# DirectFlow: Toward a DSL for Infopipes

Chuan-kai Lin Andrew P. Black

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

## Abstract

Information-flow components can be represented as objects, but this representation is at once too general and overly specific. This is a problem because the generality makes it possible to define objects that cannot be treated as components, while the specificity requires that the programmer code details that are irrelevant to the information-flow abstraction. Instead, we advocate defining information-flow components in a domain-specific language (DSL), and generating the corresponding objects.

We present a DSL called DirectFlow that exploits the relationship between control flow and dataflow in the object model to allow programmers to define objects without specifying how messages are sent and received. We also describe a compilation algorithm for DirectFlow that infers the message sends and generates code for conventional objects. DirectFlow lets the information-flow programmer concentrate on the dataflow between objects rather than on the details of how this might be realized by objects.

***Categories and Subject Descriptors*** D.3.2 [*Language Classifications*]: Data-flow languages, Specialized application languages; D.3.3 [*Language Constructs and Features*]: Classes and objects; D.3.4 [*Processors*]: Compilers

***General Terms*** Object-Oriented Programming, Dataflow Programming, Domain-Specific Languages, Communicating Sequential Processes

***Keywords*** OOP, CSP, DirectFlow, DSL

## 1. Introduction

Software systems that process large or infinite streams of information are usually structured to correspond to the flow of information. Examples include software routers for network traffic [10], data stream query systems [1], real-rate video streams [12] and highway loop detector data analysis programs [14]. We think of such systems as networks of information processing components; the output of one component streams into the input of another (or multiple others) to build an "information pipeline". Given the predominance of object-oriented technology, it seems reasonable to expect that these basic components will be implemented as objects.

Previous work by us [3] and by others [10] has indeed focussed on defining information-flow components as objects, and on developing the operation protocol that these objects should use to communicate. Although this approach can be made to work, we have come to question whether or not objects are in fact the best tool for information-flow programmers.

Objects seem to be both too general and too specific to be an ideal model for information-flow components; these problems are discussed in some depth in Sections 2 and 3. The present paper proposes an alternative: describing information-flow components in a domain-specific language (DSL). Because the DSL is less expressive than a general-purpose object-oriented language, excessive generality is avoided. The same lack of expressivity makes it possible to infer some important properties of Infopipes — properties that, in the object-oriented formulation, the programmer was forced to state explicitly, even though these properties had nothing to do with the information processing function of the component. The DSL thus avoids the unnecessary specificity of the object-oriented formulation.

Since the DSL can be compiled into objects similar to those that would have been written by hand, there is no loss of execution efficiency with our proposed approach. We believe that there will be a significant gain in programmer productivity: using a DSL, the programmer can specify the functionality of the Infopipe just once, and leave it to the compiler to generate the several specialized versions necessary for different execution contexts.

The problem of mapping information-flow components onto objects is characteristic of *any* information-flow system that uses objects in its implementation, and the general form of our solution should be applicable to any such system. For the sake of concreteness, in the remainder of this paper we will talk about the problem as it arises in an Infopipe

system [3], and we will use Infopipe terminology. Another reason for this choice is that the implementation of our DSL has so far been restricted to Infopipes.

This paper proceeds to explore the problems that arise when defining Infopipes as objects, and then proposes a solution to these problems using the *DirectFlow* domain-specific language. We make the following specific technical contributions.

1. We identify the *cause* of various problems that arise when defining information-flow components as objects: over-specification due to abstraction mismatch (Sections 2 and 3).
2. We propose the DirectFlow language, which allows the programmer to define Infopipes without specifying how object-oriented message send is used to transfer packets between them (Section 4).
3. We design an algorithm to determine the possible interfaces that an Infopipe component defined in DirectFlow can offer when realized as an object (Section 5).
4. We describe the design and implementation of a compiler that translates an Infopipe component defined in DirectFlow into an object with a suitable interface (Section 6).

## 2. Context

### 2.1 Objects and information flow

The fundamental idea of the object-oriented paradigm is that programs can be made out of objects that interact by sending and receiving messages. The programmer defines an object by specifying the messages that it can receive, the actions that it should take after receiving each of them, and how it makes use of other objects by sending messages to them.

The prominent role of messages encourages structuring object-oriented programs according to control flow. Thus, interaction between objects tends to follow the client–server model: an object A requests a service from another object B by sending it a message. The interface of an object — the specification of the messages that it can understand — specifies also the services that the object can provide.

In contrast, information-flow programs, which process potentially unbounded streams of packets, tend to be structured around their internal dataflows. Since the purpose of these programs is to process data streams, it is natural to think of them as being made of elements that can input, process, and then output packets, and not as being made of objects that provide services.

