DirectFlow: a Domain-Specific Language for Information-Flow Systems

Chuan-kai Lin and Andrew P. Black

Department of Computer Science
Portland State University
{cklin,black}@cs.pdx.edu

Abstract. Programs that process streams of information are commonly built by assembling reusable *information-flow components*. In some systems the components must be chosen from a pre-defined set of primitives; in others the programmer can create new custom components using a general-purpose programming language.
Neither approach is ideal: restricting programmers to a set of primitive components limits the expressivity of the system, while allowing programmers to define new components in a general-purpose language makes it difficult or impossible to reason about the composite system. We advocate defining information-flow components in a domain-specific language (DSL) that enables us to infer the properties of the components and of the composed system; this provides us with a good compromise between analysability and expressivity.
This paper presents DirectFlow, which comprises a DSL, a compiler and a runtime system. The language allows programmers to define objects that implement information-flow components without specifying how messages are sent and received. The compiler generates Java classes by inferring the message sends and methods, while the run-time library constructs information-flow networks by composition of DSL-defined components with standard components.

1 Introduction

Systems that stream information continuously from source to sink are commonplace; examples include software routers for network traffic [1], data stream query systems [2], surveillance systems, real-rate video streaming [3], and highway loop detector data analysis [4]. Hart and Martinez survey more than 50 current examples of environmental sensor networks, and argue that the ability to construct systems that stream environmental data continuously from the field to the scientists' laboratories will revolutionize earth system science [5]. We refer to this wide class of systems as information flow applications.

A popular strategy for building information-flow applications is by composing reusable components. Systems based on this strategy include Aurora [2], the Click modular router [1], Krasic's media streaming system [3], StreamIt [6],

Spidle [7], and our own Infopipes system [4, 8]. In these systems, each component has a set of input ports (inports) and output ports (outports), which connect to information flows. Since each application has distinct data-processing requirements, many information-flow programming systems allow programmers to define custom components.

In component-based systems, programmers construct an information-flow application by connecting the ports of components with channels, which specify the flow of data packets. In an object-oriented program, message sends directly determine how the thread of control passes between objects. In contrast, a channel connecting two ports determines only how data packets flow: it says nothing (directly) about how the thread passes between components. Control may pass from upstream to downstream or the other way around: the choice depends on not only the nature of the connected components, but also the context in which the components appear. These dependencies make managing the control flow of an information-flow application a difficult and labor-intensive task: the relationship between a component's data-flow behaviour and its control-flow interaction with other components is understood only for linear components [9]. Moreover, the context-sensitivity of components means that a local code change may have a great influence on the global control-flow.

Manually managing the control flow distracts programmers from organizing the data flow in an application, and a practical information-flow programming system should relieve programmers of this burden with the following feature:

F0. *Automatic invocation of components.* The system should handle the invocation of components in response to runtime data flow. The programmer should not have to specify how a component invokes, or is invoked by, neighboring components.

In addition to this software-engineering feature, we will argue that a practical information-flow programming system should also support the following three *information-flow* features:

F1. *Expressive custom components.* The system should allow the creation of custom components with multiple inports and outputs that can add packets to or remove packets from a flow.

F2. *Control over data-transfer latency between components.* The system should connect components with unbuffered channels to avoid introducing arbitrary latency between components.

F3. *Choice of processing mode.* Two different modes of processing are possible: data-driven processing invokes a component when a data packet arrives at one of its inports, while demand-driven processing invokes a component when a data request arrives at one of its outports. The system should allow both modes of processing to coexist in one application.

Unfortunately, no previous information flow systems support all four features F0–F3 because this rich combination of features can allow programmers to connect components in a way that has no reasonable implementation.

In this article we describe DirectFlow, which does support all four features. How do we achieve this? DirectFlow allows programmers to define custom components using a domain-specific language (DSL). The DirectFlow compiler checks the DSL modules to see if they satisfy a context condition (discussed in Sect. 3.2). If they do, the compiler generates one or more Java classes for each module. The DirectFlow runtime system allows programmers to compose these custom components with standard library components; further consistency checks are applied at composition time. The effect of the checks is to eliminate unimplementable pipelines. The composed pipeline is then instantiated in a set-up phase, and can finally be used in an ordinary Java program to perform information-flow processing tasks.

In Sect. 2 we fill-in some background about the four features listed above and explain why it is hard to support them all. The DirectFlow domain-specific language is described in Sect. 3. Section 4 describes the way that we ascertain the possible processing modes for the components defined in the DSL, and how we generate Java code. Section 5 discusses the DirectFlow Framework, which brings together library code, generated Java code, and hand-crafted Java code into deployable pipelines.

The specific technical contributions of this work are as follows:

- A CSP-inspired programming model that supports both data-driven and demand-driven data processing in a uniform manner.
- A general characterization of how the thread of control traverses through ports of information-flow components connected by unbuffered data-driven and demand-driven channels.
- A control-flow analysis algorithm that infers automatically how the thread of control enters and exits an information-flow component.
- The DirectFlow DSL based on our programming model.
- A compiler that generates several different Java implementations of a component from the DirectFlow definition, one for each possible flow of control.
- A run-time system that allows programmers to compose DirectFlow components, while ensuring that they are composed consistently.

We also show that the ideas behind DirectFlow lead to an alternative formulation of objects that does not involve methods.

2 Background

The information-flow features F1–F3 are important because of situations that naturally arise in information-flow applications. We illustrate these situations by example; here and throughout this paper we will refer to the custom components depicted in Fig. 1.

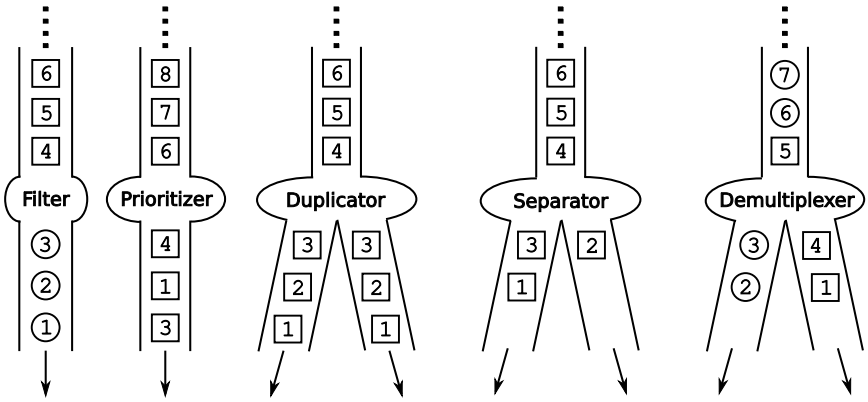

Fig. 1. Five example components

A ***filter*** transforms individual packets in a flow. For example, a filter can decompress an MPEG video block into an 8×8 block of pixels.

A ***prioritizer*** buffers packets in a flow. On request, it outputs the packet with the highest priority. The output thus depends not only on the content of the flow but also on the timing of the input and output events. Krasic uses such a component to provide high-quality video playback over an unstable network connection [3].

