Language Support for Processing Distributed Ad Hoc Data

Kenny Q. Zhu[1] Daniel S. Dantas[1] Kathleen Fisher[2] Limin Jia[1]
Yitzhak Mandelbaum[2] Vivek Pai[1] David Walker[1]

[1] Princeton University
[2] AT&T Labs Research

Abstract

The term *ad hoc data* refers to the billions of bytes of non-standard and continuously evolving data spread across all computer systems. Such data includes server logs, distributed system performance and debugging data, telephone call records, financial data and online repositories of scientific data.

This paper presents PADS/D, a system that generates monitoring, analysis and transformation tools for distributed ad hoc data from declarative specifications. The generated tools include an archiver, a database loading system, a statistical analyzer, an alert system, an RSS feed generator, and debugging tools. In addition, the system generates libraries for application developers, including modules for parsing, printing, error management, data traversal and transformation which developers can use to create their own application-specific tools. Advanced users can build new generic tools applicable to any collection of data sources.

The PADS/D data description language allows data analysts to specify *where* their ad hoc data is located, *how* to obtain it, *when* to get it (or give up trying), and *what* preprocessing the system should do when it arrives. As its name suggests, PADS/D is layered on top of the PADS sublanguage, developed in previous research efforts, for specifying the *format* of the data sources. We illustrate the expressiveness of PADS/D by giving descriptions for several different distributed systems including CoMon, the monitoring system for PlanetLab, and a monitoring system for a web hosting service provided by AT&T. We define a formal semantics for the language, describe our implementation, and evaluate its performance. We show our system is capable of scaling to distributed systems the size of CoMon, the current monitor for Planetlab's 800+ nodes.

1. Introduction

An *ad hoc data source* is any semistructured data source for which useful data analysis and transformation tools are not readily available. The data that constitutes a single, abstract source often comes from many different concrete, physical destinations distributed across the Internet. It often becomes available over a range of times and in several, evolving formats. Before users can extract the information they need from the data, it must be fetched, archived locally for historical analysis, compressed, perhaps encrypted or anonymized, and monitored for errors or deviations from the norm.

Managing ad hoc data is a bane of the implementers of distributed systems. These systems may have hundreds or thousands of heterogeneous, distributed components. Keeping these components running smoothly is a continuous maintenance task of significant complexity. Consequently, each component in a well-designed distributed system produces a continuous stream of log files that measure its performance and health. As an example, consider the data manipulated by CoMon [24], a system designed to monitor the health, performance and security of PlanetLab [26]. Every five minutes, CoMon attempts to contact each of 842 PlanetLab nodes across 416 sites worldwide.When all is well, which it never is, each node responds with an ASCII data file in mail-header format containing information ranging from the kernel version to the uptime to the memory usage to the ID of the user with the greatest CPU utilization. CoMon archives this data in compressed form and processes the information for display to PlanetLab users. CoMon is an invaluable resource for PlanetLab users who need to monitor the health and performance of their applications or experiments.

Almost all distributed systems have (or should have) similar monitoring infrastructure. Currently, the implementors of each new distributed system usually have to build "one-off" monitoring tools, which takes an enormous amount of time and expertise to do well. A substantial part of the difficulty comes from the diversity, quality, and quantity of data these systems must handle. Implementors cannot ignore errors: they must properly handle network errors and partial disconnects of the network. They cannot ignore performance issues: data must be fetched before it vanishes from remote sites and it must be archived efficiently in ways that do not burn-out hard drives by causing them to overheat. In addition, new monitoring systems also must interact with legacy devices, legacy software and legacy data, preventing implementers from using robust off-the-shelf data management tools built for standard formats like XML.

Similar problems appear in the natural and social sciences, including biology, physics and economics. For example, systems such as BioPixie [7], Grifn [6] and Golem [27], built by computational biologists at Princeton, routinely obtain data from a number of sources scattered across the net. Often, the data is archived and later analyzed or mined for information about gene structure and regulation. Figure 1 summarizes selected distributed ad hoc data sources.

This paper describes a system that facilitates the creation, maintenance, and evolution of tools for processing ad hoc data from a wide array of distributed data sources over varying periods of time. The system, called PADS/D, is a domain-specific language in which software developers describe key aspects of the data sources they wish to monitor, including any of the following.

- **Where** the data is located. The data may be in a directory on the current machine (perhaps written by another process), at some remote location, or at a collection of locations.
- **When** to get the data. The data may need to be fetched just once (right now!) or according to some schedule.
- **How** to obtain it. The data may be accessible through standard protocols such as `http` or `ftp` or it may be created via a local or remote computation.
- **What preprocessing** the system should do when the data arrives. The data may be compressed or encrypted; privacy considerations may require the data be anonymized.

Name/*Use*	Properties
CoMon [24] *PlanetLab host monitoring*	Multiple data sets Archiving every 5 minutes From evolving set of 800+ nodes
CoBlitz [23] *File transfer system monitoring*	Multiple data sets Archiving every 3 minutes From evolving set of 800+ nodes
CoralCDN [15] *Log files from CDN monitoring*	Single Format Periodic archiving From evolving set of 250+ hosts
AT&T Arrakis *Website host monitoring*	Execute programs remotely to collect data Varied fetch frequencies
AT&T Regulus *Network monitoring*	Diverse data sources Archiving for future analysis Per minute, hour, and day fetches
AT&T Altair *Billing auditing*	Thousands of data sources Archiving and error analysis
GO DB [1] *Gene function info.*	Multiple Formats Uploads daily, weekly, monthly
BioGrid [28] *Curated gene and protein data*	XML and Tab-separated Formats multiple data sets <= 50MB each Monthly data releases
NCBI [20] *Biotechnology info.*	Links to multiple bioinformatics datasets

Figure 1. Example distributed ad hoc data sources.

- **What format** the data source arrives in. The data may be in ASCII, binary, or EBCDIC. It may be tab- or comma-separated, or it may be in the kind of non-standard format that PADS [12, 16] was designed to describe.

The PADS/D system then compiles these high-level specifications into a collection of programming libraries and end-to-end tools for distributed systems monitoring. Our current tool suite includes a number of useful artifacts, inspired by the needs we have observed in a variety of ad hoc monitoring systems:

- **An archiver** that collects distributed data on the specified schedule, archives it, and maintains a "table of contents."
- **A printer** that fetches, prints, and helps debug specifications.
- **A performance monitor** that measures fetch times and also helps debug specifications.
- **A RRD database loader** that takes the data and extracts specified pieces to load into an RRD database [21]. The data is indexed by its arrival time and supports time-based queries.
- **An accumulator** that maintains a statistical profile of the data and its error characteristics.
- **An alert system** that generates alerts based on programmable conditions.
- **A selector** that extracts and records specified subcomponents of a larger data source.
- **An RSS feed generator** that wraps raw data in the appropriate headers to create an RSS feed from diverse ad hoc data sources.

The system can generate all of these tools from PADS/D descriptions and declarative tool configuration specifications. Thus for common tasks, users can manage distributed data sources simply by writing high-level declarative specifications. It is quick and it is easy. There are relatively few concepts to learn, no complex interfaces and no tricky boilerplate to master to initialize the system or thread together tool libraries. Because there is so little "programming" involved, we refer to the act of writing simple specifications and using pre-defined tools as the *off-the-shelf* mode of use. However, to avoid sacrificing flexibility and to support extensibility, PADS/D supports two other modes of use.

The second mode is for the *single-minded implementer*, who needs to build a new application for a *specific* collection of distributed data sources. Such users need more than the built-in set of tools, and consequently the system provides support for creating new tools by automatically generating libraries for fetching data, for parsing and printing, for performing type-safe data traversal, and for stream processing using classic functional programming paradigms such as `map`, `fold` and `iterate`. These generated libraries make it straightforward to create custom tools specific to particular data sources. However, there is a steeper learning curve in this mode than in off-the-shelf mode because a variety of interfaces must be learned. The average functional programmer may find these interfaces relatively intuitive, but the computational scientist who is not interested in functional programming may prefer to stick with off-the-shelf uses.

The third mode is for the *generic programmer*. Generic programmers may observe that they (or their colleagues) need to perform some task over and over again on different data sets. Rather than writing a program specific to a particular data set, they use a separate set of interfaces supplied by the PADS/D system to write a single generic program to complete the task. For example, the RRD database loader is generic because it is possible to load data from any specified source into the RRD tool without additional "programming." The generic programming mode is the most difficult to use as it involves learning a relatively complex set of interfaces for encoding Generalized Algebraic Datatypes (GADTs) [32] and Higher-Order Abstract Syntax (HOAS). These complexities are required to encode the dependent features of PADS/D and to compensate for the lack of built-in generic programming support in OCAML. Still the reward for building generic tools is very high: as more and more such tools are built, the life of the off-the-shelf user becomes easier and easier. We have already built eight useful generic tools ourselves and will continue to build more as demand requires.

