StreamIt Semantic Checks

January 29, 2007

This document contains a list of semantic checks that should be performed on StreamIt programs after they have been parsed. These assume the program is syntactically correct; it must pass through the parser correctly.

1 Statement Placement

1.1 Init, Work, and Helper Functions

1. `add` statements can only appear in init functions in split-joins and pipelines;
2. `split` and `join` statements can only appear in init functions in split-joins and feedback loops;
3. `loop`, `body`, and `enqueue` statements can only appear in init functions in feedback loops;
4. `push` statements can only appear in work and helper functions in filters.
5. `pop` and `peek` expressions can only appear in work and helper functions in filters.
6. If a work function does not declare I/O rates, it may not use `push`, `pop`, or `peek`. If it declares a push rate, though, it must call `push`.[1]
7. Every filter must have a work function.

[1] Declaring a nonzero peek rate affects the scheduling but not correctness, so you're not obligated to peek; however, you are obligated to pop the right number of items.

8. Every stream object that is not a filter must have an init function, but may have no other functions.[2]

1.2 Control Flow and Initialization Code

Several statements may only be executed once per function; other statements are sensitive to the order in which they are executed.

1. `split`, `join`, `body`, and `loop` statements must be executed exactly once if they are allowed at all. Every path of control through the initialization code must result in exactly one execution.

2 Typing

2.1 Stream Connections

"Match" within this section means an *exact* match in type; there is no implicit type promotion across tapes.

1. For a pipeline, the input type must match the input type of the first child and the output type of the last child must match the output type of the pipeline.

2. Within a pipeline, the output type of one child must match the input type of the next child.

3. The input type of each of a split-join's children must be either the input type of the split-join or `void`. Identical constraints hold for the output.

4. The output type of a feedback loop's loop stream must match the input type of its body stream. The input type of the feedback loop must be either the same type as the body input or `void`. Identical constraints hold for the output side of the feedback loop.

5. Exactly one stream in the program has the stream type `void->void`. This stream must be named.

[2] This is currently implicit in the language syntax. The constraint and syntax could both change due to reinitialization, or by adding message handlers to composite streams.

2.2 Implications of Stream Types

1. The splitter of a split-join must be `roundrobin(0)` if the input type of the stream is `void`; it may not be `roundrobin(0)` otherwise. Identical constraints hold for the output.

2. If a split-join has some children with `void` inputs and some children with non-`void` inputs, the splitter must be a round-robin splitter with a declared weight of 0 for each of the `void` children. Identical constraints hold for the output and the joiner.

3. The splitter and joiner of a feedback loop must accomodate exactly two children. If the body and loop children have non-`void` input and output types, then the joiner must be equivalent to `roundrobin(0,1)` if the input of the loop is `void`, and similarly for the splitter and the output.

4. If the input type of a feedback loop's body child is `void`, then the loop input type and loop child output type must also be `void`, and the joiner must be `roundrobin(0)`. Identical constraints hold for the loop output and splitter.

5. If the input type of a filter is `void`, all of its work functions must declare peek and pop rates of 0; if its output type is `void`, all of its work functions must have a push rate of 0.

2.3 Statements and Expressions

1. Types can be promoted: a `bit` can be implicitly converted to an `int`, an `int` can be promoted to a `float`, and a `float` can be promoted to `complex`.

2. Every referenced variable must have a declared type.

3. Every expression has a type:

   (a) The type of a variable reference is the type of the variable;

   (b) The type of a literal is the lowest type that can contain the literal (the type of `1` is `bit`, the type of `2` is `int`, the type of `2.0` is `float`, and the type of `2i` is `complex`);

   (c) The type of a unary expression is the type of the child expression. The type of a binary expression is the join of the types of the

child expression, possibly promoted to a type the operator can accept. The type of the first part of the ternary expression must be promotable to `int`, and the second and third parts must be promotable to a common type; the resulting type is the common type. In all cases, the types of child expressions must conform to the acceptable types for the operator listed in the language specification.

(d) For a field reference `a.b`, `a` must be of some structure type `A`, which must have a field named `b`. The type of the expression is the type of `b` in the structure declaration.

(e) For an array access `a[i]`, `a` must be of some array type `T[n]`, and the type of the expression is `T`.

(f) The type of a `peek()` or `pop()` expression is the input type of the filter.

(g) The parameter of a `peek()` expression and the index of an array must both be of type `int`.

(h) The type of a call to a helper function is the return type of that function.