The generality of the object model is unnecessary: all information-flow components understand the same limited protocol of push and pull messages, which say nothing about the services that they provide. Moreover, unlike objects, they do not make direct use of other components by sending them messages; instead components are connected together, and information flows from one to the next.

### 2.2 Infopipes

Our model for information-flow systems centers around an abstraction called an *Infopipe*, which is the basic unit out of which systems are composed. An Infopipe can input data through its input ports (*inports*) and output data through its output ports (*outports*). An Infopipe has no knowledge of its upstream and downstream neighbors, which are the source and destination of the packets that it processes; this information is supplied later, when a number of Infopipes are composed into an Infopipeline.

Packets pass between Infopipes over channels that link Infopipe ports; a channel provides unidirectional, unbuffered, synchronous communication between Infopipes. The Infopipe abstraction allows the programmer to concentrate on the important aspect of information-flow programs (how data move around in the program) and forget about unimportant details (such as the order in which different pieces of code are invoked).

A prototype implementation uses Smalltalk objects to represent Infopipes [3]. In this prototype, data transfer between Infopipes is accomplished by exchanging Smalltalk messages, and so each port corresponds to either a message sender or a message receiver. Programmers define Infopipes using Smalltalk, and channels are built by sending specific messages (such as $->>$) to establish connections between port objects.

### 2.3 Active and reactive Infopipes

In the Smalltalk prototype most Infopipes are *reactive* rather than *active*; they execute only when sent messages, and are otherwise idle. There are two exceptions. The first exception covers Infopipes that represent *active source and sinks*; these are external devices such as video cameras, which inject both a packet and a thread of execution into the system The second exception covers instances of a particular kind of primitive Infopipe called a Pump, which is used to explicitly introduce activity. In essence, each pump corresponds to a process executing at a given rate; there must be at least one pump on each hardware processor that hosts a segment of an Infopipeline, because otherwise the Infopipes in that segment will never execute.

In this paper we consider only reactive Infopipes — the components that fit between the pumps. Such Infopipes execute in response to messages and do not have their own thread of control. The issue of how to allocate pumps to a pipeline to maximize parallelism and minimize overhead is an important one, but is not considered here. However, an assignment of pumps to an Infopipeline also implicitly defines the reactive segments of the pipeline, to each of which we can apply the techniques discussed in this paper.

## 3. Port polarity

Objects are a good match for a dataflow-based abstraction such as Infopipes. Using objects to represent Infopipes in-

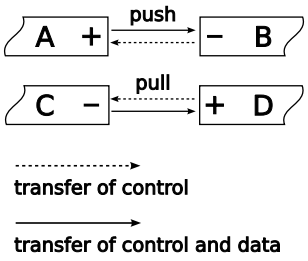


**Figure 1.** Port polarity and the use of messages to transfer data. Infopipe A transfers data to Infopipe B as an argument of the push message, and Infopipe C transfers data to Infopipe D as the return value of the pull message. In both cases data move from the left to the right. The + and − symbols represent the polarity of the port: positive ports send messages, and negative ports receives messages.

stead of investing in new language features makes it easy to integrate Infopipes into an existing object-oriented language, and message-send can transfer data between objects at low cost.

One of the key differences between objects and Infopipes is that objects send and receive messages, whereas Infopipes input and output data packets. The mapping from exchanging data to exchanging messages is not unique because there are two ways to transfer data between objects using messages. A message sender can transfer data to the message receiver as an argument (push); alternatively, a message receiver can transfer data to the message sender as a return value (pull). The difference between a message receiver and a message sender shows up in both the implementation and the interface of an object, so it seems that it must somehow be incorporated into the Infopipe abstraction.

In our previous work [3] we manifested this difference by adding a polarity attribute to each Infopipe port. A port has *positive polarity* if it transfers data by sending messages, and a port has *negative polarity* if it transfers data by receiving messages. Two ports can be connected by a channel only if they have opposite polarity and opposite direction (one is an inport while the other is an outport). It is important to realize polarity and direction are orthogonal. For example, positive output is represented by sending push messages, whereas positive input is accomplished by sending pull messages. Figure 1 illustrates how messages are used to transfer data between different kinds of ports.

### 3.1 Polarity configurations

We use the term *polarity configuration* of an Infopipe to mean the polarity assignment of all of its ports. The polarity configuration of a particular Infopipe is fixed[1] because the set of messages that an object sends and receives is fixed. Given an Infopipe, it is natural to ask whether there is another Infopipe with a different polarity configuration that exhibits the same behavior information-flow behavior. For some Infopipes the answer is yes; Figure 2 shows one such example. The PushFilter and PullFilter Infopipes are identical from the information-flow perspective: they both apply a transformation function to an information flow, and for every data packet input a corresponding packet is output. However, PushFilter and PullFilter are different objects with different interfaces: PushFilter sends and receives push messages, and PullFilter sends and receives pull messages. The implementations of these Infopipes also differ because a positive port corresponds to a set of statements that send messages, and a negative port corresponds to a method that is invoked in response to receiving a message. In this paper we will refer to an Infopipe like PushFilter as having multiple valid polarity configurations.