Contributions. The paper makes the following contributions:

- It outlines the design of a language for specifying the spatial, temporal and auxiliary properties of distributed ad hoc data sources. We are aware of no other research effort that has attempted to design, implement or analyze such a language.
- It provides a mechanism for the automatic generation of eight different data processing tools from high-level specifications.
- It supports three modes of use: off-the-shelf, single-minded implementer and generic programmer, thereby optimizing both flexibility and ease-of-use.
- It provides a formal denotational semantics for the language.
- It reports on the implementation experience and performance.

Outline. In the remainder of the paper, we describe the two examples we will use throughout the paper (Section 2), show how to describe these data sources in PADS/D (Section 3), describe the generated tool infrastructure and its different modes of use (Section 4), define a denotational semantics for the language (Section 5), discuss the implementation and evaluate its performance (Section 6), describe related work (Section 7), and conclude (Section 8).

2. Running Examples

The CoMon [24] system, developed at Princeton, monitors the health and status of PlanetLab [26] by attempting to fetch data from each of PlanetLab's 800+ nodes every 5 minutes. This data

```
let sites =
  [
    "http://pl1.csl.utoronto.ca:3121";
    "http://plab1-c703.uibk.ac.at:3121";
    "http://planet-lab1.cs.princeton.edu:3121"
  ]
feed simple_comon =
  base {|
    sources  = all sites;
    schedule = every 5 min, starting now,
               timeout 60.0 sec;
    format   = Comon_format.Source;
  |}
```

Figure 2. `simple_comon.fml`: Simple CoMon feed.

```
feed comon_1 =
  base {|
    sources  = any sites;
    schedule = every 1 min, lasting 2 hours;
    format   = Comon_format.Source;
  |}
```

Figure 3. `sites.fml`: Code fragment for data from one of many sites.

ranges from the node uptime to memory usage to kernel version. CoMon displays the data to users in tabular form and allows them to perform a number of simple queries to find, for instance, lightly loaded nodes, nodes with drifting clocks or nodes with little remaining disk space. CoMon also monitors nodes for various sorts of problems and generates reports of deviant machines or user programs. Finally, the data is archived so PlanetLab users can perform their own custom analyses of historical data.

AT&T provides a web hosting service. The infrastructure for this service includes a variety of hardware components such as routers, firewalls, load balancing machines, actual web servers, and databases, replicated and geographically distributed. Hence, a given web site may be distributed across a variety of machines running a variety of operating systems in a variety of locations. When a customer signs up for AT&T's hosting service, part of the contract specifies what kinds of monitoring AT&T will provide for the site. The Arrakis infrastructure provides this monitoring service. It tracks a variety of resources using a wide array of measures, including network bandwidth, packet loss, cpu utilization, disk utilization, memory usage, load averages, *etc.* For each machine in the hosting service and for each such resource, the monitoring system archives the values at regular intervals and issues alerts when the values exceed resource- and contract-specific levels. The archive is used to track long-term behavior of the service, allowing engineers to determine when more resources need to be provisioned, for example, adding cpus, memory, or disk space. It also allows engineers to understand the "normal" behavior for a particular site such as daily or seasonal cycles for a particular site.

3. PADS/D: An Informal Introduction

The PADS/D language allows users to describe streams of data and meta-data that we refer to as *feeds*. To introduce the central features of the language, we work through a series of examples drawn from the CoMon and Arrakis monitoring systems.

3.1 CoMon Feeds

Figure 2 presents our first attempt to define a simple CoMon statistics feed. This description specifies the `simple_comon` feed using the **`base`** feed constructor. The **`sources`** field indicates that data for the feed comes from all of the locations listed in `sites`. The **`schedule`** field specifies that relevant data is available from each source every five minutes, starting immediately. When trying to fetch such data, the system may occasionally fail, either because a remote machine is down or because of network problems. To manage such errors, the schedule specifies that the system should try to collect the data from each source for 60 seconds. If the data does not arrive within that window, the system should give up. Finally, the **`format`** field indicates that the fetched data conforms to the PADS/ML [16] description named `Source` defined in the file `comon_format`.

In contrast to the `simple_comon` feed, which returns data from all sites, the `comon_1` feed defined in Figure 3 uses the **`any`** constructor in the **`sources`** field to return only a single value per time slice: that of the first site to supply a complete set of data. This feature is particularly useful when monitoring the behavior of replicated systems, such as those using state machine replication, consensus protocols, or even loosely-coupled ones such as Distributed Hash Tables (DHTs) [4]. In these systems, the same data will be available from any of the functioning nodes, so receiving results from the first available node is sufficient. These kinds of monitoring systems are useful in the face of partial network unreachability or machine failure. Specifying this behavior at the language level provides a simpler implementation than network-centric approaches such as anycast [25].

The schedule for `comon_1` indicates the system should fetch data every minute for two hours, using the `lasting` field to indicate the duration of the feed. It omits the `starting` and `timeout` specifications, causing the system to use default settings for the start time and the timeout window.

The `simple_comon` example hard-codes the set of locations from which to gather performance data. In reality, the CoMon system has an Internet-addressable configuration file that contains a list of hosts to be queried, one per non-comment line. This list is periodically updated to reflect the set of active nodes in PlanetLab.

Figure 4 specifies a version of the `comon` feed that depends upon this configuration information. To do so, the description includes an auxiliary feed `nodes` that describes the configuration information: it is available from the `config_location`, it should be fetched every two minutes, and its format is described by the PADS/ML description `source` given in the file `nodelist`, which appears in Figure 5. This PADS/ML description specifies that `source` is a list of new-line terminated records, each containing a `nodeitem`. In turn, a `nodeitem` is either a `'#'` character followed by a comment string, which should be tagged with the `Comment` constructor, or a host name, which should be tagged as `Data`. The description also defines a helper function `is_node`, which returns true if the data item in question is a host name rather than a comment. Given this specification, the `nodes` feed logically yields a list of host names and comments every two minutes. In fact, because of the possibility of errors, the feed actually delivers a *list option* every two minutes: `Some` if the list is populated with data, `None` if the data was unavailable at the given time-slice.

Using the `nodes` specification, we define the `comon` feed as a *dependent* feed: each `nodelist` in the `nodes` feed defines a collection of sources for the `comon` feed. The `comon` **`sources`** specification processes the `nodelist` to manage errors and strip out comment fields. The code that handles this processing illustrates that the PADS/D domain-specific language is embedded in OCAML. We use OCAML terms where necessary to specify simple transformations. In particular, the `current` function checks if the `nodelist` is `None`, signaling a fetching error, in which case it uses the most recently cached list of nodes instead. The **`source`** specification filters out comment fields, and then converts the host names to URLs with the required port using the auxiliary function

```
(* Ocaml helper values and functions *)
let config_location =
  ["http://summer.cs.princeton.edu/status/ \
    tabulator.cgi?table=slices/ \
    table_princeton_comon&format=nameonly"]

let makeURL (Nodelist.Data x) =
     "http://" ^ x ^ ":3121"

let old_locs = ref []
let current list_opt =
  match list_opt with
    Some l ->  old_locs := l; l
  | None   -> !old_locs

(* Feed of nodes to query *)
feed nodes =
  base {|
    sources  = all config_location;
    schedule = every 2 min;
    format   = Nodelist.Source;
  |}

(* Dependent CoMon feed of node statistics *)
feed comon =
  foreach nodelist in nodes
  create
    base {|
      sources  = all (List.map makeURL
                     (List.filter Nodelist.is_node
                     (current nodelist)));
      schedule = once, timeout 60.0 sec;
      format   = Comon_format.Source;
    |}
```

Figure 4. `comon.fml`: Uses feed of node locations to drive data collection.

```
ptype nodeitem =
  Comment of '#' * pstring_SE(peor)
| Data of pstring_SE(peor)

let is_node item =
  match item with
  Data _ -> true
  | _ -> false

ptype source =
    nodeitem precord plist (No_sep, No_term)
```

Figure 5. `nodelist.pml`: PADS/ML description for CoMon configuration files, which contain one host name per non-commented line.

`makeURL`. The **`schedule`** for this CoMon feed is `once` (with a timeout of sixty seconds) because we want to collect the data for each host in a given `hostlist` just once. The **`foreach ... create`** construct merges the resulting data from each machine into a single feed. As before, the format of data fetched from each node matches the description `Comon_format.Source`.

With this specification, we expect to get data from all the active machines listed in the configuration file every two minutes. We further expect the system to notices changes in the configuration file within two minutes.