4. The type of the parameter of a `push()` statement must be the output type of the filter.

5. The type of the parameter of an `enqueue()` statement must be the output type of the feedback loop's loop stream.

6. The type of the right-hand side of an assignment statement must be promotable to the type of the left-hand side of the statement.

2.4 Using Names As Types

1. A bare name may be used as a type in a variable declaration, parameter list, stream type, and elsewhere. The name must match the name of a declared structure type.

2. A bare name may be used as the target type of a `Portal` type. The name must match the name of a declared stream type; the target of a portal may not be a primitive type or structure.

2.5 Messaging

1. A portal type must name a top-level stream structure containing at least one message handler.
2. A message-sending statement must call a function present in the receiver type of its portal.
3. The types of the parameters of a message-sending statement must be promotable to the types of the parameters in the receiving function.
4. Message latencies must be of integer type.
5. When a stream structure is added to a portal, the type of the stream structure must be the same as the receiver type of the portal.

3 Naming

3.1 Object Names

1. No two streams or structures may have the same name.
2. Within a single structure, no two member fields may have the same name. Field names must also be distinct from all stream and structure names.
3. Stream parameters within a single stream must have distinct names from each other and from all stream and structure names.
4. Functions declared within a filter must have unique names; these include message handlers and helper functions. The name of a function must be distinct from stream and structure names.
5. A variable may not be declared with the same name as a stream parameter. (Other variable hiding is legal.) The name of a variable must be distinct from stream, structure, and function names within the current filter.

3.2 Using Names

1. A stream constructor (the parameter to `add`, `body`, and `loop` statements) must be an anonymous stream declaration, or must have a name matching the name of a stream object with a parameter list matching the parameter list of the the stream.

2. A variable must be declared before it is used.

3. A stream parameter may not be assigned to or otherwise modified.

4. A function call must have a name matching a helper function in the current filter, and a parameter list matching the parameter list of the target function.

5. A message-sending statement must name a portal variable on the left-hand side of the ".", and a message name on the right-hand side. The message name must match a message in the target type of the portal, and the statement must have a parameter list matching the parameter list of the message target.

4 Warnings

Things in this section are technically legal, but indicate questionable code:

1. Some things in StreamIt are counted; these include split-join children (with fixed-length weighted round-robin splitters or joiners), and work function pushes, pops, and peeks. It should be a warning if the compiler cannot statically confirm that these numbers are not met exactly (or, for peeks, that the peek rate is not exceeded). For feedback loops, the compiler can also issue a warning if it cannot confirm that enough items are enqueued to cause the joiner to be fired at the start of the program.

2. The compiler may need to guess at the I/O types of anonymous streams. If the type isn't clear from context a warning should be issued.

3. A warning may be issued if the compiler cannot verify that splitters and joiners for split-joins can accomodate the exact number of children present. Note that this may only be possible with pattern-matching program text, or by fully unrolling init functions, since the following code is legitimate but data-flow wouldn't find an exact number of children:

```
float->float splitjoin FiveChildren {
  split roundrobin(1,2,3,4,5);
  for (int i = 0; i < 5; i++)
    add float->float filter { ... };
  join roundrobin(5,4,3,2,1);
```

```
}
```

5 Temporary Checks

These checks are necessary due to current constraints in the StreamIt compiler. They should be able to be removed when the compiler is improved.

1. All child-constructing statements in initialization code must have consistent types. While the front-end can perform minimal data-flow analysis to check that the types of pipelines, control flow like the following is presently illegal:

   ```
   int->int filter IntIntBody { ... }
   int->int filter IntIntLoop { ... }
   float->int filter FloatIntBody { ... }
   int->float filter IntFloatLoop { ... }

   void->int feedbackloop SketchyLoop(int p) {
     join roundrobin(0, 1);
     if (p) {
       body IntIntBody();
       loop IntIntLoop();
     } else {
       body FloatIntBody();
       loop IntFloatLoop();
     }
     split duplicate;
     enqueue(1);
   }
   ```

   The types of the loop and all of its components are consistent regardless of the value of `p`, but if `p` is zero, the `enqueue` statement takes a `float` parameter. Presently the front-end should reject this construction since it is unclear if the types around the joiner are consistent or not. This check can be removed if the compiler does stream type-checking after constant propagation.

2. None of the names listed in the "Implementation Limits" section of the language specification should be used. These include the capitalized names `SplitJoin`, `FeedbackLoop`, `Filter`, `Pipeline`, `StreamIt`, and `Complex`.