Given an Infopipe and a polarity configuration, another natural question is whether there is an Infopipe with the given polarity configuration that exhibits the same behavior as the given Infopipe. In many cases the answer is no. For example, an Infopipe with the same behavior as PushFilter cannot have a negative inport and a negative outport, because such an Infopipe would have no way to ensure that there is an output corresponding to every data packet input. An Infopipe with the same behavior as PushFilter, and which has two positive ports, cannot be reactive; it must have an internal process, and thus lie outside the scope of this paper.

This discussion suggests that there is a connection between the behavior of an Infopipe and its polarity configuration, and that the connection arises from the interaction between the semantics of message send and receive in the object model and the data I/O behavior of the Infopipe. We will characterize the interaction in Section 5 and show how to use the characterization to compile DirectFlow programs to objects in Section 6.

### 3.2 Problems with polarity

Programming Infopipes as objects and using messages to transfer data between them introduces port polarity into the Infopipe abstraction, which in turn creates several difficulties for information-flow programmers.

First, if an Infopipe has multiple valid polarity configurations, it is reasonable to expect that the Infopipe be made available in all valid configurations. This means that the Infopipe programmer must write (and subsequently maintain) multiple Infopipe objects that have the same functionality, or that a way must be found to transform an Infopipe into an equivalent one with different polarity. Transformation at the level of objects is difficult: since a positive port corresponds to a set of statements that send messages, and a negative port corresponds to a method that is invoked in response to receiving a message, changing the polarity configuration of an Infopipe requires a nontrivial refactoring.

[1] Reference [3] discusses polarity-polymorphic Infopipes, but these are in reality nothing more than two different Infopipe implementations coexisting inside the same object. For these Infopipes, the polarity assignment is fixed at connection time.

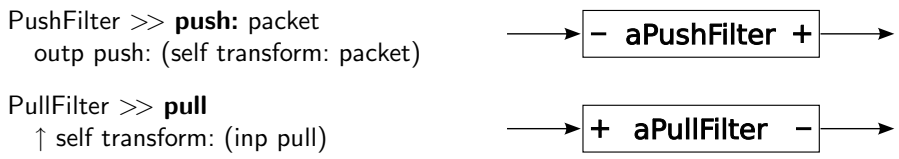


**Figure 2.** Two filter Infopipes with different polarity configurations that exhibit the same behavior. The PushFilter Infopipe inputs data from its negative inport by accepting push messages and outputs data to its positive outport by sending push messages. The PullFilter Infopipe inputs data from its positive inport by sending pull messages and outputs data to its negative outport by accepting pull messages. Even though the Infopipes have exactly the same behavior from the dataflow perspective, the information-flow programmer using objects is forced to distinguish them.

Koster et al. [11] explored an alternative approach, in which polarity was transformed by wrapping Infopipe implementations with co-routines.

Second, when an Infopipe is implemented by multiple objects with different polarity configurations, these objects will have different interfaces because they can receive different messages. This means that the information-flow programmer cannot just instantiate an Infopipe with a certain dataflow behavior, but must instantiate an Infopipe with a specific polarity configuration. The programmer must manually work out the polarity configurations of all Infopipes in the Infopipeline and check for changes whenever the connections in the pipeline are changed.

Finally, the relationship between the I/O behavior of an Infopipe and its valid polarity configurations is an open research problem. The one-inport, one-outport case has been explored empirically, but we are not aware of any systematic treatment of this issue, and the intuition of the one-inport, one-outport case does not seem to generalize to Infopipes with three or more ports.

The issues raised here may appear to contradict our previous claim that objects are a good match for Infopipes. We do not think so: we still maintain that *implementing* Infopipes as objects is a sound strategy. What needs to change is the *abstraction* used to define Infopipes. Specifically, we now believe that the notion of port polarity, while important in the implementation, should not be part of the Infopipe abstraction at all.

In this paper we show how the DirectFlow language let programmers define Infopipes without explicitly dealing with port polarity, and how a completely configured Infopipeline can be compiled to objects that interact by sending and receiving messages. Since DirectFlow programs are compiled to objects, the concept of polarity and the restrictions on polarity configurations still apply in the *implementation* of DirectFlow. However the DirectFlow language does not speak about polarity; the programmer need mention it only when specifying how the implementation of an Infopipeline interacts with other, non-Infopipe objects. Polarity hiding in DirectFlow is similar to type inference in programming languages [7, 13], where the lack of type declarations does not mean that type rules are no longer enforced. Instead it means that the burden of figuring out the types of expressions has been transferred to the compiler, leaving the programmer free to concentrate on higher-level tasks.