The previous examples all showcased feeds that contained a single type of data. PADS/D also provides a datatype mechanism that allows us to construct compound feeds containing data of different sorts. As an example where such a construct is useful, the CoMon system includes a number of administrative data sources. One example is a collection of node profiles, collecting the domain name, IP address, physical location, *etc.*, for each node in the cluster. A second example is a list of authentication information for logging into the machines. These two data sources have different formats, locations, and update schedules, but system administrators want to keep a combined archive of the administrative information present in these sources. If `sites_mime` is a feed description of the profile information and `sites_keyscan_mime` is a feed of authentication information, then the declaration

```
feed sites =
    Locale of sites_mime
  | Keyscan of sites_keyscan_mime
```

creates a feed with elements drawn from each of the two feeds. The constructors `Locale` and `Keyscan` tag each item in the compound feed to indicate its source.

3.2 Arrakis Example

We now shift to an example drawn from AT&T's Arrakis project. Like the earlier CoMon example, the `stats` feed in Figure 6 monitors a collection of machines described in a configuration file. Before we discuss the `stats` feed itself, we first explain some auxiliary feeds that we use in the definition of the `stats` feed.

The `raw_hostLists` description has the same form as the `nodes` feed we saw earlier, except it draws the data from a local file once a day. We use a *feed comprehension* to define a clean version of the feed, `host_lists`. In the comprehension, the built-in predicate `is_good` verifies that no errors occurred in fetching the current list of machines `hl`, as would be expected for a local file. The function `get_hosts` takes `hl` and uses the built-in function `get_good` to unwrap the payload data from the error infrastructure, an operation that is guaranteed to succeed because of the `is_good` guard. The function `get_hosts` then selects the host name entries and unwraps them to produce a list of unadorned host names.

We next define a feed generator `gen_stats` that yields an integrated feed of performance statistics for each supplied host. In more detail, when given a host `h`, `gen_stats` creates a five minute schedule with a one minute timeout. It then uses this schedule to describe a compound feed, which pairs two base feeds: the first uses the Unix command `ping` to collect network statistics about the route to `h` while the second performs a remote shell invocation using `ssh` to gather statistics about how long the machine has been up. Both of these feeds use the **`proc`** constructor in the **`sources`** field to compute the data on the fly, rather than reading it from a file. The argument to **`proc`** is a string that the system executes in a freshly constructed shell. The pairing constructor for feeds takes a pair of feeds and returns a feed of pairs, with elements sharing the same scheduled fetch-time being paired. This semantics conveniently produces a compound feed that for each host returns a pair of its ping and uptime statistics, grouping together the information for each host. Of course, the full Arrakis monitoring application uses many more tools than just ping and uptime to probe remote machines; the full feed description has many more branches than this simplified version.

Finally, we define the feed `stats`. The most interesting piece of this declaration is the *list feed comprehension*, given in square brackets, that we use to generate a feed of lists. Given a host list `hl`, the right-hand side of the comprehension considers each host `h` from `hl` in turn. The left-hand side of the comprehension uses the `gen_stats` feed generator to construct a feed of the statistics for `h`. The list feed comprehension then takes this collection of statistics feeds and converts them into a single feed, where each entry is a list of the statistics for the machines in `hl` at a particular

```
let config_locations =
  [("file:///arrakis/config/machine_list")];

feed raw_hostLists =
  base {|
   sources  = all config_locations;
   schedule = every 24 hours;
   format   = Hosts.Source;   |}

let get_host (Hosts.Data h) = h
let get_hosts hl =
    List.map get_host
     (List.filter Hosts.is_node (get_good hl))

feed host_lists =
  {| get_hosts hl | hl <- raw_hostLists,
                     is_good hl |}

feed gen_stats (h) =
  let s = every 5 mins, timeout 1 min in
  (
   base {|
     sources  =  proc ("ping -c 1 " ^ h);
     format   = Ping.Source;
     schedule = s; |},
   base {|
     sources  = proc ("ssh " ^ h ^ " uptime");
     format   = Uptime.Source;
     schedule = s; |}
  )

feed stats =
  foreach hl in host_lists update
     [ gen_stats h | h <- hl ]
```

Figure 6. `arrakis.fml`: Simplified version of Arrakis feed.

scheduled fetch-time. We call each such entry a *snapshot* of the system. The resulting feed makes it easy for down-stream users to perform actions over snapshots, relieving them of the burden of having to implement their own multi-way synchronization.

Given the list feed comprehension, the **`foreach...update`** construct generates a feed of snapshots from the feed of host lists. Whenever a new host list arrives, the **`foreach...update`** construct terminates the snapshot feed from the old host list and starts generating a new snapshot feed from the new host list.

Note the difference between the **`foreach...create`** construct from the CoMon example and the **`foreach...update`** construct. The create form generates a collection of feeds and merges their contents into a single all-inclusive feed. The update form generates a collection of feeds and produces a single feed by concatenating the collection, stopping one feed as soon as the next is generated. We have found the create form to be useful when the actual arrival times of the argument feed are regular because the regularity means we can give a finite schedule for the dependent feed. In contrast, the update form is useful when the argument feed is irregular and we must give an infinite schedule for the dependent feed to ensure we get the desired values. We define precise semantics for the create and update forms in Section 5.

4. PADS/D: Working with Feeds

4.1 The "Off the Shelf" User

The PADS/D system provides a suite of "off-the-shelf" tools to help users cope with standard data administration needs. After writing a PADS/D description, users can customize these tools by writing simple *configuration files*, such as shown in Figure 7. Each configuration file includes a feed declaration header and a sequence

```
feed comon.fml/comon

tool feedaccum
{
  minalert  = true;
  maxalert  = true;
  lesssig   = 3;
  moresig   = 3;
  slicesize = 10;
  slicefile = "slice.acc";
  totalfile = "total.acc";
}

tool rss
{
  title = "CoMon Memory RSS";
  link  = "http://www.comon.org/memory-rss.xml";
  desc  = "CoMon Memory Usage Information";
  path  = "<top>.[?].Mem_info";
}
```

Figure 7. `comon.tc`: Example tool configuration file.

```
=====================================================
Summary of network transmission errors
=====================================================
ErrCode: 1      ErrMsg: Misc HTTP error Count: 12
ErrCode: 5      ErrMsg: Bad message     Count: 27
ErrCode: 6      ErrMsg: No reply        Count: 2

=====================================================
Top 10 locations with most network errors
=====================================================
Loc: http://planetlab01.cnds.unibe.ch:3121    Count: 2
Loc: http://pepper.planetlab.cs.umd.edu:3121  Count: 2
Loc: http://planetlab3.cs.uchicago.edu:3121   Count: 2
... omitted ...
```

Figure 8. `comon.acc`: Fragment of accumulator output.

```
path :: =
  "<top>"
| path.ID  (field/variant name)
| path.INT (branch number (from 1) of a tuple)
| path.[?] (any one element of array/table)
| path.[*] (all elements of array/table)
| path.[INT] (nth element of array (from 0))
| path.[Key] (a table entry indexed by the Key)
```

Figure 9. Selector path language.

of tool specifications. The header specifies the path to the feed description file (`comon.fml`) and the name of the feed to be created (`comon`). Each tool specification starts with the keyword `tool` followed by the name of the tool (*e.g.*, `feedaccum` and `rss`). The body of each tool specification lists name-value pairs, where values are OCAML expressions. Some attributes are optional, and the compiler fills in a default value for every omitted attribute. PADS/D compiles a configuration file into an OCAML program that creates and archives the specified feed, configures the specified tools, and applies them to the feed in parallel. In the following paragraphs, we describe the tools we have implemented.

Archiver. The archiver saves the data fetched by a feed in the local file system, organizing it according to the structure of the feed, with one directory per base feed. It places a catalog in each directory documenting the source of the data, its scheduled arrival time and the actual arrival time. The archiver will optionally compress files.

Printer. The printer outputs the contents of a feed. If configured to print to a single file, the tool concatenates successive items with a

specified separator. If configured to print to multiple files, it outputs the contents of each base feed into a separate file.

Profiler. The profiler monitors performance, reporting throughput, average network latency and average system latencies over a period of time. Users can specify in the configuration when to profile and for how long. We used this tool to produce some of the experimental results in Section 6.4.

Accumulator. The accumulator maintains statistical profiles for feeds, including their error characteristics. For numeric data, the accumulator keeps aggregates such as averages, max/min values and standard deviations. For other data (*e.g.*, strings, URLs and IP addresses), it keeps the frequency of the top N most common values. For all data, it tracks error rates, the most common error values and their sources. The user can configure the accumulator to profile entire feeds at once, or incrementally. The latter is useful for infinite feeds, because it allows users to continuously monitor feeds and compare their current behavior with historical statistics. The accumulator can output either plain text or XML. Figure 8 shows portions of accumulator output for the CoMon example.

Alerter. The alerter allows users to register boolean functions which generate notifications when they evaluate to false on feed items. The tool appends these notifications to a file, which can be piped into other tools. The system provides a library of common alerters such as exceeding max/min thresholds or deviating from the norm (*i.e.*, trigger an alert when a selected value strays more than k standard deviations from its historical value). Users can supply their own conditions by giving arbitrary OCAML predicates in the configuration file.

