Language Support for Processing Distributed Ad Hoc Data

Kenny Q. Zhu[1] Daniel S. Dantas[2] Kathleen Fisher[3] Limin Jia[2]
Yitzhak Mandelbaum[3] Vivek Pai[2] David Walker[2]

[1] Shanghai Jiao Tong University [2] Princeton University [3] AT&T Labs Research

Abstract

This paper presents the design, theory and implementation of Gloves[1], a domain-specific language that allows users to specify the provenance (the derivation history starting from the origins), syntax and semantic properties of collections of distributed data sources. In particular, Gloves specifications indicate *where* to locate desired data, *how* to obtain it, *when* to get it or to give up trying, and *what* format it will be in on arrival. The Gloves system compiles such specification into a suite of data-processing tools including an archiver, a provenance tracking system, a database loading tool, an alert system, an RSS feed generator and a debugging tool. In addition, the system generates description-specific libraries so that developers can create their own applications. Gloves also provides a generic infrastructure so that advanced users can build new tools applicable to any data source with a Gloves description. We show how Gloves may be used to specify data sources from two domains: CoMon, a monitoring system for PlanetLab's 800+ nodes, and Arrakis, a monitoring system for an AT&T web hosting service. We show experimentally that our system can scale to distributed systems the size of CoMon. Finally, we provide a denotational semantics for Gloves and use this semantics to prove two important theorems. The first shows that our denotational semantics respects the typing rules for the language, while the second demonstrates that our system correctly maintains the provenance.

1. Introduction

One of the primary tasks in developing a distributed system is keeping it running smoothly over long periods of time. Consequently, well-designed distributed systems include a subsystem responsible for monitoring the health, security and performance of its constituent parts. CoMon [24], designed to monitor PlanetLab [25], is an illustrative example. CoMon operates by attempting to gather a log file from each of 800+ PlanetLab nodes every five minutes. When all is well (which it never is) each node responds with an ASCII data file in mail-header format containing the node's kernel version, its uptime, its memory usage, the ID of the user with the greatest CPU utilization, *etc.* CoMon archives this data in compressed form and processes the information for display to PlanetLab users. CoMon also tracks various networking problems, maintains lists of "problem nodes" and supports on-going time-indexed queries on the data. These features make CoMon an invaluable resource for users who need to monitor the health and performance of their PlanetLab applications or experiments.

Almost all distributed systems should have similar monitoring infrastructure. However, the implementors of each new distributed system currently 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. In addition, implementors cannot ignore errors: they must properly handle network errors, partial disconnects and corrupted data. They also 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. Last but not least, new monitoring systems must interact with legacy devices, legacy software and legacy data, often preventing implementers from using robust off-the-shelf data management tools built for standard formats like XML and RSS.

Name/*Use*	**Properties**
CoMon [24] *PlanetLab host monitoring*	Multiple data sets Archiving every 5 minutes From evolving set of 800+ nodes
CoralCDN [13] *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 [27] *Curated gene and protein data*	XML and Tab-separated Formats multiple data sets $<=$ 50MB each Monthly data releases

Figure 1. Example distributed ad hoc data sources.

Systems researchers are not alone in their struggles with distributed collections of ad hoc data sources. Similar problems appear in the natural and social sciences, including biology, physics and economics. For example, systems such as BioPixie [21], Grifn [20] and Golem [26], 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 used in these and other applications.

We have developed a new domain-specific language and system called Gloves to facilitate the creation, maintenance and evolution of tools for processing ad hoc data from distributed sources. The language allows developers to describe the provenance, syntax and semantics of data sources they wish to monitor, including:

- **Where** the data is located. The data may be in a file 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 repeating 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.

[1] We call the system Gloves because it helps users get their hands on things that are difficult to handle.

- **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.
- **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 XML. It may be in the kind of non-standard format that PADS [12, 16] was designed to describe or for which the user has a well-typed parser.

The GLOVES system 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 including an archiver, provenance tracking system, database loader and others.

The GLOVES system can generate all of these tools from declarative descriptions and tool configuration specifications. Thus for common tasks, users can manage distributed data sources simply by writing high-level declarative specifications. 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.

To provide extensibility, GLOVES 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. To meet this need, the system provides support for creating new tools by 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. The cost of this flexibility is a steeper learning curve because the programmer must learn a variety of interfaces. Functional programmers may find these interfaces intuitive, but computational scientists 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 GLOVES system to write a single generic program to complete the task. For example, the Round Robin Database (RRD) loader is generic because it is possible to load data from any specified source into the RRD tool [22] 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 GLOVES and to compensate for the lack of built-in generic programming support in OCAML. Still, the reward for building generic tools is high: as more and more such tools are built, the life of the off-the-shelf user becomes easier and easier. We used this infrastructure to build the off-the-shelf tools described earlier.