### 3.3 Proposals to eliminate push–pull variation

There have been several proposals to eliminate the need for both push and pull mechanisms in operating systems and programming languages; as a consequence, these proposals would also eliminate the need for polarity. Unfortunately, none of them is general and expressive enough to be completely satisfactory.

***Multiple threads*** One approach is to put each Infopipe in a separate thread and let the system scheduler sort things out. However, thread scheduling introduces additional execution overhead and makes it difficult to fine-tune the interaction between Infopipes. It also overloads the channels between Infopipes with a synchronization role. If the channels are unbuffered, each data handoff becomes a synchronization point, which introduces excessive context-switch overhead and the possibility of deadlock. If the channels are buffered, it becomes difficult to control the latency of data traveling between Infopipes, which was one of the chief motivations for the Infopipe architecture. Neither alternative is satisfactory.

***Asymmetric I/O*** The Eden system [2] explored making streams asymmetric by eliminating all active (*i.e.*, positive) output primitives (write, corresponding to our push). However, this meant that performing concurrent inputs that may block (due to lack of data) requires either polling, the use of multiple threads (problematic, see previous paragraph), or using asynchronous callbacks (active output in disguise). Reactive objects in O'Haskell [15, 16], which eliminates active input (read, pull in our terms), face similar problems.

***Lazy streams*** Lazy streams typically appear in programs written in functional languages like Scheme or Haskell, but the technique also has close ties with dataflow languages like SISAL [5]. Lazy streams work under the assumption that information-flow components are stream transformers (*i.e.*, they map one stream to another) which is not true for all Infopipes. One counterexample is a reordering buffer that always outputs the packet with the highest priority; in this

case the output stream also depends not only on the input stream, but also on the interleaving of input and output.

***Ad hoc treatment*** Some systems, such as the Click modular router [10], treats the filter component shown in Figure 2 as a special case and provide language constructs for defining these components. The work on thread transparency using coroutines [11] also focuses on the case of filters. None of these techniques generalizes to the case of Infopipes that process multiple input or output flows.

These efforts suggest that eliminating port polarity from Infopipes is a nontrivial problem.

## 4. DirectFlow for programming Infopipes

Defining Infopipes as objects simplifies their integration with existing object-oriented libraries and frameworks, but requiring programmers to define methods (which implicitly specifies the polarity configuration of an Infopipe object) leads to all the problems described in the previous section. We have designed the DirectFlow programming language to address this problem. DirectFlow provides the best of both worlds: programmers do not need to define methods and polarity configurations because Infopipes are modeled as concurrent processes, but integration with object-oriented code remains simple because Infopipes defined in DirectFlow are compiled to objects. In this section we describe the design of the DirectFlow programming language.

DirectFlow is based on Hoare's Communicating Sequential Processes (CSP) [8] and therefore bears some resemblance to Occam [18]. From the programmer's perspective, a DirectFlow program is a sequentially executed thread that uses input and output statements to conduct data transfers through ports, so port polarity is no longer visible to the programmer. Another distinguishing feature of DirectFlow is that it provides nondeterministic branches, so that a DirectFlow program can perform concurrent blocking I/O operations in the style of the POSIX select system call.

DirectFlow is not a complete programming language that stands on its own. DirectFlow contains no support for arithmetic or other data manipulation operations, so to be useful it must be integrated with a general-purpose programming language (called the *host language*). The embedded design strategy has the benefit of reducing the difficulty of both language design (there is no need to reinvent all the wheels and do a bad job) and the porting of existing programs (only the data input–output interface needs to be changed).

Figure 3 shows the syntax of DirectFlow. The most notable features of DirectFlow are the five new primitive constructs, which we now explain in detail.

**Conditional** The program evaluates the condition statement (first parameter). If the result is `true`, the program runs the first branch. If the result is `false`, the program runs the second branch.

| | | |
|---|---|---|
| *prog* | = Pipe *name* '{\|' *ports* '\|}' '\|' *vars* '\|' *stmts* .. | |
| *name* | = Infopipe name | |
| *ports* | = *port ports* \| $\epsilon$ | |
| *port* | = port identifier | |
| *vars* | = *var vars* \| $\epsilon$ | |
| *var* | = Smalltalk variable identifier | |
| *stmts* | = ( *stmt* \| *cond* \| *par* \| *alt* \| *in* \| *out* ) . *stmts* \| $\epsilon$ | |
| *stmt* | = Smalltalk statement | |
| *cond* | = cond *stmt* [*stmts*] [*stmts*] | (conditional) |
| *par* | = par [*stmts*] [*stmts*] | (parallel) |
| *alt* | = alt [*stmts*] [*stmts*] | (alternative) |
| *in* | = *var* := *port* ? | (packet input) |
| *out* | = *port* ! *stmt* | (packet output) |

**Figure 3.** Syntax of Smalltalk DirectFlow. The top level nonterminal symbol is *prog*. Port and variable declaration delimiters are quoted to avoid confusion with alternative grammar productions.