Database loader. This tool allows users to load numerical data from a feed into a Round Robin Database (RRD) [21]. Users specify a function to transform feed items into numeric values and RDD parameters such as data source type and sampling rate. RRD indexes the data by arrival time. It periodically discards old data to make space for new. The tool supports time-indexed queries and displays historical data as graphs.

Selector. The selector allows users to choose subcomponents of feed elements using path expressions. It returns a feed of the selected subcomponents, which may then be fed into other tools. Figure 9 shows the path expression language, partly inspired by XPATH [8].

RSS feed generator. The RSS feed generator converts a PADS/D feed to an RSS feed. Users specify the title, link (source), description, update schedule and contents of the RSS feed. Content specifications are written in the path expression language.

4.2 The Single-Minded Implementer

In addition to the off-the-shelf tools, PADS/D includes an API for manipulating generated feeds. The API provides users with a feed abstraction representing a potentially infinite series of elements. This abstraction is related to that of a lazy list, but extends it with support for data timing and provenance information. Therefore, we model the feed API on the list APIs of functional languages but provide two levels of abstraction. One level allows users to manipulate feeds like any lazy list of data elements (ignoring where they come from), while the other exposes the metadata as well.

For example, consider PlanetLab users looking for a desirable set of nodes on which to run their experiments. They can use the API generated from the CoMon description to monitor PlanetLab for a few minutes to find the least loaded nodes. Figure 10 shows an OCAML code fragment that collects the nodes with the lowest average loads over 10 minutes and then prints them. We omit the details for maintaining the table of top values, as it is orthogonal to our discussion. First, we use `Feed.split_every` to split the feed when 600 seconds (10 minutes) have elapsed. Then, we use `Feed.map` to project the load data from the CoMon elements.

```
let (sample, _) = Feed.split_every 600. comon in
let select_load = function
    Some {Comon_format.Source.
          loads = (_, load::_)} -> Some load
  | None -> None in
let loads    = Feed.map select_load sample in
let load_tbl = Feed.fold update empty_tbl loads
in  print_top 10 load_tbl
```

Figure 10. Code fragment finding least loaded PlanetLab nodes.

```
let update_m  tbl adata =
  let meta = Feed.get_meta adata in
  let data = Feed.get_contents adata in
  match meta, data with
    (h, Some basemeta), Some load ->
      let location = Meta.get_link basemeta in
      update tbl (location, data)
  | _ -> tbl  (* no change to tbl *) in
let load_tbl = Feed.fold_m update_m empty_tbl loads
in print_top_with_loc 10 load_tbl
```

Figure 11. Revised code fragment involving provenance metadata.

Finally, we use `Feed.fold` to collect the data into a table. Function `update` adds an entry to the table, and `empty_tbl` is the initially empty table. After filling the table, `print_top 10` processes each node's loads and prints the ten lowest average loads.

However, this solution is not enough – the CoMon data format does not include the node location in the data, so the code in Figure 10 cannot report the names of the nodes with the lowest average loads. In such situations, provenance metadata is essential. We therefore replace the last two lines of Figure 10 with the code in Figure 11, to exploit metadata. First, we sketch an `update_m` (update with meta) function that uses metadata to associate a location with every load in the table. It relies on the `Meta` module, which we provide to facilitate management of metadata from the feed. Next, we show a call to the lower-level fold, `fold_m` (fold with meta), which passes the data with its metadata to the folding function. Last, the call `print_top_with_loc 10` prints the ten lowest average loads with their locations.

It should be clear from these examples that the single-minded implementer has a number of new interfaces to master relative to the off-the-shelf user, but gains a correspondingly higher degree of flexibility and can still write relatively concise programs.

4.3 The Generic Programmer

Occasionally, users might want to develop functions that can manipulate *any* feed. Often, such functions can be written as parametric in the type of the feed element, much like the feed library functions discussed above. However, the behavior of many feed functions depends on the structure of the feed and its elements. Such functions can be viewed as *interpretations* of feed descriptions. To support their development, we provide a framework for writing feed interpreters.

Two core examples of feed interpretations are the feed creator and the feed accumulator. The behavior of these tools depends essentially on the structure of the feed. Functions like these require as input a runtime representation of the feed, complete with the details of the feed description that they represent. The obvious choice for representing feed descriptions in OCAML is a datatype. However, standard OCAML datatypes are not sufficiently typeful to express the types of many generic feed functions. For example, the feed creation function has the type: `feed_create : 'a prefeed -> 'a feed` where the type `'a prefeed` is an AST of a feed description and feed elements have type `'a`. This limitation of datatypes has been widely

(host-language base types)
b ::= `bool` | `string` | `loc` | `time` | `sched`

(host-language types)
τ ::= b | τ `option` | $\tau_1 * \tau_2$ | $\tau_1 + \tau_2$ | τ `list` | $\tau_1 \rightarrow \tau_2$

(host-language values)
v ::=

`false` | `true`	booleans
w | ℓ | t | s	strings, locations, times, schedules
`None` | `Some` v	optional values
(v_1, v_2)	pairs
`inl` v | `inr` v	sum values
$[v_1, \ldots, v_n]$	list values
$\lambda x{:}\tau.e$	function values

(host-language expressions)
e ::=

x	variables
v	data values
`None` | `Some` e	option expressions
...	more typed lambda expressions

Figure 12. Host Language Syntax.

discussed in the literature, and various solutions have been proposed [11, 31, 32, 34]. We have chosen to represent our AST using a variant of the Mogensen-Scott encoding [18, 30] which exploits higher-order abstract syntax to encode variable binding in feed descriptions. This implementation strategy exploits OCAML's module system to type the encodings in F_ω. Our earlier work on PADS/ML [11] exploited a similar strategy, but there we only sought to encode the OCAML type of the data, not the entire PADS/ML description, which is where higher-order abstract syntax becomes useful.

The result of our work is that developers can interpret feed-description representations by case analysis on their structure, while still achieving the desired static guarantees. Moreover, we have successfully used this framework to develop *all* of the tools presented in this paper, including the feed creator. The compiler only infers appropriate type declarations from feed descriptions and compiles the feed syntax into our representations. However, as one might expect, interfaces using higher-order abstract syntax and Mogensen-Scott encodings are one step more complex than those involving the more familiar maps and folds. Consequently, the learning curve for the generic programmer is one step steeper than the curve for the single-minded implementor, and two (or perhaps ten) steps steeper than the curve for the off-the-shelf user.

5. PADS/D Semantics

Developing a formal semantics for PADS/D has been an integral part of our language design process. We have used the semantics to communicate our ideas precisely and to explore the nuances of design decisions. Furthermore, the semantics provides users with a tool to reason about the feeds resulting from PADS/D descriptions, including subtleties related to synchronization, timeouts and errors.

To express locations, schedules, and constraints, the feed calculus depends upon a *host language*, which we take to be the simply-typed lambda calculus. Figure 12 presents its syntax, which includes a collection of constants to simplify the semantics: strings (w), locations (ℓ), times (t), and schedules (s). We assume times may be added and compared and we let ∞ represent a time later than all others. We treat schedules as sets of times and use the notation $t \in s$ to refer to a time t drawn from the set s. We use a similar notation to refer to elements of a list. The host language also includes standard structured types such as options, pairs, sums, lists and functions. We omit the typing annotations from lambda expressions when they can be reconstructed from the context.

(feed types)
σ ::= τ | τ `option` | $\sigma_1 * \sigma_2$ | $\sigma_1 + \sigma_2$ | σ `list`

(core feed spec)
C ::=

{ `src` $= e_1$;	source specification
`sched` $= e_2$;	schedule specification
`win` $= e_3$;	time-out window specification
`pp` $= e_4$;	pre-processor
`format` $= e_5$; }	format specification

(feed specs)
F ::=

`all` C	all sources
`any` C	one of multiple sources
$\emptyset$	empty feed
$[e_1 \mid x \in e_2]$	computed feed
$\{\|e \mid x \leftarrow F\|\}$	feed comprehension
`filter` F `with` e	filter feed
`let` $x = e_1$ `in` F_2	let feed
$F_1 \cup F_2$	union feed
$F_1 + F_2$	sum feed
(F_1, F_2)	synchronous pair
$x{:}F_1 * F_2$	dependent continuous pair
$x{:}F_1 ** F_2$	dependent local pair
`foreach*` x `in` F_1 `create` F_2	for each x create continuous F_2
`foreach**` x `in` F_1 `update` F_2	for each x update local F_2
$[F \mid x \leftarrow e]$	list comprehension feed

Figure 13. Feed Language Syntax.