To guide the design and implementation of GLOVES, we have developed an idealized, first-order calculus and associated type system to model its core elements. We have equipped this calculus with a denotational semantics that specifies for each data source description the set of (meta-data, data) pairs that it should produce. The semantics allows users to calculate and reason about the data that they should be receiving. We have proven the type system sound with respect to the semantics. Moreover, we have used the semantics to prove *dependency correctness*, a key theorem inspired by earlier work on provenance in databases by Cheney *et al.* [8]. This theorem guarantees the correct provenance meta-data is associated with every data item.

In addition to being of theoretical interest, the calculus and its meta-theory have served as a guide for our implementation infrastructure. In particular, the compilation strategy for our surface-level language was influenced by observations about how higher-level constructs could be compiled into combinators from our calculus. We also reorganized the way earlier versions of our system processed and propagated provenance meta-data in order to obey the principle of dependency correctness.

Contributions. The paper makes the following contributions:

- It describes the design of a domain-specific language for specifying provenance, syntax and semantic properties of distributed ad hoc data sources.
- It provides a formal denotational semantics for our language and proves the key properties of Type Soundness and Dependency Correctness.
- It describes the architecture of the system and how it enables multiple modes of use.
- It demonstrates the practicality of our architecture and its implementation by showing the infrastructure will scale to handle systems the size of PlanetLab.

Outline. In the rest of the paper, we describe the examples we will use throughout the paper (Section 2), show how to describe these data sources in GLOVES (Section 3), describe the generated tool infrastructure and its modes of use (Section 4), define a denotational semantics and prove our key correctness properties (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 [25] by attempting to fetch data from each of PlanetLab's 800+ nodes every 5 minutes. This data 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 problems and generates reports of deviant machines or user programs. Finally, CoMon archives the data so PlanetLab users can perform 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 which may include daily or seasonal cycles.

```
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 feed (simple_comon.fml).

```
feed comon_1 =
  base {|
    sources  = any sites;
    schedule = every 5 min, lasting 2 hours;
    format   = Comon_format.Source;
  |}
```

Figure 3. Description fragment for data from one of many sites (sites.fml).

3. GLOVES: An Informal Introduction

The GLOVES 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 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.

The last field in a base feed constructor is the **format** field, which specifies the syntax of the fetched data by supplying a parser for it. In this case, Comon_format.Source is actually a parser generated from a PADS/ML [16] specification file (comon_format.pml), which we have omitted because of space constraints. While it is not strictly necessary for GLOVES programmers to use PADS/ML specifications in their descriptions, and the key ideas in this paper can be understood without a deep knowledge of PADS/ML, the two languages have been designed to fit together elegantly. Moreover, several of our generated tools exploit the common underlying infrastructure to enable useful data analyses and transformations over feeds whose formats are specified by PADS/ML descriptions.

A simple variation of our first description appears in Figure 3. In contrast to simple_comon, which returns data from *all* sites per time slice, comon_1 returns data from just *one* site per time slice. This difference between the two is specified using the **any** constructor instead of the **all**. 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) [5]. 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. The schedule for comon_1 indicates the system should fetch data every five minutes 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.

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

(* Feed of nodes to query *)
feed nodes =
  base {|
    sources  = all config_locations;
    schedule = every 5 min;
    format   = Nodelist.Source;
  |}

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

(* 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 (value nodelist))));
      schedule = once, timeout 60.0 sec;
      format   = Comon_format.Source;
    |}
```

Figure 4. Node location feed drives data collection (comon.fml).

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

So far, our examples have hard-coded the set of locations from which to gather data. In reality, however, 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 called nodes that describes the configuration information: it is available from the config_location, it should be fetched every five minutes, and its format is described by the PADS/ML description `source` given in the file `nodelist.pml`, which appears in Figure 5.

The PADS/ML description in Figure 5 specifies that `source` is a list (`plist`) of new-line terminated records (`precord`) 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 five minutes. In fact, because of the possibility of errors, the feed actually delivers a *list option* every five minutes: `Some` if the list is populated with data, `None` if the data was unavailable at the given time-slice. Furthermore, to record provenance information, each element in the feed is actually a pair of meta-data and the payload value.

