body part of a feedback loop. This statement must be executed at most once in the initialization code of a feedback loop, and is invalid anywhere else. If a feedback loop does not contain a `body` statement, an `Identity` filter of the appropriate type is used by default.

This statement takes a *stream constructor*; see section 4.2.

3.1.3 `enqueue` Statement

The `enqueue` statement is valid only within the initialization code of a feedback loop. It causes the specified item to be placed after all other previously enqueued items on the input to the feedback loop joiner. This statement takes a single value expression.

3.1.4 `join` Statement

The `join` statement declares the joiner type of a feedback loop or split-join. It only appears in initialization code, and is invalid in initialization code for any other construct. This statement takes a joiner specification; the only one valid currently is `roundrobin`, see section 4.3.2.

3.1.5 `loop` Statement

The `loop` statement adds a child stream as the loop part of a feedback loop. This statement must be executed at most once in the initialization code of a feedback loop, and is invalid anywhere else. If a feedback loop does not contain a `loop` statement, an `Identity` filter of the appropriate type is used by default.

This statement takes a *stream constructor*; see section 4.2.

3.1.6 `split` Statement

The `split` statement declares the splitter type of a feedback loop or split-join. It only appears in initialization code, and is invalid in initialization code for any other construct. This statement takes a splitter specification; only `duplicate` and `roundrobin` are currently legal, see sections 4.3.1 and 4.3.2.

3.2 Work-only Statements

These statements may only appear within the `work` function.

3.2.1 `push` Statement

The `push` statement takes a value expression and pushes its value on to the output tape of the current filter. The type of the expression must match the output type of the filter.

3.3 General-use Statements

These statements may appear anywhere code is legal.

3.3.1 Statements with Java-like Semantics

`do`, `for`, and `while` loops have the same syntax and semantics they do in Java. The `continue` and `break` statements are also recognized, though they may not take a label reference.

For statements whose syntax match exactly

```
for (i = S; i < N; i++)
  /* body */ ;
```

cause the induction variable `i` to be compile-time constant within the loop if the start and end indices `S` and `N` are also compile-time constant.

3.3.2 Assignment Statements

Assignment statements have the form `lhs = expr;`. The left-hand side may be a variable, a field reference, or an array element reference. Each of these is evaluated to a location, with the result being either a primitive value, a structure, or an entire array. If it is a primitive value, the right-hand side must be a value expression. Otherwise, the right-hand side must be an array or structure of identical type, and causes all of the elements from the right-hand-side object to be copied to the left-hand-side object.

3.3.3 Variable Declarations

Variable declarations have the form `type name;`. `type` must be a data type; `name` may be any legal name not corresponding to another variable previously declared in the current block. A declaration must appear before the variable is used. The variable declaration may also contain an initialization, as in `type name = expr;` The initialization acts exactly as an initialization statement, above.

Variables declared without initialization are initialized to 0 as best as possible. `complex` variables are initialized to `0+0i`. Each element of an array variable is initialized to 0 as described here. Similarly, each field of a structure variable is initialized to 0 in the same way.

4 Expressions

4.1 Value Expressions

A *value expression* carries some value of a data type. Literal values are value expressions; these can be signed integer or real values, e.g. `17`, `2.45`, or `1.4e6`. A literal value can also be a pure imaginary number, e.g. `17i` or `3.4i`. Unary, binary, and ternary expressions of value expressions are also value expressions, as described in section 2.1.1. Thus, while `3+4i` is not a literal, it is a valid value expression of complex type.

Field references to structure objects are also value expressions. These have the form `a.b`, where `a` is a variable of structured type `A` and `b` is a field in `A`. Similarly, array references `a[b]` are value expressions; `a` must be an array variable, and `b` must be an integer-valued value expression.

4.1.1 peek and pop

The peek and pop expressions look at items on the incoming tape in a filter's work function; they are illegal in any other context. `peek(n)` examines the n-th item on the tape without removing it; `peek(0)` returns the next incoming item, `peek(1)` the one afterwards, and so on. `pop()` returns the same item `peek(0)` returns, but also removes it from the tape.

Note: There is not a guaranteed evaluation order for StreamIt expressions, and the implementation of `pop()` in C doesn't give any guarantees for when the item is actually removed from the tape. Thus, an expression should contain no more than one `pop()`, and if there is one, there should be no `peek()`s.