5.1 Feed Syntax and Typing

The abstract syntax for our feed calculus and its typing rules appear in Figures 13 and 14, respectively. The feed typing judgment has the form $\Gamma \vdash F : \sigma$ `feed`, which means that in context Γ mapping variables to host language types τ, feed F produces a sequence of values of type σ. As shown in Figure 13, we define σ in terms of host language types, stratified to facilitate the proof of semantic soundness. Feed typing depends upon a standard judgment for simply typing lambda calculus expressions: $\Gamma \vdash e : \tau$. We discuss the syntax and typing for each construct in turn.

Core feeds express the underlying structure of base feeds, describing the data sources (`src`), schedule (`sched`), window (`win`), preprocessing function (`pp`), and file format (`format`). The source field may contain pseudo-locations that model the `proc` form found in the implementation. Instead of expressing time-out conditions in the schedule as we did in the source language, the calculus requires such conditions be specified in the window field, which slightly simplifies the semantics. The preprocessor and the format parser both map values with option type to option type, where the value `None` indicates a networking or data formatting error. (For the sake of simplicity, we do not model the variety of error codes that the implementation supports.) Consequently, if the format intuitively describes values of type τ, the feed will return a sequence of values of type τ `option`, allowing for the possibility of errors. The typing

for core feeds reflects our choice to have their semantics be a pair of the underlying schedule and the actual feed elements.

The feed `all` C selects all the data from the core feed C, while `any` C selects a representative good value for each time in the schedule for C. It inserts a `None` if no such value exists.

The empty feed ($\emptyset$) contains no elements and has polymorphic type a là the empty list. The computed feed ($[e_1 \mid x \in e_2]$) allows programmers to generate a feed with schedule e_2 and elements $(\lambda x.e_1)\ t$, where t is drawn from the schedule. Likewise, the feed comprehension ($\{|e \mid x \leftarrow F|\}$) creates a feed with elements $(\lambda x.e)\ v$ when v is an element of F. The feed `filter` F `with` e eliminates elements v from F when $e\ \ v$ is `false`. Let feeds `let` $x = e_1$ `in` F_2 provide a convenient mechanism for binding intermediate values. The union feed merges two feeds with the same type into a single feed. In contrast, the sum feed takes two feeds with (possibly) different types and injects the elements of each feed into a sum before merging the results into a single feed.

The calculus also contains three different pair constructors, each providing a different way to combine elements from the subcomponent feeds. The first pair, written (F_1, F_2), is a *synchronous pair*. Elements of F_1 are paired with elements of F_2 that are *scheduled* at exactly the same time, regardless of when those elements actually *arrive*. Synchronous pairs are most useful when the underlying subcomponent feeds share the same schedule, as in the Arrakis example from Section 3. Synchronous pairs lack a dependent variant, however, because in our domain it is not sensible to schedule the acquisition of an element that depends upon another element scheduled at the same time. To express dependencies, we use two forms of dependent pair, *continuous* and *local*. In the continuous variant, each element x of F_1 is paired with all elements of the feed F_2 that depend on x. In the local variant, each element x of F_1 is paired with all elements of the feed F_2 *until the next element* y *of* F_1 *is scheduled*. The denotational semantics, presented in the next subsection, makes this idea precise. Local pairing enables a paradigm in which programmers define an infinite feed F_2 that gets truncated and regenerated whenever a new element in F_1 is scheduled.

The final elements of the calculus include the `foreach` feeds and the list comprehension feed. Intuitively, a `foreach` is identical to a dependent pair in which the first element of the pair is omitted from the data representation. Consequently, there are two forms of `foreach` – one for each form of dependent pair. The notation $*$ or $**$ serves as a syntactic mnemonic for the connection. The list comprehension feed generates a feed of lists where each element of the list is scheduled at the same time. It is akin to the synchronous pair operation.

5.2 Feed Semantics

We give the semantics of our formal feed language in a denotational style. The principal semantic functions are $\mathcal{C}[\![C]\!]_{E\,U}$ and $\mathcal{F}[\![F]\!]_{E\,U}$, defining core feeds and feeds, respectively. In these definitions, E is an *environment* mapping variables to values and U is a *universe* mapping pairs of schedule time and location to arrival time and a string option representing the actual data. Intuitively, the universe models the network. When $U(t_1, \ell) = (t_2, \texttt{Some}\ w)$, the interpretation is that if the run-time system requests data from location ℓ at time t_1 then string data w will be returned at time t_2. The time t_2 must be later than t_1. When $U(t_1, \ell) = (\infty, \texttt{None})$, networking errors have made location ℓ unreachable.

The semantic functions yield a set of metadata/data pairs, where the metadata is coded as follows.

m	$::=$	(t, ℓ)	base metadata
	$\mid$	$(t, (m_1, m_2))$	pair metadata
	$\mid$	$(t, \texttt{inl}\ m)$	sum metadata
	$\mid$	$(t, \texttt{inr}\ m)$	sum metadata
	$\mid$	$(t, [m_1, \ldots, m_k])$	list metadata

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{loc list} \quad \Gamma \vdash e_2 : \texttt{sched} \quad \Gamma \vdash e_3 : \texttt{time} \\ \Gamma \vdash e_4 : \texttt{string option} \rightarrow \texttt{string option} \\ \Gamma \vdash e_5 : \texttt{string option} \rightarrow \tau\ \texttt{option}\end{array}}{\begin{array}{l}\Gamma \vdash \{\texttt{src} = e_1;\ \texttt{sched} = e_2;\ \texttt{win} = e_3; \\ \quad \texttt{pp} = e_4;\ \texttt{format} = e_5;\} : \texttt{sched} * (\tau\ \texttt{option feed})\end{array}} \ (\textit{t-core})$$

$$\frac{\Gamma \vdash C : \texttt{sched} * (\sigma\ \texttt{feed})}{\Gamma \vdash \texttt{all}\ C : \sigma\ \texttt{feed}} \ (\textit{t-all})$$

$$\frac{\Gamma \vdash C : \texttt{sched} * (\sigma\ \texttt{feed})}{\Gamma \vdash \texttt{any}\ C : \sigma\ \texttt{feed}} \ (\textit{t-any})$$

$$\frac{}{\Gamma \vdash \emptyset : \sigma\ \texttt{feed}} \ (\textit{t-empty})$$

$$\frac{\Gamma \vdash e_2 : \texttt{sched} \quad \Gamma, x{:}\texttt{time} \vdash e_1 : \tau}{\Gamma \vdash [e_1 \mid x \in e_2] : \tau\ \texttt{feed}} \ (\textit{t-compute})$$

$$\frac{\Gamma \vdash F : \sigma\ \texttt{feed} \quad \Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \{|e \mid x \leftarrow F|\} : \tau\ \texttt{feed}} \ (\textit{t-comph})$$

$$\frac{\Gamma \vdash F : \sigma\ \texttt{feed} \quad \Gamma \vdash e : \sigma \rightarrow \texttt{bool}}{\Gamma \vdash \texttt{filter}\ F\ \texttt{with}\ e : \sigma\ \texttt{feed}} \ (\textit{t-filter})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ F_2 : \sigma_2\ \texttt{feed}} \ (\textit{t-let})$$

$$\frac{\Gamma \vdash F_1 : \sigma\ \texttt{feed} \quad \Gamma \vdash F_2 : \sigma\ \texttt{feed}}{\Gamma \vdash F_1 \cup F_2 : \sigma\ \texttt{feed}} \ (\textit{t-union})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash F_1 + F_2 : \sigma_1 + \sigma_2\ \texttt{feed}} \ (\textit{t-sum})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash (F_1, F_2) : \sigma_1 * \sigma_2\ \texttt{feed}} \ (\textit{t-synch-pair})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma, x{:}\sigma_1 \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash x{:}F_1 * F_2 : \sigma_1 * \sigma_2\ \texttt{feed}} \ (\textit{t-cont-pair})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma, x{:}\sigma_1 \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash x{:}F_1 ** F_2 : \sigma_1 * \sigma_2\ \texttt{feed}} \ (\textit{t-local-pair})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma, x{:}\sigma_1 \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash \texttt{foreach}*\ x\ \texttt{in}\ F_1\ \texttt{create}\ F_2 : \sigma_2\ \texttt{feed}} \ (\textit{t-foreachcont})$$

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \texttt{feed} \quad \Gamma, x{:}\sigma_1 \vdash F_2 : \sigma_2\ \texttt{feed}}{\Gamma \vdash \texttt{foreach}**\ x\ \texttt{in}\ F_1\ \texttt{update}\ F_2 : \sigma_2\ \texttt{feed}} \ (\textit{t-foreachlocal})$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{list} \quad \Gamma, x{:}\tau \vdash F : \sigma\ \texttt{feed}}{\Gamma \vdash [F \mid x \leftarrow e] : \sigma\ \texttt{list feed}} \ (\textit{t-listf})$$

Figure 14. Feed Language Typing.