Given the `nodes` specification, we can define the `comon` feed using the notation **`foreach`** `nodelist` **`in`** `nodes` **`create`** `. . .` . In this declaration, each element of `nodes` is bound in turn to the variable `nodelist` for use in generating the new feed declared in "`...`" The final result of the **`foreach`** is the union of all such newly generated feeds. Both the payload data *and* the provenance meta-data of `nodelist` may be used in creating the dependent feed. In this example, we use the function `value` to select only the payload portion, ignoring the meta-data. The complementary function `meta` provides access to the provenance information.

To complete the construction of the `comon` feed, a small amount of functional programming allows the user to manage errors and strip out comment fields. Any such simple transformations may be written directly in OCAML, the host language into which we have embedded GLOVES. In particular, the `current` function checks if the `nodelist` value is `Some l`, in which case it caches `l` before returning it as a result. Otherwise, if the `nodelist` value is `None` (indicating an error), the most recently cached list of nodes is used instead. The rest of the **`sources`** specification filters out comment fields and converts the host names to URLs with the required port using the auxiliary function `makeURL`.

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

The previous examples all showcased feeds containing a single type of data. GLOVES also provides a datatype mechanism so we may construct compound feeds containing data of different sorts. As an example of where such a construct is useful, the CoMon system includes a number of administrative data sources. One such source is a collection of node profiles, collecting the domain name, IP address, physical location, *etc.*, for each node in the cluster. A second such source 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 its definition.

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 extract the payload data from the provenance and error infrastructure, an operation that is guaranteed to succeed because of the `is_good` guard. The function `get_hosts` then selects the non-comment 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. When given a host `h`, `gen_stats` creates an every-five-minute schedule lasting twenty-four hours with a one minute timeout. It uses this schedule to describe a compound feed that 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 produces a compound feed that for each host returns a pair of its ping and uptime statistics, conveniently grouping 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 so the full feed description has many more components 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 element `hl`, the right-hand side of the comprehension uses the `value` function to extract the payload from the meta-data and then considers each host `h` from the resulting list 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 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...create`** construct generates a feed of snapshots from the feed of host lists.

4. Working with Feeds

4.1 The "Off the Shelf" User

The GLOVES system provides a suite of "off-the-shelf" tools to help users cope with standard data administration needs. After writing a GLOVES 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 of tool specifications. The header specifies the path to the feed de-

```
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 hl)

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

feed gen_stats (string h) =
  let s = every 5 mins,
          timeout 1 min,
          lasting 24 hours 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 create
     [ gen_stats (h) | h <- value hl ]
```

Figure 6. Simplified version of Arrakis feed (`arrakis.fml`).

```
feed comon.fml/comon

tool provtrack
{
  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. Example tool configuration file (`comon.tc`).

```
======================================================
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
... omitted ...
```

Figure 8. Fragment of provenance tracker output (`comon.acc`).

scription 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.*, `provtrack` 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. GLOVES 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 some of 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.

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.

Provenance tracker. The provenance tracker maintains statistical profiles for feeds. These include error rates, most common errors and their source locations and times. For numeric data, the tracker 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. The user can configure the tracker 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 tracker can output either plain text or XML. Figure 8 shows portions of provenance tracker 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 an RRD. Users specify a function to transform feed items into numeric values and RRD 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.

RSS feed generator. The RSS feed generator converts a GLOVES 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, GLOVES 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 provenance information. Therefore, we model the feed API on the list APIs of functional languages but provide two levels

```
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 9. 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 10. Revised code fragment with provenance meta-data.

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 meta-data 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 9 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. 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, if we want a report of the names of the nodes that have the lowest average loads, the above solution is not good enough because the CoMon data format does not include the node location in the data. In such situations, provenance meta-data is essential. We therefore replace the last two lines of Figure 9 with the code in Figure 10 that exploits the meta-data. First, we give the `update_m` (update with meta) function that uses meta-data to associate a location with every load in the table. It relies on the `Meta` module, which Gloves provides to facilitate management of meta-data. Next, we show a call to the meta-aware fold `fold_m`, which passes the payload and its meta-data to the folding function. Last, the call `print_top_with_loc 10` prints the ten lowest average loads and 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 parametrically 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 interpretation are the feed creator and the provenance tracker. 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 discussed in the literature, and various solutions have been proposed [11, 31, 32]. 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. Gloves Semantics

Developing a formal semantics for Gloves 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 Gloves descriptions, including subtleties related to synchronization, timeouts, errors and provenance.