**Parallel** The program runs both branches either in parallel or in some unspecified interleaving.

**Alternative** The program runs one of the two branches. The choice between the two branches depends on the outstanding input or output requests at runtime, and the programmer has no control over it.

**Packet input** The program inputs a packet from the specified inport and stores the packet in the variable.

**Packet output** The program evaluates the statement and outputs the result to the specified outport.

The DirectFlow primitive constructs correspond closely to the ones in CSP, but DirectFlow remains less expressive than CSP because it does not allow recursion (a DirectFlow program has no way to invoke itself or another DirectFlow program). In contrast with the CSP choice operator, the `alt` construct in DirectFlow does not place any restrictions on the first statements of the two branches.

Figure 4 shows two Infopipes defined in DirectFlow-extended Smalltalk. The literals enclosed in {| |} brackets are Infopipe port names, the pair of vertical bars delimit local variables declarations, and a double period ends the program. Infopipe port names 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. We choose the concrete syntax of DirectFlow to match that of Smalltalk, but the concrete syntax can be easily changed when integrating DirectFlow with other host languages.

## 5. Relating DirectFlow programs to objects

One attractive feature of DirectFlow is that Infopipes defined in DirectFlow can be compiled into objects. In this section we give a general description of how DirectFlow programs relate to objects by investigating how an object responds to

```
Pipe duplicate {| inp outp1 outp2 |}
  | aPacket |
  aPacket := inp ?.
  par [ outp1 ! aPacket ]
      [ outp2 ! aPacket ]..

Pipe separate {| inp outp1 outp2 |}
  | aPacket |
  aPacket := inp ?.
  alt [ outp1 ! aPacket ]
      [ outp2 ! aPacket ]..
```

**Figure 4.** Two Infopipes in Smalltalk DirectFlow. The duplicate Infopipe outputs each data packet it inputs to both outports, and the separate Infopipe outputs each data packet it inputs to one of the outports depending on which downstream Infopipe requests data first.

the messages it receives, and in Section 6 we will describe the compilation algorithm in greater detail.

The analysis here does not consider the effects of inheritance, overloading, reflection, special language features such as overriding the doesNotUnderstand method in Smalltalk, or concurrency control mechanisms on the behavior of an object. We consider an object only as a reactive entity that can receive and send messages.

### 5.1 The role of methods in objects

The standard way to define an object is to define its methods. Methods play an important role in an object because they define the following features, which in turn ensure that the object satisfies the object-oriented message send protocol described in Section 3.

**The set of accepted messages** The names of methods in an object represent the set of messages the object can receive.

**The mapping from message names to code** Each method in an object has a unique name; these names define a mapping from a received message to the piece of code to execute.

The key observation here is that we can define an object in *any* programming language as long as there is a way to infer both features from the source code.

### 5.2 From DirectFlow programs to methods

Even though programmers do not specify the set of messages that a DirectFlow program accepts (after all, the design goal of DirectFlow is to free programmers from having to think in these terms), we can still recover this information by performing static analysis on the DirectFlow program in the form of constraint solving. We can also infer the method bodies corresponding to each message.

Compilation works in two stages.

| Infopipe | inp | outp1 | outp2 |
|---|---|---|---|
| | − | + | + |
| duplicate | + | − | + |
| | + | + | − |
| separate | + | − | − |

**Figure 5.** Valid polarity configurations of the duplicate and separate Infopipes in Figure 4. The three valid polarity configurations of the duplicate Infopipe generalizes the case of the filter Infopipes in Figure 2.

```
Pipe switch {| inp1 inp2 outp1 outp2 |}
  | aPacket |
  alt [ aPacket := inp1 ? ]
      [ aPacket := inp2 ? ].
  alt [ outp1 ! aPacket ]
      [ outp2 ! aPacket ]..
```

**Figure 6.** The switch Infopipe does not have a valid polarity configuration because there is no way to satisfy C1 and C2 simultaneously.

1. Compute the valid polarity configurations of the Infopipe defined in DirectFlow.
2. For each valid polarity configuration, generate a class with one method for each negative port in the configuration.

For simplicity we require the compiler to generate an Infopipe object for each valid polarity configuration of the Infopipe defined in DirectFlow. We outline the compilation stages in the rest of this section.

***Computing polarity configurations*** The polarity of each port can be either positive or negative, so an Infopipe with $n$ ports has at most $2^n$ polarity configurations. We want to find the configurations for which we can generate Infopipe objects using the characterizations described in Section 5.1. We define a *valid polarity configuration* as one in which:

C1. Each execution path contains exactly one negative port access. This constraint is necessary because invocation of a method is triggered by receiving one message.