$$
\begin{array}{lcl}
\mathcal{C}[\![\{\texttt{src} = e_{src}; & = & (S, \{((t,\ell), \mathcal{E}[\![e_f\ (U'(\ell,t))]\!]_E)) \mid t \in S \text{ and } \ell \in \mathcal{E}[\![e_{src}]\!]_E\}) \\
\quad \texttt{sched} = e_{sched}; & & \quad \text{where} \\
\quad \texttt{win} = e_{win}; & & \qquad S = \mathcal{E}[\![e_{sched}]\!]_E \\
\quad \texttt{pp} = e_{pp}; & & \qquad \texttt{timeout} = \lambda(x_t, (x_{at}, x_s)).\texttt{if}\ x_{at} \leq x_t + \mathcal{E}[\![e_{win}]\!]_E\ \texttt{then}\ x_s\ \texttt{else}\ \texttt{None} \\
\quad \texttt{format} = e_f; \}]\!]_{E\,U} & & \qquad U' = \lambda(x_\ell, x_t).\mathcal{E}[\![e_{pp}]\!]_E\ (\texttt{timeout}\ (x_t, U(x_\ell, x_t)))
\end{array}
$$

$$
\mathcal{F}[\![\texttt{all}\ C]\!]_{U\,E} = A \quad \text{where } (S, A) = \mathcal{C}[\![C]\!]_{U\,E}
$$

$$
\begin{array}{lcl}
\mathcal{F}[\![\texttt{any}\ C]\!]_{U\,E} & = & \{i_t \mid t \in S\} \\
& & \quad \text{where} \quad (S, A) = \mathcal{C}[\![C]\!]_{E\,U} \\
& & \qquad\qquad A_t = \{(m, \texttt{Some}\ v) \mid (m, \texttt{Some}\ v) \in A \text{ and } m.t = t\} \\
& & \qquad\qquad i_t = \begin{cases} \mathsf{select\_one}(A_t) & \text{if} \quad |A_t| > 0 \\ ((t, \mathsf{nowhere}), \texttt{None}) & \text{if} \quad |A_t| = 0 \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}[\![\emptyset]\!]_{E\,U} & = & \{\ \} \\
\mathcal{F}[\![[e_1 \mid x \in e_2]]\!]_{E\,U} & = & \{((t, \mathsf{nowhere}), \mathcal{E}[\![(\lambda x.e_1)\ t]\!]_E) \mid t \in \mathcal{E}[\![e_2]\!]_E\} \\
\mathcal{F}[\![\{|e \mid x \leftarrow F|\}]\!]_{E\,U} & = & \{((m.t, \mathsf{nowhere}), \mathcal{E}[\![(\lambda x.e)\ v]\!]_E) \mid (m, v) \in \mathcal{F}[\![F]\!]_{E\,U}\} \\
\mathcal{F}[\![\texttt{filter}\ F\ \texttt{with}\ e]\!]_{E\,U} & = & \{(m, v) \mid (m, v) \in \mathcal{F}[\![F]\!]_{E\,U} \text{ and } \mathcal{E}[\![e\ v]\!]_E = \texttt{true}\} \\
\mathcal{F}[\![\texttt{let}\ x = e_1\ \texttt{in}\ F_2]\!]_{E\,U} & = & \mathcal{F}[\![F_2]\!]_{(E, x \mapsto \mathcal{E}[\![e_1]\!]_E)\,U} \\
\mathcal{F}[\![F_1 \cup F_2]\!]_{E\,U} & = & \mathcal{F}[\![F_1]\!]_{E\,U} \bigcup \mathcal{F}[\![F_2]\!]_{E\,U} \\
\mathcal{F}[\![F_1 + F_2]\!]_{E\,U} & = & \{((m.t, \texttt{inl}\ m), \texttt{inl}\ v) \mid (m, v) \in \mathcal{F}[\![F_1]\!]_{E\,U}\} \bigcup \{((m.t, \texttt{inr}\ m), \texttt{inr}\ v) \mid (m, v) \in \mathcal{F}[\![F_2]\!]_{E\,U}\} \\
\mathcal{F}[\![(F_1, F_2)]\!]_{E\,U} & = & \{((m_1.t, (m_1, m_2)), (v_1, v_2)) \mid (m_1, v_1) \in \mathcal{F}[\![F_1]\!]_{E\,U} \text{ and } (m_2, v_2) \in \mathcal{F}[\![F_2]\!]_{E\,U} \text{ and } m_1.t = m_2.t\} \\
\mathcal{F}[\![x{:}F_1 * F_2]\!]_{E\,U} & = & \{(m_2.t, (m_1, m_2)), (v_1, v_2)) \mid (m_1, v_1) \in \mathcal{F}[\![F_1]\!]_{E\,U} \text{ and} \\
& & \qquad (m_2, v_2) \in \mathcal{F}[\![F_2]\!]_{(E, x \mapsto v_1)\,U} \text{ and } m_2.t > m_1.t\} \\
\mathcal{F}[\![x{:}F_1 ** F_2]\!]_{E\,U} & = & \{(m_2.t, (m_1, m_2)), (v_1, v_2)) \mid (m_1, v_1) \in \mathcal{F}[\![F_1]\!]_{E\,U} \text{ and} \\
& & \qquad (m_2, v_2) \in \mathcal{F}[\![F_2]\!]_{(E, x \mapsto v_1)\,U} \text{ and } m_2.t > m_1.t \\
& & \qquad ((m_1', v_1') \in \mathcal{F}[\![F_1]\!]_{E\,U} \text{ implies } (m_1'.t \leq m_1.t \text{ or } m_1'.t > m_2.t))\} \\
\mathcal{F}[\![\texttt{foreach}*\ x\ \texttt{in}\ F_1 & = & \{(m_2, v_2) \mid (t, (m_1, m_2)), (v_1, v_2)) \in \mathcal{F}[\![x{:}F_1 * F_2]\!]_{E\,U}\} \\
\qquad \texttt{create}\ F_2]\!]_{E\,U} & & \\
\mathcal{F}[\![\texttt{foreach}**\ x\ \texttt{in}\ F_1 & = & \{(m_2, v_2) \mid (t, (m_1, m_2)), (v_1, v_2)) \in \mathcal{F}[\![x{:}F_1 ** F_2]\!]_{E\,U}\} \\
\qquad \texttt{update}\ F_2]\!]_{E\,U} & & \\
\mathcal{F}[\![[F \mid x \leftarrow e]]\!]_{E\,U} & = & \{((t, [m_1, \ldots, m_k]), [v_1, \ldots, v_k]) \mid \exists t.\forall i : 1 \ldots k.(m_i, v_i) \in \mathcal{F}[\![F]\!]_{E[x \mapsto z_i]\,U} \text{ and } m_i.t = t\} \\
& & \quad \text{where} \quad [z_1, \ldots, z_k] = \mathcal{E}[\![e]\!]_E
\end{array}
$$

Figure 15. Feed Language Semantics.

Since every metadata item contains a top-level time t, that time can be used to serialize the set of items as a stream, and our implementation does just that. Items scheduled at the same time may appear in any order in the implementation's serialized stream. To refer to the top-level time in any metadata item m, we write $m.t$.

Figure 15 presents the semantic definitions for $\mathcal{C}$ and $\mathcal{F}$, using conventional set-theoretic notations. The semantics depends upon a semantics for the simply-typed host language, written $\mathcal{E}[\![e]\!]_E$, whose definition we omit. We assume that given environment E with type Γ and expression e with type τ in Γ, $\mathcal{E}[\![e]\!]_E = v$ and $\vdash v : \tau$.

The meaning of core feed C is a pair consisting of the meaning of the schedule of C (written S) and the set of metadata/data pairs for the feed. To calculate this data, the `timeout` function checks whether the item arrival time x_{at} is within the window ($\mathcal{E}[\![e_{win}]\!]_E$) of the scheduled time ($x_t \in S$), returning `None` if not. Otherwise, `timeout` returns its data argument (x_s), which may be `None` because of other networking errors. Using the `timeout` function, we define an alternate universe U' that retrieves data from the outside world, checks for a timeout, and applies the preprocessor ($\mathcal{E}[\![e_{pp}]\!]_E$) before returning. The feed data is then pairs of base metadata (t, ℓ) with data defined by $\mathcal{E}[\![e_f\ (U'(\ell, t))]\!]_E$ where e_f is the PADS-generated parser.

The semantics of both the `all` C and the `any` C feeds first computes the meaning (S, A) of the core feed C. The `all` feed simply returns the data component A. The `any` feed returns a set with one element for each time t in the schedule S. If the data A contains at least one good value at time t, the `any` feed picks an arbitrary member of this set, using the function $\mathsf{select\_one}$. Otherwise, the feed adds error value $((t, \mathsf{nowhere}), \texttt{None})$ for t.