A ***duplicator*** sends each packet in the input flow to two output flows. A programmer might use it to copy a stream for logging.

A ***separator*** sends each packet in the input flow to one of two output flows. The chosen flow is the one that is ready to receive the packet first. A separator allows us to connect a "ticket dispenser" component that produces unique numbers to multiple clients.

A ***demultiplexer*** sends each packet in the input flow to one of two output flows depending on the packet's contents. For example, a demultiplexer can extract video and audio streams from an MPEG system stream.

2.1 Why the Information-Flow Features are Important

If an information-flow system is to allow programmers to define custom components like these examples, it must exhibit features F1–F3. Without F1 (expressivity), there is no way to implement a demultiplexer because it has two outports with different data rates. Without F2 (latency-control), the system is no longer compositional: splitting a component into two components can introduce arbitrary latency [8, §2.1]. Moreover, buffered channels change the behaviour of the prioritizer because they alters the timing of input and output operations and thus interfere with the priority-reordering.

The result of sacrificing F3 (choice of processing mode) depends on the specifics of the system. If the system supports only data-driven processing, there

is no way to build a separator without buffering or blocking. If the system supports only demand-driven processing, there is no way to use a duplicator or a demultiplexer without buffering or blocking.

Full support of F3 permits not only pure data-driven and demand-driven components but also components that are partially data-driven and partially demand-driven. For example, the prioritizer has a data-driven inport and a demand-driven outport; it stores packets whenever they arrive and outputs one whenever a request is received from downstream. If a system does support F3, components like the filter and duplicator present additional complications: they can naturally work in multiple processing modes, and the programming system ought to support this generality. While the case of multi-mode single-inport, single-outport components is well understood [8, 10], there is little discussion of the general case in the literature beyond the statement that the problem is "complicated" [9].

2.2 Information-Flow Features in Existing Systems

Given the importance of features F1–F3, it seems clear that an information-flow programming system should aim to provide them. Unfortunately, all of the systems we surveyed have given up some of F1–F3 in order to support F0 (automatic invocation). The consequences of supporting F0 depend on system design:

1. Systems such as Click [1], which use buffered channels and rely on a runtime scheduler to invoke components, can run into deadlocks or unbounded buffer growth.
2. In thread-transparent Infopipes [9], adding components with multiple inports or outports introduces the risk of resuming a stale coroutine.
3. Shivers' and Might's online transducers with multiple inports or outports [11] risk using stale channel continuations.

These problems show up only in specific pipeline configurations (for example, connecting a data-driven outport to a demand-driven inport); they suggest that the combination of F1–F3 allows programmers to specify pipelines that are internally inconsistent and for which there is no reasonable implementation. Existing systems get around these problems by sacrificing one of the information-flow features, or by giving up F0 and requiring the programmer to specify the control-flow interaction of components.

Instead of sacrificing one of F1–F3 to achieve F0, DirectFlow allows programmers to define internally inconsistent pipelines but uses a context condition to detect these cases at compile time. It is the restricted expressivity of the DSL that makes this possible.

2.3 Comparison with Infopipes

DirectFlow evolved from the Infopipes middleware system [8]; following the terminology developed there, we call a component a *pipe* and a composition of components a *pipeline*. (To avoid confusion, we reserve "Infopipes" exclusively for the previous Smalltalk-based system.) The Infopipes system sits on top of an object-oriented language; it represents a pipe as an object and implements inter-pipe packet transfer with message send.

Infopipes support features F1–F3, but require programmers to manually arrange the invocation of pipes by declaring their *polarity configurations*. A polarity configuration defines the control-flow interface of a pipe by assigning a positive or negative polarity to each port. A positive port transfers data by sending a message (`push` for outports and `pull` for inports), and a negative port transfers data when it receives a message. Specifying the polarity configuration of a pipe is more complicated than merely writing a declaration: for each negative port the programmer has to define a method to run in response to messages received. Thus, the polarity configuration dictates how the functionality of the pipe must be divided into methods. Moreover, there are no general rules to help programmers decide what polarity configurations a pipe can have; although multi-mode ("polymorphic") pipes can be constructed, all of the methods necessary to make them work must be written by hand, and the programmer is responsible for ensuring that the data-driven and demand-driven behaviours are the same.

DirectFlow differs from Infopipes in three main ways. First, DirectFlow programmers define their components at a higher level; rather than defining *objects* that implement both the data-flow and the control-flow behaviour of their pipes, they define only the data-flow behaviour using a DSL. The DirectFlow compiler infers the control flow and generates the objects (specifically, Java classes). Second, DirectFlow deals with composition more consistently than did Infopipes: composite pipelines can be treated in exactly the same way as custom pipes, since they can be instantiated as many times as needed. Third, while both Infopipes and DirectFlow use ports, their role is different in the two systems. Infopipe ports are objects that exist at runtime: all packets flowing into or out of a pipe have to pass though a port. In DirectFlow, ports exist at configuration time, but they are eliminated before runtime, thus also eliminating a level of indirection in the information flow.

3 The DirectFlow DSL

The DSL component of DirectFlow is an embedded language for programming information-flow components. We developed the language under the following assumptions:

A1. A pipe transfers data packets through a fixed set of inports and outports.
A2. Pipes are *reactive*. A pipe does not own a thread of control: it runs only in response to external data transfer requests.

A3. An outport of an upstream pipe is connected to a single inport of a downstream pipe by an unbuffered channel.
A4. At most one thread of control is executing in a pipeline at a given time.
A5. The thread moves from one pipe to another along with the data; this is similar to the way that message sends in an object-oriented system cause the thread of control to move from one object to another.

We introduced assumption A1 to simplify the design of DirectFlow, while assumptions A3 and A5 are inherited from the Infopipes system. Assumptions A2 and A4, which may seem rather strong, apply not to the entire information-flow application, but only to individual segments implemented in DirectFlow. Programmers can implement any segment of the system that they wish to run without scheduling overhead as a DirectFlow pipeline and then create higher-level logic to connect these segments. For example, a voice-mail server might use separate pipelines to serve separate clients, and each client might have its own pipeline that takes packets from the network, decrypts them, and presents audio to the user. Even though A2 and A4 still hold within each pipeline, the application as a whole involves multiple threads and run-time scheduling.

3.1 Language Overview and Syntax

The DirectFlow language is inspired by Hoare's Communicating Sequential Processes (CSP) [12] and therefore bears some resemblance to occam [13]. Using CSP abstracts away from the polarity configurations of the objects; programmers see a DirectFlow module as a sequentially executed thread that uses input and output statements to perform data transfers. Like occam, DirectFlow supports nondeterministic branches, so a DirectFlow module can perform concurrent blocking I/O operations in the style of the POSIX `select` system call.