Note that `pop()` removes an item from the tape, so `peek()`s afterwards use different indices. A work function must pop a fixed number of items, and declare the number of items in its rate declaration. It may not peek at more items than it declares in its rate declaration, though it may peek at fewer. The peek declaration declares the furthest position on the tape examined from the point of view of the beginning of the function.

```
int->int filter PeekRates {
  work pop 1 peek 3 push 1 {
    int a, b, c, d, e;
    a = peek(0);
    b = peek(1);
    c = pop(); // Same value as a
    d = peek(0); // Same value as b
    e = peek(1);
    push(a + b + c + d + e);
  }
}
```

This function pops one item and pushes 1. It declares a peek rate of 3 because `e` peeks at the third item that was on the tape at the start of the function – one is popped, and `e` contains the second item beyond that.

4.1.2 Function Calls

Function calls are value expressions; certain cases of built-in functions may return values that are compile-time constant if their parameters are compile-time constant. The function call syntax `fn(p1, p2)` is identical to that of Java. All parameters are always passed by value; this differs from Java's semantics, in which objects and arrays are passed by reference.

Function calls may be made to declared helper functions, or to built-in math functions. The math functions are listed in table 2.

Note: As an implementation detail, any function name not optimized as a built-in function in the front-end or recognized as a helper function is treated as a call to a function in the Java package `java.lang.Math`; see *Java in a Nutshell* or another Java reference for details. These get blindly translated again to calls to the C math library.

Note: It is conceivable that calls to side-effect-free functions with compile-time constant parameters return compile-time constant results. This is only the case for built-in functions in the current implementation of the compiler; adding this involves inlining helper functions, and doing analysis to determine if a helper is in fact side-effect free.

`abs(v)`	Real absolute value or complex magnitude
`arg(v)`	Polar angle of complex value
`exp(v)`	Real or complex exponent
`log(v)`	Real or complex natural logarithm
`sin(v)`	Real or complex sine
`cos(v)`	Real or complex cosine
`sqrt(v)`	Real or complex square root
`csqrt(v)`	Complex square root

Table 2: StreamIt Math Functions

4.1.3 Typecasts

If a primitive type in parentheses precedes an expression, that expression is interpreted as being of the particular type. See the listing of types in Section 2.1.1. Conversions to types later in the list take place in the obvious way: `boolean` values convert to `bit` values with 0 for `false` and 1 for `true`, `bit` values convert to `int` values with the same value, and so on. `float` values convert to `complex` values with an imaginary part of 0. Conversions to types earlier in the list may lose data; conversions from `complex` are not possible, `bit` converts to `boolean` in the obvious way. The exact conversion from `float` to `int` is not specified. `int` values of 0 convert to `bit` values of 0, and other `int` values convert to 1.

```
/* For each input, pushes (float) 1 if the integer representation
 * of the input has the 64-bit set. */
float->float filter TypeCast {
  work pop 1 push 1 {
    complex c = (complex)pop();
    float f = abs(c); // can't directly cast from complex
    int i = (int)f;
    bit b = (bit)(i & 64);
    push((float)b);
  }
}
```

4.2 Stream Constructors

A *stream constructor* indicates that a new stream object should be created. Stream constructors appear in `add`, `body`, and `loop` statements in composite stream initialization code.

Stream constructors may reference an already-declared stream type. In this case, the stream constructor form is `Name(param, ...)`. There must be exactly as many parameters in the constructor as parameters for the stream type. The parameters must all be compile-time constant.

An anonymous stream declaration is also a valid stream constructor; see section 2.2.7. A statement containing an anonymous stream declaration does not need to be terminated with a semicolon, since it ends with a close-brace.

A stream constructor may be followed by the keyword `to` and the name of a portal variable. This causes the newly created stream to be registered with the named portal.

4.3 Splitters and Joiners

Splitters have a single input of a given type and multiple outputs of the same type; joiners have multiple inputs and a single output. They appear in feedback loops and split-joins.

Note: Other types of splitters and joiners besides duplicate and round-robin are conceivable; in particular, structure and array packing and unpacking splitters and joiners are obvious. There also may be demand for "custom" splitters and joiners, where a data-reordering operation is described by the programmer.

4.3.1 `duplicate` Splitters

`duplicate` splitters take each incoming item and push the same item to each of the outgoing tapes, duplicating data.