The meaning of the empty feed is the empty set. Computed feeds yield one value per time in the given schedule. The dummy location (nowhere) in the computed feed metadata indicates the value had no physical source. Feed comprehensions also use the dummy location. The filter feed and feed comprehensions together model the full power of the feed comprehensions found in our implementation. The semantics for `let`, union and sum feeds are all straightforward. The union feed is simply the set-theorectic union of its constituent feeds. The sum feed injects the elements of its constituent feeds into a sum and likewise takes their union.

We must take more care when defining the semantics of pairs. The synchronous pair (F_1, F_2) is formed by finding all elements of F_1 at a given time (including erroneous elements) and all elements of F_2 at that time (again including erroneous elements) and generating their Cartesian product. Notice that if the schedules do not intersect, a synchronous pair will empty. The scheduled time of the composite item is the same as the scheduled times of the two underlying feeds.

The elements of the continuous dependent pair $(x{:}F_1 * F_2)$ are calculated by first determining the elements of F_1. For each element v in F_1, we calculate the elements of F_2 by binding x to v in F_2's definition. The elements of the composite feed include all elements x of F_1 paired with the corresponding elements of F_2 scheduled later than x. As with synchronous pairs, we pair the metadata from the constituent feeds in the composite feed. Unlike synchronous pairs, the two elements of the pair do not share a schedule time. We adopt the later time (*i.e.*, the time of the element from F_2) as the schedule time for the composite feed element.

Continuous and local dependent pairs differ in that local dependent pairs suppress further data dependent upon item v_1 from F_1 when the next item v_1' from feed F_1 arrives. In the semantics, the existence of another element (m_1', v_1') in F_1 after (m_1, v_1) implies that the schedule time for the item under consideration in F_2 (v_2) must fall between the schedule times for v_1 and v_1' (otherwise v_2 is not included in the final composite feed).

The semantics for the two variants of `foreach` are defined in terms of the two variants of dependent pairs, simply dropping the first component of each item in the pair feed in both cases. The semantics of the list comprehension feed extends the semantics of synchronous pairs to lists: each list in the resulting feed contains the elements of the generated list with the same scheduled time.

We have proven a soundness theorem for the semantics: the values contained in each feed are pairs of meta-data and data values with the appropriate type. More specifically, if the feed typing rules give feed F type σ `feed`, then its data has type σ and its meta data has type $\mathsf{meta}(\sigma)$ where $\mathsf{meta}(\sigma)$ is defined as follows.

$$
\begin{array}{lcl}
\mathsf{meta}(\tau) & = & \texttt{time} * \texttt{loc} \\
\mathsf{meta}(\tau\ \texttt{option}) & = & \texttt{time} * \texttt{loc} \\
\mathsf{meta}(\sigma_1 * \sigma_2) & = & \texttt{time} * (\mathsf{meta}(\sigma_1) * \mathsf{meta}(\sigma_2)) \\
\mathsf{meta}(\sigma_1 + \sigma_2) & = & \texttt{time} * (\mathsf{meta}(\sigma_1) + \mathsf{meta}(\sigma_2)) \\
\mathsf{meta}(\sigma\ \texttt{list}) & = & \texttt{time} * (\mathsf{meta}(\sigma)\ \texttt{list})
\end{array}
$$

Theorem 1 (Semantic Soundness)
If $\Gamma \vdash F : \sigma\ \texttt{feed}$ *and for all* x *in* $\mathsf{dom}(\Gamma)$, $\vdash E(x) : \Gamma(x)$ *and* $\vdash U : \texttt{time} * \texttt{loc} \rightarrow \texttt{time} * (\texttt{string option})$ *then for all* $(m, v) \in \mathcal{F}[\![F]\!]_{E\,U}$, $\vdash (m, v) : \mathsf{meta}(\sigma) * \sigma$.

The proof follows by induction on the structure of F.

6. PADS/D Implementation and Evaluation

The PADS/D implementation has three parts: the compiler, the runtime system, and the built-in tools library. We describe these parts in turn and evaluate the overall system performance and design.

```
let simple_comon =
{frep = fun ff ->
 ff.all
 {Combinators.format = Comon_format.Source.parse;
  print = Comon_format.Source.print;
  format_rep = Comon_format.Source.tyrep;
  incremental = false;
  header_format = None;
  locations = sites;
  schedule =
    Schedule. every (Time.now (), 10.,
                     Schedule.default_duration, 60.);
  has_records = Comon_format.__PML__has_records;
  pp = None}}
```

Figure 16. Code fragment of compiled simple_comon feed

6.1 The Compiler

The PADS/D compiler consists of `tcc`, the tool configuration compiler for .tc files, and `fmlc`, the compiler for feed declarations (.fml files). Both compilers convert their sources into OCAML code, which is then compiled and linked to the runtime libraries. We implemented both tools with `Camlp4`, the OCAML preprocessor.

The `fmlc` compiler performs code generation in two steps. First, the code generator emits the type declarations for each feed. Second, it generates representations for each feed description. The compiler constructs these representations by extracting elements from the concurrently generated PADS/ML libraries and using polymorphic combinators to build structured descriptions. Figure 16 shows a fragment of the compiled code for the simple CoMon feed in Figure 2.

6.2 The Runtime System

We implement each PADS/D feed as a lazy list of feed items. Following the semantics in Section 5, a feed item is a metadata/data pair, although the meta-data in the implementation is richer, adding data arrival times and more detailed error information.

The PADS/D runtime system is a multi-threaded concurrent system that follows the master-worker implementation strategy. Each worker thread either fetches data from a specified location and parses the data into an internal representation (the *rep*), or synthesizes its data by calling a generator function. Using error conditions, location, scheduled time and arrival time, the worker generates the appropriate metadata, pairs it with the rep and pushes the feed item onto a queue. The master thread pops the feed item from the queue on demand, *i.e.*, when the user program requests the data. The worker thread is *eager*, which guarantees that all data will be fetched and archived, but the master thread is *lazy*, which allows application programs to process only relevant data.

We used the `Ocamlnet 2` library [29] to implement the fetching engine. It batches concurrent fetch requests into groups of 200, a size which balances maximizing throughput with avoiding overwhelming the operating system with too many open sockets.

6.3 Tools Library

As explained in Section 4, we implemented the PADS/D off-the-shelf tool suite using our generic tool framework. Some tools depend upon auxiliary tools. For instance, the feed selector calls a data selector built under the PADS/ML generic tools framework [11] for base feeds. Other tools depend upon external libraries. For instance, the `feed2rrd` tool requires the RRD round-robin database [21] and the `feed2rss` tool uses the XML-Light package [19] for parsing and printing.

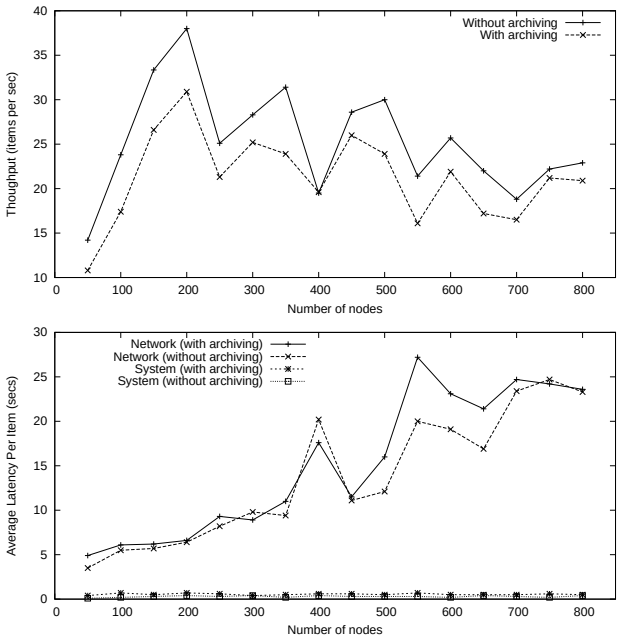

Figure 17. Average throughput and latencies per node

6.4 Experiments

To assess performance, we measure the average network latency to fetch a data item, the average system latency to return a data item after receiving it from the network and the throughput of the system using the CoMon feed description in Figure 4. The throughput measures the average number of items fetched per second.

All the experiments were conducted on a Mac Powerbook G4 computer with a 1.67GHz CPU and 2GB memory running Mac OS X 10.4. In each experiment, we randomly selected 16 subsets of PlanetLab nodes, with increasing size from 50 to 800 in increments of 50. For each set, we applied the profiler tool for the CoMon feed twice, one without archiving and one with it, to measure the throughput and latencies as the system fetched from these node lists. We repeated the experiment ten times and calculated the average values.

Figure 17 shows the average throughput and the average network and system latencies. Generally, the throughput hits peaks at multiples of 200 since the system supports up to 200 concurrent fetches. An anomaly occurred at 400 nodes, as a number of nodes were unreachable because of DNS failures. Note that while network latency increases with the number of nodes, system latency remains almost constant and relatively low, showing that the PADS/D runtime system adds little overhead to the inevitable network fetching cost. Also note that the network latency is almost linear in the number of nodes. The experiments show that the system can fetch from 800 nodes and archive the resulting data in under 40 seconds, well under the 5 minute turnaround time currently supported by CoMon. Taken together, these results suggest that PADS/D is capable of supporting PlanetLab-scale monitoring applications.