C2. Each alt branch should access a unique negative port. This constraint ensures that each received message is mapped to a single piece of code.

C3. No negative port access should appear in only one branch of a cond construct. This constraint ensures that the set of accepted messages is fixed.

The compiler computes the valid polarity configurations by solving these constraints in the context of the DirectFlow program. Figure 5 shows the valid polarity configurations of the duplicate and the separate Infopipes. Some DirectFlow programs, such as the one shown in Figure 6, do not have a valid polarity configuration. In that case the compiler signals

| Code pattern | Rewrite result |
|---|---|
| *t1*. alt [ *s1* ] [ *s2* ] | alt [ *t1*. *s1* ] [ *t1*. *s2* ] |
| cond *p t1* [ alt *s1 s2* ] | alt [ cond *p t1 s1* ] [ cond *p t1 s2* ] |
| par *t1* [ alt *s1 s2* ] | alt [ par *t1 s1* ] [ par *t1 s2* ] |

**Figure 7.** The rewrite rules for alt lifting. This table shows only the rules where alt appears at the right hand side; the cases where alt appears at the left are analogous and thus not listed.

an error because it cannot compile the DirectFlow program into an Infopipe object.

***Generating Infopipe objects*** The code generator in the compiler takes a DirectFlow program and a valid polarity configuration and produces an Infopipe object by mapping the execution of the DirectFlow program to the invocation of a method. The code generator translates cond constructs to the standard conditional branch constructs in the host language, and alt branches to separate methods. The constraint that each alt branch accesses a unique negative port ensures that the code generator can generate a suitable push or pull method for each branch.

## 6. Compiling DirectFlow programs

In Section 5 we gave a high-level view of how DirectFlow programs relate to objects; we now explain the compilation process in greater detail. The compiler first normalizes the DirectFlow source program through a procedure called *alt lifting*, after which it computes the polarity configuration and generates code as outlined in Section 5.2.

### 6.1 alt Lifting

The purpose of alt lifting is to normalize a DirectFlow program by lifting all alt constructs to the top level. An alt-lifted program is a set of nondeterministic alternatives which we will call *branches*. Each branch corresponds to a method in the object that will be generated by the compiler; the object will rely on the message dispatch mechanism to select the appropriate branch at run time.

Figure 7 shows the rewrite rules for alt lifting. The alt construct distributes over cond, par, and the statement sequencing operator of the host language (written as a period in Smalltalk), so the alt lifting process simply performs the distribution rewriting of alt iteratively until all the alt constructs appear at the top level.

**Example: Applying alt lifting to the separate Infopipe**

We illustrate the alt lifting process by applying it to the separate Infopipe in Figure 4. The only applicable rewrite rule is to distribute alt over statement sequencing, which results in the program shown in Figure 8.

```
Pipe separate {| inp outp1 outp2 |}
  | aPacket |
  alt [ aPacket := inp ?.
        outp1 ! aPacket ]
      [ aPacket := inp ?.
        outp2 ! aPacket ]..
```

**Figure 8.** The separate Infopipe after alt lifting. The alt construct is moved to the top level by pushing the input statement into both alt branches.

### 6.2 Computing polarity configurations

The DirectFlow compiler computes the valid polarity configurations of a DirectFlow program by solving a set of constraints on the alt-lifted version of the program. We have described the constraints and the rationale behind them in Section 5.2. Instead of using an off-the-shelf constraint solver, we have developed an efficient constraint solving algorithm that takes advantage of the structure of the constraints.

If $B$ is an ordered set of all lifted alt branches in a DirectFlow program, the constraint-solving algorithm works by constructing, for each branch in $B$, the set of ports that may be negative. We start by assuming that all the ports may be negative, and then eliminate those ports for which a negative polarity would violate one of our constraints. More precisely, we proceed as follows.

S1. For each $b \in B$, compute the set $P_b$ of all the ports accessed in $b$. The set $P_b$ represents the candidate negative ports for the alt branch $b$.

S2. For each $b \in B$, compute the set $Q_b$ using the following equation:

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

$Q_b$ is the set of ports that are accessed only in branch $b$ and nowhere else. This step enforces constraint C2.

S3. For each $b \in B$, iterate through all the cond constructs. If a port $p$ is accessed in only one branch of a cond construct, eliminate port $p$ from $Q_b$. This step enforces constraint C3.

S4. We compute the negative ports in valid polarity configurations by choosing one negative port from the set $Q_b$ for each branch $b$. The set $N$ of valid configurations is given by

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

where $\Pi$ represents Cartesian product on sets. Once we know the set of negative ports in a polarity configuration, we can easily find the positive ports in the configuration. This step and step S1 together enforce constraint C1.