DirectFlow is not a complete, stand-alone programming language; for example, it contains no support for arithmetic or other data manipulation operations. To be useful the DSL must be embedded into a general-purpose programming language, which we call the *host language*. This embedded design strategy has the benefit of reducing the work of language design (there is no need to reinvent all the wheels and do a bad job). It also simplifies the task of porting an existing program to DirectFlow: the data input–output interface will need to be changed, but the code that actually manipulates the data can be brought over from the host language unchanged.

In the work described here we embedded DirectFlow DSL into Java, and matched DirectFlow's concrete syntax to that of Java. However, it is a simple matter to change the concrete syntax when integrating DirectFlow with another host language: we developed a previous prototype using Smalltalk syntax and integrated it with Squeak.

The syntax of the DirectFlow/Java language is shown in Fig. 2. A DirectFlow/Java module contains the name of the pipe, an optional Java superclass

declaration (discussed in Sect. 4.4), port declarations, and a data-processing process block. The process block defines how the pipe performs input–output and processes data packets in each iteration; there is an implicit unbounded loop around the block. Code in the block can make use of three new kinds of primitive statement: *alternative*, *input* and *output*.

alternative. The meaning of alt $stmt_1$ with $stmt_2$ with $stmt_3$ is that one of the branches $stmt_1$, $stmt_2$ or $stmt_3$ is executed. The programmer of a pipe has no direct control over which; this is determined at run time based on the outstanding input and output requests from other pipes. The meaning of an alt statement does not depend on the order of the branches.

input. The meaning of *var* = *port* ? is that the pipe inputs a packet from the inport *port* and stores the packet in the variable *var*.

output. The meaning of *port* ! *expr* is that the pipe evaluates the expression *expr* and outputs the result through the outport *port*.

The primitive statements in our DSL correspond closely to those of CSP, but with a few modifications. For example, a DirectFlow module has no way to invoke itself, or indeed any other DirectFlow module. We have also eliminated the CSP parallel composition operator || because it is not useful in our context (due to A4). In contrast with the CSP choice operator [], which requires that the first statement in each branch performs a distinct input or output operation, the alt statement in the DirectFlow DSL does not restrict the first statements in the branches.

3.2 The Context Condition

To ensure that a DirectFlow module corresponds to a reactive pipe, we introduce one *context condition* restricting how the pipe performs input–output through

---

module ::= pipe name (extends super)? '{' *portdecl*+ process '{' *stmt*+ '}' '}'
portdecl ::= (inport | outport) *port* (',' *port*)+ ';'
var ::= JAVA VARIABLE IDENTIFIER
expr ::= JAVA EXPRESSION
stmt ::= JAVA STATEMENT | *alternative* | *input* | *output*
alternative ::= alt *stmt* (with *stmt*)+
input ::= *var* = *port* '?' ';'
output ::= *port* '!' *expr* ';'

Fig. 2. DirectFlow/Java syntax in Backus-Naur form extended with ? (indicating 0 or 1 repetitions), and + (indicating 1 or more repetitions). We list only those grammar rules that augment Java syntax. The top level nonterminal symbol is *module*; the *alternative*, *input*, and *output* statements can appear at any place in a DirectFlow module at which an ordinary Java statement can appear.

its ports. Before we can state the context condition we need to introduce two definitions.

- An *alt specialization* of a module is the module that results from replacing each alternative statement by one of its branches. For example, the module { a; alt b; with c; } has two alt specializations: { a; b; } and { a; c; }.
- A port is an *index port* of an alt specialization if the port is accessed exactly once in the execution of the alt specialization, regardless of control flow, and if the port is *not* accessed in any *other* alt specialization of the module. It does not matter if the port is an inport or an outport.

The context condition we impose on DirectFlow is that *every alt specialization in a valid module must have at least one index port*. The motivation behind this requirement is twofold. First, we want to ensure that every alt specialization is triggered by data transfer at a certain port. The index port of the alt specialization is that port; transferring data through the index port executes the alt specialization. Second, we need to determine which branch of an alt to execute. By requiring that each alt specialization has an *exclusive* index port not accessed elsewhere, we can always make this determination when a data transfer request arrives at an index port. In essence, this rule lets us implement angelic nondeterminism, *i.e.*, we make alt choices not arbitrarily but in a way that ensures that the pipe can conduct the necessary data transfer.

```
pipe Filter {
  inport in;
  outport out;

  process {
    Object packet;
    packet = in ?;
    out ! packet;
  }
}
```

```
pipe Duplicate {
  inport in;
  outport out0, out1;

  process {
    Object packet;
    packet = in ?;
    out0 ! packet;
    out1 ! packet;
  }
}
```

```
pipe Separate {
  inport in;
  outport out0, out1;

  process {
    Object packet;
    packet = in ?;
    alt   out0 ! packet;
    with  out1 ! packet;
  }
}
```

Fig. 3. DirectFlow modules for the Filter, Duplicate and Separate pipes of Sect. 2.

Fig. 3 shows three pipes in DirectFlow/Java. Port names in a pipe are not values in the host language and thus cannot be stored in variables; they can be used only as the subjects of the ? and ! constructs.

Since the Filter module does not contain an alt statement, its alt specialization is trivially the same as the module itself. Both in and out are index ports of this program, and thus it satisfies the context condition. Likewise, the alt specialization of the Duplicate module is the same as itself. Ports in, out0 and out1

```
pipe Switch {
  inport in0, in1;
  outport out0, out1;

  process {
    Object packet;
    alt    packet = in0 ?;
    with   packet = in1 ?;
    alt    out0 ! packet;
    with   out1 ! packet;
  }
}
```

Fig. 4. This Switch pipe definition is not a valid DirectFlow module because none of its alt specializations has an index port.

are all index ports, and the module satisfies the context condition. The Separate module has two alt specializations: { Object packet; packet = in ?; out0 ! packet; } and { Object packet; packet = in ?; out1 ! packet; }. The first has out0 as an index port; the second has out1 as an index port. Note that the port in appears in *both* alt specializations, so it cannot be an index port of either.

The Switch module in Fig. 4 is an example of an invalid DirectFlow module. It has four alt specializations: { packet = in0 ?; out0 ! packet; }, { packet = in0 ?; out1 ! packet; }, { packet = in1 ?; out0 ! packet; }, and { packet = in1 ?; out1 ! packet; }. However, none of these alt specializations has an index port because each port is accessed in two different specializations.

4 Compiling DirectFlow Modules

In the previous section we described how DirectFlow uses a CSP-inspired programming model to eliminate the polarity-declaration requirement in Infopipes. Instead the DirectFlow compiler infers the polarity configurations of a pipe and generates the pipe objects. In this section we first discuss the inference process conceptually, and then present the details of our inference algorithm. Finally, we explain how the compiler generates code.