4.3.2 `roundrobin` Splitters and Joiners

`roundrobin` splitters take each item and send it to exactly one of the child streams, in order. Either `roundrobin` or `roundrobin()` causes one item to be sent to each output, in order; `roundrobin(2)` causes two items to be sent to the first stream, two to the second, and so on. `roundrobin(2, 4, 2)` requires there to be exactly three children, and sends two to the first child, four to the second, and two to the third. A round-robin weight may be 0; in this case, the child input must be of type `void`. A `roundrobin(0)` splitter on a split-join means that no children take any input, and the input type of the entire split-join is `void`.

`roundrobin` joiners are identical to `roundrobin` splitters, except that they read from the input tapes in the specified pattern and write data to the output tape.

5 Built-in Types

This section describes built in data and stream types in StreamIt. These types are implicitly declared; code can just use these types without explicitly importing or otherwise declaring them.

5.1 Built-in Data Types

The `complex` type is actually implemented as a structure, though the compiler implements syntactic sugar for complex variables. The structure definition is shown above in section 2.1.2. It is legal to explicitly reference the `real` and `imag` fields of complex numbers as though they were the structure type instead.

5.2 Built-in Streams

Several stream types are in common use, and are either simple enough or require non-standard language functionality that the compiler should support them directly. These include parameterized `Identity`, `FileReader`, and `FileWriter` types, as shown in figure 4.

```
template<T> T->T filter Identity {
  work pop 1 push 1 { push(pop()); }
}
template<T> void->T filter FileReader
    (string filename) {
  work push 1 { /* Implementation defined */ }
}
template<T> T->void filter FileWriter
    (string filename) {
  work pop 1 { /* Implementation defined */ }
}
```

Figure 4: Built-in Stream Types

6 Future Additions

This section describes features that are likely to be present in a future release of the language. In particular, parts of the compiler already have full support for pre-work functions, phased filters, and messaging.

6.1 Prework Functions

A filter may specify an additional work function that is run in place of the normal work function the first time the filter executes. This function uses the `prework` keyword, and is otherwise identical to the normal work function.

Prework functions can be very useful for setting up initial conditions based on the first set of inputs, or for pushing extra items on the first invocation. For example, consider the following implementation of a Delay:

```
float->float filter Delay(int N) {
  prework push N {
    for (int i=0; i<N; i++) {
      push(0);
    }
  }

  work push 1 pop 1{
    push(pop());
  }
}
```

6.2 Phased Filters

If the work function does not have rate declarations, it may not perform `push` and `pop` operations directly. However, in a phased filter, it may call *phase functions*, which are declared like work functions but begin with the keyword `phase` and a name, and always have rate declarations. The runtime system ensures that, when a phase function is called, its `peek` and `pop` rates are satisfiable, even if other filters need to run for later phases to execute. Beyond this special use, phase functions may not be directly called. For example:

```
int->int filter BlockReader {
  // Each block has a 4-int header, 24 ints of data,
  // and a 4-int footer.
  work {
    readHeader();
    for (int i = 0; i < 24; i++)
      echoData();
    readFooter();
  }
  phase readHeader pop 4
     { pop(); pop(); pop(); pop(); }
  phase echoData pop 1 push 1 { push(pop()); }
  phase readFooter pop 4
     { pop(); pop(); pop(); pop(); }
}
```

6.3 Messaging

It is sometimes necessary to send occasional messages outside of the normal stream data flow. A radio application might contain an in-band signal requesting a change in the listening frequency, for example; this signal would happen infrequently, but it would be detected by a filter late in the application and require a change in a filter early in the application. StreamIt provides an out-of-band messaging system to accomodate this sort of application.

Only filters can receive messages. The possible messages are defined by message handler functions; these functions have the keyword `handler` in place of their return type. A message is sent to a *portal*, which can have multiple targets registered with it.

Portals. A portal is a special type of variable. It is associated with a single message receiver type; the full type name is `portal<ReceiverType>`. Unlike other variables in StreamIt, portals are passed by reference in stream parameters. Declaring a portal as a local variable, and then passing that as a parameter to a child stream which adds a filter to the portal, results in the filter also being added to the portal in the original container. When a filter is created, it may be added to a portal by adding the keyword `to` and the portal variable to the end of the declaration.