**Example: Polarity configurations for separate**

We number the alt branches in Figure 8 according to their order of appearance and use the program to demonstrate how to compute valid polarity configurations.

S1. $P_1 = \{$ inp, outp1 $\}$, $P_2 = \{$ inp, outp2 $\}$

S2. $Q_1 = P_1 - P_2 = \{\,\mathsf{outp1}\,\}$, $Q_2 = P_2 - P_1 = \{\,\mathsf{outp2}\,\}$

S3. No change as there is no cond construct in the program

S4. $N = Q_1 \times Q_2 = \{\ (\mathsf{outp1}, \mathsf{outp2})\ \}$

The set $N$ has only one element, so separate has one valid polarity configuration. The ports outp1 and outp2 are negative, and inp is positive.

Solving the conjunction of all applicable constraints will result in zero, one, or multiple port polarity configurations. If a DirectFlow program has no valid polarity configurations, then there is no way to compile it into an object because it does not conform to the restrictions on the behavior of objects described in Section 4.

### 6.3 Generating object code

Given a valid polarity configuration, a DirectFlow program can be compiled into an Infopipe object. The goal of the code generation process is to replace all DirectFlow primitives in the program with constructs in the host language. Here we explain the translation of each DirectFlow primitive.

**Conditional** The cond construct simply translates to the conditional branch construct in the host language.

**Parallel** Since the statement sequences in the par construct can be interleaved in any order; the compiler simply runs them in sequence. A multiprocessor implementation may choose to run the statement sequences in parallel, but we have not explored that option.

**Alternative** After alt lifting, all occurrences of alt will be at the top level, and each nondeterministic alt branch is translated to a method in the Infopipe object. The form of the method depends on the direction of the negative port; a branch with a negative inport is compiled to a push method, and a branch with a negative outport is compiled to a pull method.

**Packet input and output** The compiler replaces the ? (input) and ! (output) statements with the object-oriented message send protocol shown in Figure 1. If the subject port is positive, then ? is replaced by sending a pull message, and ! by sending a push message. If the subject port is negative, then ? is replaced by reading the argument of the push method, and ! is replaced by storing the value to be sent in a temporary variable, which is returned as the result of the invocation at the end of the pull method's body.

Figure 9 shows the result of compiling the duplicate and the separate Infopipes.

## 7. Implementation

We have implemented a compiler for DirectFlow that uses Smalltalk as the host language. The compiler is written in Haskell [9] and produces changesets that can be loaded into the Squeak Smalltalk environment.

Many Infopipes depend on hand-written code to process or to transform data packets. The *generation gap pattern*, in which hand-written code inherits from machine-generated code, is often used to integrate code from different sources [20]. Straightforward application of the pattern requires that the machine-generated class has a fixed name and interface, which is not true in our situation because there may be multiple generated classes that need to interact with the same piece of hand-written code. We solved this problem by inverting the generation gap pattern and make the machine-generated code inherit from the hand-written code. The compiler accepts the name of the hand-written class as an argument, which makes it possible to use the same DirectFlow program in different contexts without modifying the source code.

The source code of the compiler is available at the URL http://amstel.cs.pdx.edu/infopipes/artifacts/.

## 8. Related work

The work presented in this paper is part of the Infopipes project, whose goal is to develop a programming abstraction and system for the compositional construction of information-flow programs suitable for a range of application domains. This work has close ties to several other research fields, and we discuss some of the most notable ones in this section.

**Practical information-flow systems**

The Click [10] modular router system supports compositional construction of software routers. A Click router is built by connecting ports of data-processing elements, and Click distinguishes between push- and pull-processing in the same way that Infopipe implementations distinguish between positive and negative ports. Our techniques should be applicable to the Click system with little modification. Another example of a practical system is Aurora, a data stream management system that builds query graphs by connecting stream operators with data channels [1]. These examples show that compositional construction of information-flow programs is an established technology.

**Information-flow languages**

In addition to DirectFlow, languages like StreamIt [19] and Spidle [4] are also designed for developing information-flow systems. None of these languages match the rich feature set of DirectFlow: multiple input and output streams, dynamic relationship between input and output rates, and compilation to objects for efficient execution. These features are essential for building components like demultiplexers and priority-reordering buffers.

**Dataflow languages**

Dataflow programming languages like SISAL [5] typically adopt single-assignment semantics to facilitate the construc-

```
Duplicate >> inppush: arg
  | aPacket |
  aPacket := arg.
  outp1 push: aPacket.
  outp2 push: aPacket
```

```
Separate >> outp1pull
  | retval aPacket |
  aPacket := inp pull.
  retval := aPacket.
  ↑ retval
```

```
Separate >> outp2pull
  | retval aPacket |
  aPacket := inp pull.
  retval := aPacket.
  ↑ retval
```

**Figure 9.** Infopipe objects compiled from the DirectFlow programs in Figure 4. For the duplicate Infopipe, which has three valid polarity configurations, we choose the configuration with negative inport to illustrate the compilation of push methods.

tion of dataflow graphs. As we explained in Section 3.3, dataflow languages share many features with lazy streams and thus cannot support Infopipes like reordering buffers. Dataflow languages are typically designed for programming massively-parallel computers, and we have not yet considered how DirectFlow might support the development of highly parallel information-flow programs.