4.1 Principles of the Compilation Process

An object is characterized by its behaviour: its protocol (the messages that it can understand), and the responses that it makes to those messages. Most "object-oriented" languages, including Java, specify both of these things by defining a set of methods: the names of the methods define the protocol, and the bodies of

the methods define the response to each message. So we have:

$$\text{methods in an object} \implies \begin{cases} \text{messages the object can understand} \\ \text{code to run when receiving a message} \end{cases}$$

However, this is not the only way of defining the behaviour of an object. DirectFlow modules can also be viewed as defining objects. Their protocol is limited to a subset of the messages push and pull, while the response to these messages is given implicitly by the DirectFlow module. The task of the compiler is to determine if the object corresponding to a DirectFlow module understands push, pull, or both, and to generate Java methods that implement the data-processing functionality defined in the DirectFlow code. So we start by inverting the diagram above:

$$\text{methods in the object} \impliedby \begin{cases} \text{messages an object can understand} \\ \text{code to run when receiving a message} \end{cases}$$

How can we decide what messages a pipe object can understand? The DirectFlow code might say that a pipe performs output on a certain port, but it does not explicitly say whether the port is positive or negative. If the port is positive, it will *send* messages to transfer data to or from some other pipe, but will never *receive* messages. Only negative ports can receive messages: negative inports receive push messages and negative outports receive pull messages. Thus, if we can determine the possible polarity configurations of a DirectFlow program, we can infer its protocol. Can we also infer the body of the method to execute in response to an incoming message? We can, *provided that*, for each negative port, there is a single piece of code that executes a data transfer on the port. So we constrain valid polarity configurations to ensure that each negative port must be associated with one alt specialization, and that it is accessed exactly once in the execution of one cycle of the pipe. In other words, a port can be negative only if it is an index port as defined in Sect. 3.2.

Thus the following strategy lets us derive an implementation for a DirectFlow module as a Java class:

> For each alt specialization, mark one of its index ports with negative polarity. Mark all other ports with positive polarity. The code to run when receiving a message through a negative port is the alt specialization that accesses the port.

This strategy determines how many polarity configurations a pipe has. The context condition of Sect. 3.2 ensures that a DirectFlow module has at least one polarity configuration, but some pipes have more than one. Consider the Filter pipe in Fig. 3. Because both in and out are index ports, applying this strategy to Filter produces the data-driven and demand-driven polarity configurations as documented in the filter design pattern [10]. Fig. 5 shows the results of applying this strategy to the Duplicate and the Separate pipes.

Infopipe	in	out0	out1
	−	+	+
Duplicate	+	−	+
	+	+	−
Separate	+	−	−

Fig. 5. Polarity configurations of the Duplicate and Separate pipes in Fig. 3.

4.2 Mapping an Invalid DirectFlow Module to Objects

Let us try to implement the Switch DirectFlow module in Fig. 4 (which violates the context condition) as a pipe object. We will fail, but it is instructive to see why. We start by considering which ports should have negative polarity.

- Suppose that in0 and in1 are negative, and the other ports are positive. Since branches in the second alt construct do not access any negative ports (out0 and out1 are both positive), the object cannot use the received message to decide whether to send the outgoing packet through out0 or out1.
- Suppose ports in0, in1, and out0 are negative. This pipe cannot respond to data request on out0 in a useful way because it has no way of requesting data from upstream, but must instead block until an upstream pipe pushes data in (which may never happen).
- Suppose that ports in1 and out0 are negative. With this polarity configuration we can eliminate the blocking behaviour by making the object pull from in0 when it receives a pull message associated with out0, and push to out1 when it receives a push message associated with in1. However, such an object does not implement the Switch Infopipe faithfully because our arbitrary choice rules out the execution paths in0 — out1 and in1 — out0.
- Suppose that all ports are positive. Since pipe objects do not have their own threads, and this pipe object does not have a thread entrance, it will never acquire a thread and will just sit idly doing nothing. This behaviour is quite useless, so we disallow the all-positive configuration.

Even though these discussions are based on our intuitive understanding of reactive objects, they all relate back to the context condition. These lines of reasoning should help programmers to understand the cause of the problem when the DirectFlow compiler rejects a module that violates the context condition, or why the compiler does not allow a pipe to have a specific polarity configuration.

4.3 Inferring Polarity Configurations of DirectFlow Modules

The compilation process reflects the preceding discussion. It starts by identifying the alt specializations of the module using a mechanism called *alt lifting*. It then determines if each alt specialization has one or more index ports; if there is no index port, the context condition has been violated. The next step is to

Code pattern	Rewrite result	
alt $\langle\ s_1, ..., s_m\ \rangle$; t	alt $\langle\ s_1$; $t, ..., s_m$; $t\ \rangle$	(seq-a)
s ; alt $\langle\ t_1, ..., t_n\ \rangle$	alt $\langle\ s$; $t_1, ...,$ s ; $t_n\ \rangle$	(seq-b)
if (c) { alt $\langle\ s_1, ..., s_m\ \rangle$ } t	alt $\langle$ if (c) s_1 $t, ...,$ if (c) s_m $t\ \rangle$	(if-a)
if (c) s { alt $\langle\ t_1, ..., t_n\ \rangle$ }	alt $\langle$ if (c) s $t_1, ...,$ if (c) s $t_n\ \rangle$	(if-b)
alt $\langle\ s_1, ...,$ alt $\langle\ t_1, ..., t_n\ \rangle, ..., s_m\ \rangle$	alt $\langle\ s_1, ..., t_1, ..., t_n, ..., s_m\ \rangle$	(alt)

Fig. 6. The rewrite rules for alt lifting in an abstract-syntax notation. We enclose alt branches in angle brackets separated by commas. The semicolon (;) in the seq rules is the statement sequence operator.

compute the polarity configurations by selecting one index port from each alt specialization and marking it as a negative port. The compiler can then generate a pipe object with a push method and a pull method. The push method contains alt specializations with negative inports, and the pull method contains alt specializations with negative outports. If the pipe has multiple polarity configurations, we generate a separate pipe object for each configuration. We now describe the processes of alt lifting and of computing the polarity configurations in more detail.

Alt Lifting. The alt lifting procedure computes the alt specializations of a DirectFlow module through a series of rewrites. The rewrite rules, listed in Fig. 6, make use of the property that conditional, looping, and statement sequencing constructs all distribute over alt, so we can "lift" alt up without changing the meaning of the module. Since code duplicated by the rewrites goes into different alt branches, and an alt construct always executes only one branch, the duplication cannot cause multiple executions of the same statement. The procedure repeatedly rewrites the module until all alt constructs are at the top level, at which point each alt branch would be an alt specialization. The observant reader will notice that there is no rule for loops. A module that contains an alt in a loop always violates the context condition, so the compiler never needs to lift it.

Example: Applying alt lifting to Separate and Switch. We illustrate the alt lifting procedure by applying it to two examples. The Separate pipe in Fig. 3 contains only one alt construct and the rewrite terminates in one step. The Switch pipe in Fig. 4 contains two alt constructs; the rewrite starts by lifting either of the alt constructs and then proceeds by flattening the nested alt constructs using the alt rewrite rule. Fig. 7 shows the pipes after alt lifting; each top-level alt branch corresponds to an alt specialization of the module.