Sending messages. A filter work function with a portal variable may send a message through the portal. This looks like a function call in Java: the portal name is followed by a period, the message name, and a parenthesized parameter list. This may, optionally, be followed by a message latency, which specifies the minimum and maximum delay to receive the message. The latency specification is an open bracket, an optional minimum latency, a colon, an optional maximum latency, and a close bracket. If the latency specification is omitted, it defaults to `[:]`, allowing any latency.

When specified, message latency is stated in terms of number of items pushed or popped by the sending filter. If a message is sent with latency `[10:12]`, it will be received by upstream targets immediately before a work function execution that would generate an item that would affect the tenth through twelfth next data item that would be received by the sender; downstream filters would receive the message after a work function execution that processes data derived from the tenth through twelfth pushed items.

A typical message-handling application might use a message to change an amplification factor. The following code demonstrates such an application. The `Volume` pipeline creates the portal, passes it as a parameter to the detector, and adds the actual amplifier to it.

```
float->float filter VolumeControl {
  float amplification;
  init { amplification = 1.0; }
  work pop 1 push 1 { push(pop() * amplification); }
  handler setAmplification(float val)
    { amplification = val; }
}

float->float filter VolDetect
    (portal<VolumeControl> vol) {
```

```
  work pop 1 push 1 {
    float val = pop();
    if (/* detected */)
      vol.setAmplification(0.5) [10:12];
    push(val);
  }
}

float->float pipeline Volume {
  portal<VolumeControl> vol;
  add VolDetect(vol);
  add VolumeControl() to vol;
}
```

6.4 The keep Construct

The `keep` function will look at, but not remove, items on the output tape of a filter. `keep(0)` returns the item that was most recently pushed, even if that was on a previous filter iteration. Just as `pop` affects indices for future `peek` calls, `push` adds another item to the output tape and hence changes the indices for future `keep` calls.

`keep` can be used to implement IIR filters. For example, given this LTI filter:

$$\begin{aligned} H(z) &= \frac{1+z^{-1}}{1-z^{-1}} \\ (1-z^{-1})Y(z) &= (1+z^{-1})X(z) \\ y[n]-y[n-1] &= x[n]+x[n-1] \\ y[n] &= x[n]+x[n-1]+y[n-1] \end{aligned}$$

We can write equivalent StreamIt code:

```
float->float filter IIRExample {
  prework push 1 {
    push(0);
  }
  work push 1 pop 1 peek 2 keep 1 {
    // y[n] = x[n] + x[n-1] + y[n-1]
```

```
    push(peek(0) + peek(1) + keep(0));
    pop();
  }
}
```

Of note, since the work function can not fire unless there is at least one item on its output tape, there must be a prework function that pushes that item; otherwise, the program is unschedulable.

6.5 Helper Packages

In many cases the programmer would benefit from writing a set of stateless helper functions that can be used in multiple filters. For example, the GSM encoder we have now uses the C preprocessor to include a set of functions to perform 16-bit arithmetic. Syntax for this functionality does not yet exist, though we may create a `static` keyword:

```
static MyHelpers {
  int square(int x) { return x * x; }
}
int->int filter SquareIt {
  work pop 1 push 1 { push(MyHelpers.square(pop())); }
}
```

7 Implementation Limits

The following limits apply to the current implementation of StreamIt:

- There is no guaranteed order of evaluation. In particular, a statement in a work function should not contain more than a single `pop()`, and if contains any `pop()`s at all, it should contain no `peek()`s.
- The names `SplitJoin`, `FeedbackLoop`, `Filter`, `Pipeline`, `StreamIt`, and `Complex` are reserved. Certain variable names beginning with underscores may cause conflicts if used as local variables.

8 History

8.1 StreamIt 2.0

The first formal language specification for StreamIt documented StreamIt 2.0. Version 2.0 of the language adds a completely new syntax with simplified constructs for declaring stream objects but keeping a Java-like syntax for the bodies of init and work functions. It also changes from delay functions to `enqueue` statements in feedback loops. The `null` splitter/joiner type was replaced with `roundrobin(0)`.

8.2 StreamIt 1.0

The initial StreamIt compiler was based on a restricted subset of Java, dubbed StreamIt 1.0. This version of the language used pure Java syntax for stream constructs, and was compilable with a normal Java compiler given an appropriate run-time library. Early publications referred to the language as StreaMIT, but this capitalization is now deprecated.