6.5 Language or Library

Our feed language is a veneer on OCAML built with the `Camlp4` preprocessor. A natural question is whether the system would be better implemented as a library rather than a language extension. For the reasons described in the following paragraphs, we chose to present our work as a language.

Automatic elimination of boilerplate code. The compiler eliminates boilerplate code by (a) generating both type declarations and values from descriptions (particularly record types and datatypes), something that cannot be done in a library, (b) packaging definitions in modules for name-space management and functor usage, (c) automatically filling in defaults for values omitted from configuration files, and (d) generating complete, stand-alone executables from declarative descriptions and configurations.

Syntax and simplicity of coding style. The underlying interfaces are *very* higher-order, which, without surface sugar, would force a complex coding style on the off-the-shelf user. For instance, almost every line of a description would be translated to an increasingly nested combinator application, and every variable binding would induce a use of higher-order abstract syntax.

Generic programming. OCAML (and most other potential host languages) has no direct support for the generic programming needed to implement the tool suite. After considerable study, the most effective way we have found to provide the required generic programming interface involves judicious use of unsafe casts under the covers. By generating type representations using the compiler, we guarantee these casts cannot go wrong.

***Integration with* PADS.** Core PADS [12, 13, 14, 16] has had success as a language extension on top of C as well as OCAML. Its purpose is to describe and document properties of ad hoc data sources as well as to facilitate generation of local, single-source tools. Extending such descriptions to include source location, availability and access mode helps complete the documentation in a single centralized specification and through a uniform notation. It gives off-the-shelf users everything they need in a single language. Forcing a division of the specification into part library/part language would ruin its cohesiveness, particularly in the context of dependent feeds where there is tight interplay between access mode, location, schedule and format.

Though we believe our current design is well motivated, we also believe the ideas presented here can transcend their current implementation. By defining a compact feed calculus with a precise semantics, we allow the possibility for others to embed our abstractions directly in a language such as Haskell that provides superior support for generic programming.

7. Related Work

Because of space considerations, we survey only the most closely related work.

Systems monitoring. One early and widely-used protocol for system monitoring is SNMP, the simple network management protocol [5], which is supported by commercial tools such as HP's OpenView [2] and free tools such as MRTG [22]. It provides an open protocol format, where vendors supply management information bases (MIBs) that provide a hierarchical description of the hardware's monitoring information. By separating the data description into the MIB, SNMP can be more concise than XML, but it has poor support for ad hoc data, and it is more difficult to update with new data types or even changes to the data format.

For Grid or cluster environments, two popular monitoring tools are Ganglia [17] and Nagios [3]. Ganglia focuses more on continuous monitoring of usage information and consolidates information provided by OS tools like vmstat, iostat, uptime, *etc.* Nagios focuses more on availability information, and logs (or delivers) failure and recovery events. Ganglia uses raw data in XDR for its native fields and XML-encapsulated fields for extensions. Nagios has no standard data format, but instead gathers all data by periodically executing user-specified commands described in a configuration file. The commands use standardized return values to express

status and are typically restricted to no more than 4KB of monitoring data.

What distinguishes PADS/D from systems like SNMP or Ganglia is the ability to automatically parse and monitor virtually any kind of ad hoc data, from node-level information like that collected by Ganglia or SNMP, all the way down to application-level or even protocol-level data. These areas are the ones that are not well served by today's general-purpose monitoring systems. Moreover, the ability to use the same data description to automatically build parsers, in situ tools, and monitoring systems directly from declarative descriptions represents an ease of use not available in other systems.

Functional Programming. The implementation of PADS/D depends upon a tremendous body of past research in both functional stream processing and generic programming. Rather than competing with these technologies, PADS/D builds upon them and makes them highly accessible to off-the-shelf users in an important new domain – that of distributed ad hoc data processing.

Reactive Functional Programming. FRP [9] shares a tenuous relationship to work on PADS/D in that both technologies arise from the need to perform processing on a schedule over time-indexed values. However, FRP focuses on the definition and implementation of *continuous* time-varying values. So far, we have not found use for such values in our domain, only for discrete time-indexed sequences of events.

Web Mashups. Web Mashup languages such as MashMaker [10] and Yahoo Pipes [33] allow naive web programmers to extract data from web sites and RSS feeds and recombine them, often using conventional functional programming paradigms such as map and filter. The focus is on end-user programming with relatively small amounts of data that can be displayed to a user in a web browser. Errors are generally ignored as completeness or absolute correctness of information is not critical in the domains of interest. Unlike PADS/D, which allows users to write rich descriptions expressing the location, format, schedule and access mode of the data, Yahoo Pipes, for instance, acquires data through a fixed collection of black boxes. For this reason, PADS/D and mashup languages have the potential to be complementary technologies, with PADS/D descriptions serving to define new ad hoc data sources for mashups. In fact, this idea motivated the design and implementation of the PADS/D ad hoc-to-RSS conversion tool.

8. Conclusions

The explosive growth of the Internet has made monitoring and managing data systems distributed across wide-area networks increasingly important. The possibility of partial failure and the need to synchronize makes such code tedious and difficult to write correctly, particularly for data experts whose skills are in domains other than networking. In this paper, we describe the PADS/D system, which allows users to declaratively specify their data systems and then generate a wide-variety of tools for manipulating the data: from stand-alone tools, to simple libraries for writing their own analyses, to generic libraries for building new generic tools. We precisely specify the meaning of our language via a sound denotational semantics and show via experimentation that the system has acceptable performance overheads.

Acknowledgments

This material is based upon work supported by the NSF under grants 0612147 and 0615062. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Gene ontology project. http://www.geneontology.org/.

[2] HP OpenView products. http://www.managementsoftware.hp.com/products/.

[3] Nagios. http://www.nagios.org/.

[4] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, 2003.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). RFC 1157, May 1990.

[6] M. CL, B. D, H. MA, H. C, and T. OG. Finding function: evaluation methods for functional genomic data. *BMC Genomics*, 7:187, 2006.

[7] M. CL, R. D, W. A, H. M, C. C, T. CL, D. K, and T. OG. Discovery of biological networks from diverse functional genomic data. *Genome Biology*, 6(13), 2005.

[8] J. Clark and S. DeRose. XML path language (XPath), version 1.0. http://www.w3.org/TR/xpath.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[10] R. Ennals and D. Gay. User-friendly functional programming for web mashups. In *ICFP*, pages 223–233, 2007.

[11] M. Fernandez, K. Fisher, J. Foster, M. Greenberg, and Y. Mandelbaum. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *PADL*, pages 133–149, 2008.

[12] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304, 2005.

[13] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL*, 2006.

[14] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *POPL*, pages 421–434, Jan. 2008.

[15] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *NSDI*, 2004.

[16] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A functional data description language. In *POPL*, 2007.

[17] M. L. Massie, B. N. Chun, , and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.

[18] T. A. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[19] Motion-Twin. XML-Light. http://tech.motion-twin.com/xmllight.html.

[20] NCBI. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.

[21] T. Oetiker. Round robin database tool. http://oss.oetiker.ch/rrdtool/index.en.html.

[22] T. Oetiker and D. Rand. Multi Router Traffic grapher. http://people.ee.ethz.ch/oetiker/webtools/mrtg.

[23] V. Pai and K. Park. CoBlitz: A Scalable Large-File Transfer Service over HTTP. http://codeen.cs.princeton.edu/coblitz/.

[24] V. Pai and K. Park. CoMon: Monitoring infrastructure for PlanetLab. http://comon.cs.princeton.edu/.

[25] C. Partridge, T. Mendez, and W. Milliken. Host anycast service. http://www.ietf.org/rfc/rfc1546.txt.

[26] PlanetLab. An open testbed for developing, deploying and accessing planetary-scale services, September 2002.

[27] S. RSG, H. M, H. C, M. CL, and T. OG. GOLEM: An interactive graph-based gene ontology navigation and analysis tool. *BMC Bioinformatics*, 7:443, 2006.

[28] C. Stark, B.-J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers. BioGRID: A general repository for interaction datasets. *Nucl. Acids Res.*, 34:D535–539, 2006.

[29] G. Stolpmann and P. Doane. Ocamlnet 2. http://projects.camlcity.org/projects/ocamlnet.html.

[30] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10:189–199, 1998.

[31] S. Weirich. Encoding intensional type analysis. In *ESOP*, pages 92–106, 2001.

[32] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, pages 224–235, 2003.

[33] Yahoo pipes. http://pipes.yahoo.com.

[34] Z. Yang. Encoding types in ML-like languages. In *ICFP*, 1998.