Computing Polarity Configurations. The DirectFlow compiler computes the polarity configurations of a DirectFlow module by first identifying the index ports of each alt specialization, and then computing the Cartesian product of the sets of index ports. The algorithm proceeds as follows.

```
pipe Separate {
  inport in;
  outport out0, out1;

  process {
    alt {
      Object packet;
      packet = in ?;
      out0 ! packet;
    } with {
      Object packet;
      packet = in ?;
      out1 ! packet;
    }
  }
}
```

```
pipe Switch {
  inport in0, in1;
  outport out0, out1;

  process {
    alt {
      Object packet;
      packet = in0 ?;
      out0 ! packet;
    } with {
      Object packet;
      packet = in1 ?;
      out0 ! packet;
    } with {
      Object packet;
      packet = in0 ?;
      out1 ! packet;
    } with {
      Object packet;
      packet = in1 ?;
      out1 ! packet;
    }
  }
}
```

Fig. 7. The Separate and Switch pipes after alt lifting. The original DirectFlow source for these two pipes are shown in Figs. 3 and 4.

Step 1. Let B be the set of alt specializations of the program; for each $b \in B$, compute the set P_b of all the ports accessed in b.

Step 2. For each $b \in B$, compute the set Q_b using the following equation:

$$Q_b = P_b \quad - \bigcup_{i \in B - \{b\}} P_i$$

Q_b is the set of ports that are accessed in the alt specialization b and nowhere else.

Step 3. For each $b \in B$, eliminate from Q_b all ports accessed in a loop, accessed in only one branch of an if construct, or accessed multiple times in an execution path. This step is a simple control-flow analysis that enforces the "exactly once" part of the definition of index ports. The resulting sets Q_b contain the index ports of the alt specializations in B.

Step 4. We compute the negative ports in a polarity configuration by choosing one port from the set Q_b for each specialization b. We represent a polarity configuration by a tuple of its negative ports, and the set N of configurations is given by

$$N = \prod_{i \in B} Q_i$$

where $\prod$ represents Cartesian product on sets.

In our experience, the crude control-flow analysis algorithm in Step 3 works fairly well. If it later turns out that imprecision in this algorithm rules out some polarity configurations on useful DirectFlow modules, we can apply more sophisticated control-flow analysis algorithms. (These could, for example, recognize mutually exclusive if conditions to infer a larger set of polarity configurations.)

Example: Polarity configurations of Separate. We number the alt branches in Fig. 7 according to their order of appearance and use the module to demonstrate how to compute polarity configurations.

Step 1. $P_1 = \{ \mathsf{in}, \mathsf{out0} \}$, $P_2 = \{ \mathsf{in}, \mathsf{out1} \}$
Step 2. $Q_1 = P_1 - P_2 = \{ \mathsf{out0} \}$, $Q_2 = P_2 - P_1 = \{ \mathsf{out1} \}$
Step 3. No changes to Q_1 and Q_2
Step 4. $N = Q_1 \times Q_2 = \{ \langle \mathsf{out0}, \mathsf{out1} \rangle \}$

The set N has only one element, so Separate has one polarity configuration. The ports out0 and out1 are negative, and hence in is positive.

Example: Polarity configurations of Switch. We now apply the same constraint solving process to the lifted Switch Infopipe in Fig. 7.

Step 1. $P_1 = \{ \mathsf{in0}, \mathsf{out0} \}$, $P_2 = \{ \mathsf{in1}, \mathsf{out0} \}$, $P_3 = \{ \mathsf{in0}, \mathsf{out1} \}$, $P_4 = \{ \mathsf{in1}, \mathsf{out1} \}$
Step 2. $Q_1 = Q_2 = Q_3 = Q_4 = \emptyset$

At this point we can conclude that the Switch DirectFlow module does not have a polarity configuration because none of its alt specializations has an index port, and the Cartesian product of any set with an empty set is empty. In this case the compiler reports an error instead of continuing with the code generation process.

4.4 Generating Code for Pipe Objects

When a DirectFlow module is compiled, the compiler analyses the program, determines how many polarity configurations are possible for the pipe that it defines, and (assuming that there is at least one legal configuration) outputs several Java class definitions. A Java *logical* pipe class is always created; the logical pipe is like an abstract model of the pipe's behaviour. Inside this class, amongst other things, is a method validConfigurations that returns a set of polarity configurations; in Fig. 8 we assume that there are three. For each configuration, the compiler outputs a Java class definition for a *physical* pipe class. We don't yet know which of these classes will be used, because this depends on the context

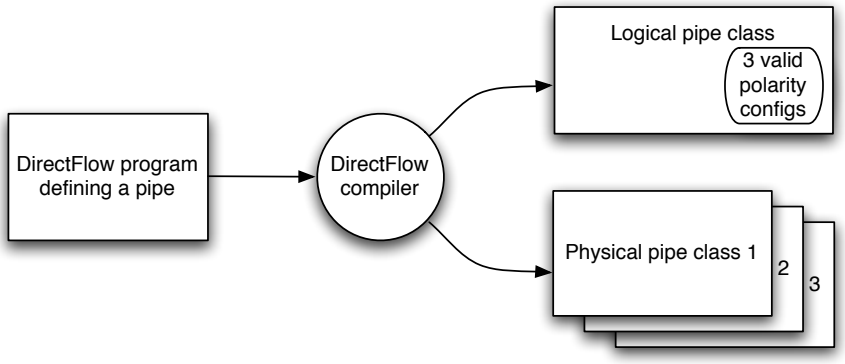

Fig. 8. Defining a DirectFlow pipe using the domain-specific language

into which the pipe is eventually deployed, so the most useful thing to do is to generate all of them.

Each physical pipe Java class implements the data-processing functionality of the DirectFlow module. In addition to the ability to transfer data packets to and from other pipe objects, a DirectFlow physical pipe object can:

1. invoke methods in Java objects,
2. maintain persistent state between executions, and
3. make its persistent state accessible from other Java objects.

In this subsection we describe the design of physical pipe objects and how the DirectFlow compiler generates these objects from a DirectFlow module.

At the core of a physical pipe object are its push and pull methods. The push method accepts a port number and a data packet; the pull method accepts a port number and returns a data packet. (The compiler translates port names in DirectFlow to port numbers.) Both methods contain a top-level test of the port number, and execute a different segment of code depending on the port through which data transfer occurred; see SeparatePhysicalPipePNN in Fig. 9 for an example. Each of these segments is derived from one alt specialization, and the derivation depends on whether the specialization has a negative inport or a negative outport. If an alt specialization has a negative *inport*, it works in data-driven mode and therefore the code belongs in the push method (because an upstream component will push data into this component). If an alt specialization has a negative *outport*, it works in demand-driven mode and the code belongs in the pull method.