**Coordination languages**

Infopipes bear 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 separately addresses both the computation and communication aspects of a component. DirectFlow also demonstrates that analyzing the data input–output behavior of an Infopipe can help programmers produce code that is more reusable.

**Communicating sequential processes**

The design of DirectFlow is based on Hoare's CSP [8], and in practice DirectFlow plays a role similar to CSP-libraries like JCSP [21]. DirectFlow is designed to support only the development of information-flow programs instead of as a general-purpose programming language. We have therefore eliminated certain features such as recursion and process creation to enable DirectFlow programs to be more easily translated to objects.

**Design patterns**

The way we build information-flow programs by exchanging messages between Infopipe objects has been documented under the name *filter pattern* [6]. The PushFilter and PullFilter Infopipes appear in the filter pattern as sink and source filters, but the pattern literature 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 source and sink variants of a filter by hand.

## 9. 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 behavior in domain-specific terms.

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

The design of DirectFlow seeks a balance between language expressivity and implementation efficiency. The language allows an Infopipe to alter its input–output behavior 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 programs and to compile them to objects. Whether DirectFlow is expressive enough to support a wide range of common information-flow programs is a question that we plan to explore in our future work.

A collection of DirectFlow components is not useful unless it is possible to connect them into an Infopipeline. We have therefore implemented a companion language, InterFlow, for specifying interconnections between components, and have started to consider how to specify composite components that are built from a library of simpler ones.

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 are in the process of defining a formal semantics of DirectFlow and establishing the rate relations between streams processed by Infopipes. 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] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *International Journal on Very Large Data Bases*, 12(2):120–139, August 2003.

[2] Andrew P. Black. An asymmetric stream communication system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 4–10, New Hampshire, October 1983.

[3] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: an abstraction for multimedia streaming. *Multimedia Systems*, 8(5):406–419, December 2002.

[4] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: A DSL approach to specifying streaming applications. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering: Second International Conference*, volume 2830 of *LNCS*, pages 1–17, Erfurt, Germany, September 2003.

[5] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990. Special issue: data-flow processing.

[6] Mark Grand. *Patterns in Java: a catalog of reusable design patterns illustrated in UML*, volume 1, chapter 6, pages 155–163. John Wiley & Sons, 1998.

[7] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, Upper Saddle River, NJ, USA, 1985.

[9] Simon Peyton Jones, editor. *Haskell 98 Languages and Libraries*. Cambridge University Press, revised edition, April 2003.

[10] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[11] Rainer Koster, Andrew P. Black, Jie Huang, Jonathan Walpole, and Calton Pu. Thread transparency in information flow middleware. In Rachid Guerraoui, editor, *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 121–140, Heidelberg, Germany, November 2001. Springer-Verlag.

[12] Charles Krasic, Jonathan Walpole, and Wu-chi Feng. Quality-adaptive media streaming by priority drop. In Christos Papadopoulos and Kevin C. Almeroth, editors, *Network and Operating System Support for Digital Audio and Video, 13th International Workshop, NOSSDAV 2003*, pages 112–121, Monterey, CA, USA, June 2003.

[13] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[14] Emerson Murphy-Hill, Chuan-kai Lin, Andrew P. Black, and Jonathan Walpole. Can Infopipes facilitate reuse in a traffic application? In *Companion to the 20th OOPSLA Conference*, pages 100–101, San Diego, CA, USA, October 2005. ACM Press.

[15] Johan Nordlander and Magnus Carlsson. Reactive objects in a functional language: an escape from the evil "I". In *Proceedings of the Third Haskell Workshop*, Amsterdam, The Netherlands, June 1997.

[16] Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburtz, and Andrew Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 155–158, Washington, D.C., USA, April 2002.

[17] George A. Papadopoulos and Farhad Arbab. *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, chapter Coordination Models and Languages, pages 329–400. Academic Press, September 1998.

[18] SGS-THOMSON Microelectronics Ltd. *occam 2.1 Reference Manual*, 1995.

[19] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In R. Niegel Horspool, editor, *Compiler Construction: 11th International Conference*, volume 2304 of *LNCS*, pages 179–195, Grenoble, France, April 2002.

[20] John Vlissides. *Pattern Hatching: Design Patterns Applied*, chapter 3, pages 85–101. Addison Wesley, June 1998.

[21] P. H. Welch. Process oriented design for Java: Concurrency for all. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.