This translation realizes our intuition that negative ports act as thread entrances, while positive ports act as thread exits. Since each index port (and therefore, each negative port) is accessed exactly once, we can be certain that each invocation of a data-driven alt specialization consumes one data packet,

```
public class DuplicatePhysicalPipeNPP {
  public void push(int index, Object input) {
    if (index == 0) {
      Object packet;
      packet = input;
      out0.push(packet);
      out1.push(packet);
      return;
    }
  }
}
```

```
public class DuplicatePhysicalPipePNP {
  public Object pull(int index) {
    Object output;
    if (index == 1) {
      Object packet;
      packet = in.pull();
      output = packet;
      out1.push(packet);
      return output;
    }
  }
}
```

```
public class SeparatePhysicalPipePNN {
  public Object pull(int index) {
    Object output;
    if (index == 1) {
      Object packet;
      packet = in.pull();
      output = packet;
      return output;
    }
    if (index == 2) {
      Object packet;
      packet = in.pull();
      output = packet;
      return output;
    }
  }
}
```

Fig. 9. Excerpt of physical pipe objects compiled from the DirectFlow modules in Fig. 3. We show only two of the three classes generated from the Duplicate pipe because the PPN case is analogous to the PNP one (left bottom).

and each invocation of a demand-driven alt specialization produces one data packet. Fig. 9 shows the physical pipes generated from the Duplicate and the Separate Infopipes.

We use the generation-gap pattern [14] in the physical pipe objects to integrate the code generated from the DirectFlow DSL with ordinary Java code written by hand. The original formulation of the generation-gap pattern puts machine-generated code in a superclass of the hand-written code; this allows the machine-generated code to be customized by subclassing. Such an approach does not quite work in our case because the compiler can generate *multiple* physical pipe classes from a DirectFlow module, and because the number of generated physical pipe classes may change with the contents of the DirectFlow module. Instead we make the machine-generated classes *subclasses* of the hand-written code; this allows all of the generated classes to reuse the same hand-written code. This is the purpose of the extends *superclass* clause mentioned in Sect. 3.1: it defines the name of the hand-written class from which the generated physical pipe classes should inherit. We call this variant the *inverse generation-gap* pattern.

The inverse generation-gap pattern enables a physical pipe object to maintain persistent state in the fields of its superclass; it also allows other Java objects to access that persistent state through the public methods of the superclass. The

DirectFlow compiler copies all Java statements in a DirectFlow module into the translated alt specializations in the physical pipe classes, so the programmer is able to invoke other Java methods from within the DirectFlow module.

5 The DirectFlow Framework

The DirectFlow framework is designed to help the programmer build an information flow system in several steps. Although this may at first sight appear unnecessarily complicated, the stepwise development allows for maximal reuse. Let us draw an analogy with object-oriented programming languages. If what one wants is objects, classes may seem like an unnecessary complication — why not define objects directly? However, it turns out that classes are quite useful when one needs to make many objects that are almost the same. Similarly, generic classes may seem like an unnecessary complication, but they turn out to be quite useful when one needs to make several classes that differ parametrically.

5.1 Building a Pipeline

The first step in using DirectFlow is to define any custom pipes that are necessary for the application at hand. There is a library of standard pipes, which we expect to grow over time, but let us assume that at least one custom pipe is required, for example, a pipe that diverts suspicious data packets (say, into a log) to help track down a suspected sensor malfunction.

As described in Sect. 4.4, the result of compiling these custom pipe modules is a collection of logical and physical pipe classes. The next step is to take these generated classes and to write a Java program that composes them with appropriate library classes to build the desired information pipeline. This process is illustrated in Fig. 10 for the simple case of a pipeline containing only two components. The thread configuring the pipeline first creates a CompositeFactory object, and then creates the two logical pipes, src and sink, that will be its components. src and sink are added to the factory, and the appropriate connections between the output port of src and the input port of sink are set up. Incremental checks are performed during this process; for example, we check to make sure that two outputs are not connected together. At this stage it is also possible to specify that an internal port of one of the components is "forwarded" to become an external port of the whole composition. When the programmer has completed the "wiring up" of the composite factory, the factory is told to instantiate the composition. Some non-local checks can now be carried out, for example, to ensure that there are no disconnected ports inside the composition. If all is well, the result of instantiation is another *logical* pipeline.

The composite logical pipeline has no open ports, so it can be instantiated as a physical pipeline: a pipeline that can actually carry data. This is not a simple matter, because the appropriate mode of processing of each component

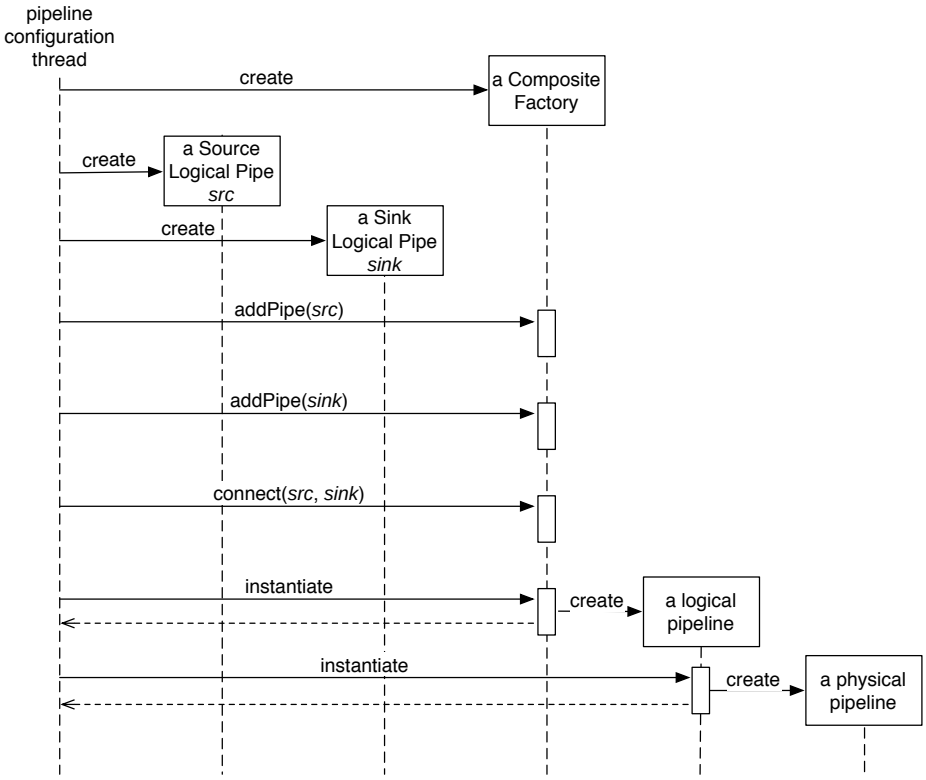

Fig. 10. A UML Sequence diagram showing how a physical pipeline is created.

in the composition depends on the mode of its neighbors. Now we see the value of building the logical pipeline first: it provides an abstract model of the composite pipeline that can be explored to gain a global understanding of the whole pipeline. This makes it possible to select the best physical pipe class to implement each logical pipe.

5.2 Composite Pipes

The DirectFlow system supports two kinds of pipes — simple pipes and composite pipes — and goes to some lengths to treat them on an equal footing. Simple pipes are created by instantiating a logical pipe object, which is an instance of the logical pipe class generated by the DirectFlow compiler. Composite pipes, which are compositions of other pipes, are constructed by instantiating a logical pipeline object, but in this case the logical pipe object is created by a composite factory.

To construct a composite pipe, the programmer starts by creating a CompositeFactory object and then invokes its methods to describe the structure of the desired composite pipe.

- addPipe(String n, LogicalPipe op) This method adds an internal pipe op with name n to the composite pipe.
- newInport(String ident) and newOutport(String ident) These methods adds an input port or an output port to the composite pipe. A composite pipe with ports can be further composed with other pipes.
- connect(Channel c) A Channel object identifies a pair of ports, each in a specific component. The connect method connects the two ports described by c. The programmer can connect an inport of the composite pipe to an inport of an internal pipe, an outport of an internal pipe to an inport of an internal pipe, or an outport of an internal pipe to a outport of the composite pipe.

Finally, invoking the instantiate() method on the CompositeFactory creates a CompositeLogicalPipe object. The resulting CompositeLogicalPipe object is like any other logical pipe object, and the programmer can pass it to addPipe to compose it with other pipes.

To create a physical pipe object that can perform data processing operations, the programmer *terminates* a logical pipe by connecting its inports and outports to special Source and Sink logical pipes, and then instantiates the logical pipe to produce a CompositePhysicalPipe object.

Because the resulting physical pipe has no open ports, it cannot be sent push or pull messages. Instead we use the getInternal method to obtain references to the Source and Sink physical pipes, and use these objects to inject data and control into the pipeline. This design hides the communication protocol between physical pipe objects and allows it to evolve without changing the public interface of the pipeline.

6 Related Work

Historical Information-Flow Systems. Computer programs have long been structured to process streams of information from external devices. The idea of a stream as a first-class object can be traced back to Stoy and Strachey's OS6 operating system [15]. OS6 also incorporated stream functions, which could be applied to an argument stream to construct another stream (with different contents) as the result. A related I/O system using (bidirectional) streams appeared in Unix [16]. In both OS6 and Unix, the stream subsystem is regarded as peripheral to the "main program", and it incurs high overhead due to the reliance on runtime scheduling and the need to move data across process boundaries.

Coordination Languages. Our work bears some resemblance to control-based coordination languages [17] in that they model a program as a collection of

entities connected by point-to-point channels. Unlike existing coordination languages, which treat processes as purely computational entities, DirectFlow also captures the communication aspects of a component by addressing it separately from the computation aspects of the component.

Communicating Sequential Processes. DirectFlow is inspired by Hoare's CSP [12]. Unlike libraries such as JCSP [18], which faithfully implement the semantics of CSP, DirectFlow eliminates certain restrictions (such as the prohibition against different alt branches starting with the same operation) while adding others (such as the context condition) to better support the development of information-flow programs.

Design Patterns. The way we build information-flow programs by exchanging messages between Infopipe objects has been documented under the name *filter pattern* [10]. The filter pattern literature describes (data-driven) sink filters and (demand-driven) source filters, but it does not discuss how to generalize these two cases to filters with multiple inputs or outputs. DirectFlow provides a more general and more elegant mechanism for building filters because it relieves the programmer from the responsibility of implementing both the data-driven and demand-driven variants of a filter.

Information-Flow Systems without F1. Some information-flow programming systems achieve automatic component invocation by restricting the form or the behaviour of custom components. Both thread-transparent Infopipes [9] and StreamIt [6] require a component to have exactly one inport and one outport, and Spidle [7] requires all channels to have the same data rate.

Information-Flow Systems without F2. Some information-flow programming systems connect components with buffered channels and achieve automatic invocation by runtime scheduling. Click [1] and StreamIt [6] both fall into this category. Such designs suffer from three problems. First, the reliance on runtime scheduling makes it difficult to understand the interaction between components and to ensure liveness. Second, channel buffering introduces latency and therefore breaks compositionality. Finally, buffering interferes with components such as a prioritizer whose operation depends on the interleaving of input and output.

Information-Flow Systems without F3. Some information-flow programming systems achieve automatic component invocation by supporting only data-driven or only demand-driven processing. The Eden operating system [19] explored making streams asymmetric by eliminating active output (data-driven) operations. Reactive objects in O'Haskell [20, 21] eliminate active input (demand-driven) operations. Dataflow languages like SISAL [22] and lazy streams, which typically appear in programs written in functional languages like Scheme or Haskell, support only data-driven operation. These systems cannot support the

prioritizer described in Sect. 2, which uses both data-driven and demand-driven data processing.

Polarity-Polymorphic Ports. Our previous Infopipes system [8] supports a feature called *polarity polymorphism* that allows an Infopipe object to assume multiple polarity configurations. In the system, an Infopipe defines a polarity-polymorphic port by providing it with both push and pull methods and specifying its polarity using a variable that is instantiated to either + or −. For example, a filter has polarity configuration $\alpha \to \bar{\alpha}$ that is instantiated to $+ \to -$ or $- \to +$. The runtime system determines the polarity configuration of an Infopipeline by unifying the polarity specifications of connected ports.

The Click modular router [1] uses *agnostic ports* for the same purpose. The agnostic polarity specifies that the port can work in either demand-driven or data-driven mode, and the programmer is required to specify a *flow code* for any processing element that contains agnostic ports to help the runtime system decide the polarity configurations of the element. A flow code is a sequence of *port codes* that specify the internal dataflow of an element; if the codes of two ports share the same letter, data packets arriving from one port may exit from the other. For example, a filter has flow code "x/x" indicating that packets arriving from the inport can exit from the outport. We do not completely understand how Click computes the polarity configurations of an element from its flow code, but the design of agnostic ports appears to be similar to polarity polymorphism. Both designs share the following drawbacks.

1. When defining a component, the programmer must identify the ports that can assume either polarity and provide additional information on the relationship between the polarity of ports. There is no tool support for checking whether the supplied information is consistent with the behaviour of the component, and therefore the programmer is solely responsible for the correctness of the port polarity specifications.
2. The programmer must implement both push and pull methods for each polymorphic/agnostic port and ensure that the component exhibits the same information flow behaviour regardless of whether it is used with push or pull.
3. The mechanisms used to specify port polarity relations are not general enough to capture the polarity relations of components with more than two ports. There is no way to define a polymorphic Infopipe that works in all three configurations of the duplicate pipe in Fig. 5. Likewise, even though a duplicator, a separator, and a demultiplexer all have different information-flow behaviour and polarity configurations, they cannot be distinguished in the Click system because they share the same flow code "x/xx".

In comparison, our proposed technique both requires less programmer intervention (by automatically inferring the information flow behaviour of a pipe and generating the methods corresponding to each polarity configuration) and works in more general contexts (because it can distinguish between and correctly characterize duplicators, separators, and demultiplexers).

7 Experiences

We first prototyped DirectFlow as a Smalltalk-embedded language and then adapted it for embedding in Java. We implemented a DirectFlow/Java compiler in Haskell [23] and a corresponding run-time library in Java. Excluding comments and blank lines, the compiler is about 400 lines of Haskell, and the run-time library is about 600 lines of Java. The implementation is available through http://infopipes.cs.pdx.edu/.

One of the design goal for DirectFlow is to allow programmers to define pipes that correspond to data stream operators like those in the Aurora Stream Query Algebra (SQuAl) [2]. SQuAl defines a wide variety of primitive operators, which include standard relational algebra operators like Map (projection) and Union, generalized relational algebra operators like Filter (selection-based demultiplexing), and data-stream-specific operators like Resample. Some of these operators have multiple inports, some have multiple outports; only Map has a fixed relationship between its input data rate and its output data rate.

We studied SQuAl and concluded that DirectFlow is expressive enough to support all its operators. The only difficulty is that implementing SQuAl operators with variable arity requires a daisy-chain of pipes. For example, SQuAl has an n-stream union operation; because a DirectFlow pipe has a fixed set of ports, this must be implemented using a chain of two-stream unions components. However, since this can be implemented by a $1..n$ loop at pipeline configuration time, we do not see this as a problem.

Our study further suggests that using DirectFlow can improve the expressivity of existing data-stream management systems. To simplify system design and implementation, data-stream management systems such as Niagara and Aurora support only data-driven components. While they work well with data sources that produce infrequent discrete events, they are not well-suited for data sources that produce a continuous stream of time-varying data, such as thermometers or light sensors. Such sources are most naturally demand-driven: operating them in data-driven mode requires that they produce frequent updates, which wastes resources when the user does not need those values. However, reducing the frequency of updates compromises data freshness. DirectFlow naturally supports both data-driven and demand-driven components, so it would allow a data-stream management system to request data from continuous sources on demand, achieving the best of both worlds.

8 Conclusions and Future Work

Abstraction mismatch between the programming language and the application domain makes software development unnecessarily complicated. This is because making programmers use a language that exposes aspects of the system that are irrelevant to the domain forces over-specification and thus reduces reusability.

Dually, a language that hides aspects of the system that *are* relevant to the domain makes it more difficult to define and to reason about system behaviour in domain-specific terms.

We investigated the problem of abstraction mismatch in the information-flow domain and proposed the DirectFlow language to address the shortcomings of programming this domain with objects. By allowing programmers to define pipes without specifying their polarity configurations, DirectFlow eliminates the need to define and maintain multiple pipe objects that differ only in their polarity configurations.

The design of DirectFlow seeks a balance between language expressivity and implementation efficiency. The language allows a pipe to alter its input–output behaviour based on its internal state, which makes it possible to define demultiplexers and reordering buffers, components that are commonly found in information-flow programs. At the same time, DirectFlow is sufficiently restrictive to permit the compiler to perform static analysis on DirectFlow modules and to compile them to objects. DirectFlow is expressive enough to implement the Aurora stream algebra; whether it will be equally successful on other information-flow tasks is a question that we plan to explore in our future work.

In this paper we have demonstrated that DirectFlow simplifies the development of information-flow components by hiding the control flow interaction between them. It remains to be seen if DirectFlow facilitates reasoning about information-flow programs. We hope that deeper understanding of the semantics of DirectFlow will lead to progress in quality-of-service verification and in thread allocation for information pipelines.

Acknowledgments. This work is partially supported by the National Science Foundation of the United States under grants CCR–0219686 and CNS–0523474.

References

1. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The Click modular router. ACM Transactions on Computer Systems **18**(3) (August 2000) 263–297
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. International Journal on Very Large Data Bases **12**(2) (August 2003) 120–139
3. Krasic, C., Walpole, J., Feng, W.: Quality-adaptive media streaming by priority drop. In Papadopoulos, C., Almeroth, K.C., eds.: Network and Operating System Support for Digital Audio and Video, 13th International Workshop, NOSSDAV 2003. (June 2003) 112–121
4. Murphy-Hill, E., Lin, C., Black, A.P., Walpole, J.: Can Infopipes facilitate reuse in a traffic application? In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (October 2005) 100–101

5. Hart, J.K., Martinez, K.: Environmental sensor networks: A revolution in the earth system science? Earth-Science Reviews **78** (2006) 177–191
6. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In Horspool, R.N., ed.: Compiler Construction: 11th International Conference. Volume 2304 of LNCS. (April 2002) 179–195
7. Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., Pu, C.: Spidle: a DSL approach to specifying streaming spplications. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering: Second International Conference. Volume 2830 of LNCS. (September 2003) 1–17
8. Black, A.P., Huang, J., Koster, R., Walpole, J., Pu, C.: Infopipes: an abstraction for multimedia streaming. Multimedia Systems **8**(5) (December 2002) 406–419
9. Koster, R., Black, A.P., Huang, J., Walpole, J., Pu, C.: Thread transparency in information flow middleware. In Guerraoui, R., ed.: Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms. Volume 2218 of LNCS., Heidelberg, Germany, Springer-Verlag (November 2001) 121–140
10. Grand, M.: 6. In: Patterns in Java: a catalog of reusable design patterns illustrated in UML. Volume 1. John Wiley & Sons (1998) 155–163
11. Shivers, O., Might, M.: Continuations and transducer composition. In: PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (June 2006) 295–307
12. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA (1985)
13. SGS-THOMSON Microelectronics Ltd.: occam 2.1 Reference Manual. (1995)
14. Vlissides, J.: 3. In: Pattern Hatching: Design Patterns Applied. Addison Wesley (June 1998) 85–101
15. Stoy, J.E., Strachey, C.: OS6 — an experimental operating system for a small computer. Part 2: input/output and filing system. Computer Journal **15**(3) (August 1972) 195–203
16. Ritchie, D.M.: A stream input-output system. AT&T Bell Laboratories Technical Journal **63**(8) (October 1984) 1897–1910
17. Papadopoulos, G.A., Arbab, F.: Coordination Models and Languages. In: The Engineering of Large Systems. Volume 46 of Advances in Computers. Academic Press (September 1998) 329–400
18. Welch, P.H.: Process oriented design for Java: concurrency for all. In Arabnia, H.R., ed.: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000). Volume 1., CSREA Press (June 2000) 51–57
19. Black, A.P.: An asymmetric stream communication system. In: Proceedings of the Ninth ACM Symposium on Operating System Principles. (October 1983) 4–10
20. Nordlander, J., Carlsson, M.: Reactive objects in a functional language: an escape from the evil "I". In: Proceedings of the Third Haskell Workshop. (June 1997)
21. Nordlander, J., Jones, M.P., Carlsson, M., Kieburtz, R.B., Black, A.P.: Reactive objects. In: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (April 2002) 155–158
22. Feo, J.T., Cann, D.C., Oldehoeft, R.R.: A report on the Sisal language project. Journal of Parallel and Distributed Computing **10**(4) (December 1990) 349–366 Special issue: data-flow processing.
23. Jones, S.P., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge, UK (May 2003)