To express locations, times, schedules and constraints, the feed calculus depends upon a *host language*, which we take to be the simply-typed lambda calculus. Figure 11 presents its syntax, which includes a collection of constants to simplify the semantics: strings (w), times (t) and locations (ℓ). We assume times may be added and compared and we let ∞ represent a time later than all others. We assume that the set of locations includes the constant `nowhere`, indicating the associated data was computed rather than fetched. 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 and we use pattern matching where convenient (*e.g.,* $\lambda(x, y).e$ is a function over pairs).

5.1 Feed Syntax and Typing

The abstract syntax for our feed calculus and its typing rules appear in Figures 12 and 13, respectively. The feed typing judgment has

(host-language base types)
$b ::= \mathsf{bool} \mid \mathsf{string} \mid \mathsf{time} \mid \mathsf{loc}$

(host-language types)
$\tau ::= b \mid \tau\ \mathsf{option} \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \mid \tau\ \mathsf{list} \mid \tau\ \mathsf{set} \mid \tau_1 \to \tau_2$

(host-language values)
$v ::=$

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

(host-language expressions)
$e ::=$

x	variables
$\mid\ v$	data values
$\mid\ \mathtt{None} \mid \mathtt{Some}\ e$	option expressions
$\mid$ ...	more typed lambda expressions

Figure 11. Host Language Syntax.

(feed payload types)
$\sigma ::= \tau \mid \tau\ \mathsf{option} \mid \sigma_1 * \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma\ \mathsf{list}$

(core feeds)
$C ::=$

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

(feeds)
$F ::=$

$\mathtt{all}\ C$	all sources
$\mid\ \mathtt{any}\ C$	one of multiple sources
$\mid\ \emptyset$	empty feed
$\mid\ \mathtt{One}(e_v, e_t)$	singleton feed
$\mid\ \mathtt{SchedF}(e)$	schedule to feed
$\mid\ F_1 \cup F_2$	union feed
$\mid\ F_1 + F_2$	sum feed
$\mid\ (F_1, F_2)$	pair feed
$\mid\ [F \mid x \leftarrow e]$	list comprehension feed
$\mid\ \{\lvert F_2 \mid x \leftarrow F_1 \rvert\}$	feed comprehension
$\mid\ \mathtt{filter}\ F\ \mathtt{with}\ e$	filter feed
$\mid\ \mathtt{let}\ x = e\ \mathtt{in}\ F$	let feed

Figure 12. Feed Language Syntax.

the form $\Gamma \vdash F : \sigma$ feed, which means that in the context Γ mapping variables to host language types τ, F is a feed of σ values. The core typing judgment, which has the form $\Gamma \vdash C : \sigma$ core, conveys the same information for core feeds.

Intuitively, a feed carrying values of type σ is a sequence of payload values of type σ. However, to record provenance information, we pair each payload value with meta-data, so a feed is actually a sequence of (meta-data, payload) pairs. At the top-level, meta-data consists of a triple of the scheduled time for the payload, a *dependency set* that records the origin and scheduled time of any data that contributed to the payload, and a nested meta-data field whose form depends upon the type of the payload.

Formally, we let m range over top-level meta-data, ds range over dependency sets, and n range over "nested" meta-data:

m	$::=$	(t, ds, n)	top-level meta-data
ds	$::=$	$\{(t_1, \ell_1), \ldots, (t_n, \ell_n)\}$	dependency set
n	$::=$	$(t, \ell, \mathtt{None})$	base meta-data (timeout)
	$\mid$	$(t, \ell, \mathtt{Some}\ t)$	base meta-data (success)
	$\mid$	(n_1, n_2)	pair meta-data
	$\mid$	$\mathtt{inl}\ n$	sum meta-data
	$\mid$	$\mathtt{inr}\ n$	sum meta-data
	$\mid$	$[n_1, \ldots, n_k]$	list meta-data

Given meta-data m, we write $m.t$, $m.ds$ and $m.nest$ for the first, second and third projections (respectively) of m. Base meta-data is a triple of the scheduled time, the location of origin and an optional arrival time where None indicates the data did not arrive in a timely fashion.

As shown in Figure 12, we define the feed payload type σ in terms of host language types, stratified to facilitate the proof of semantic soundness. We use the function $\mathsf{meta}(\sigma)$ to define the type of meta-data associated with payload of type σ:

$$\begin{aligned}
\mathsf{meta}(\sigma) &= \mathsf{time} * \mathsf{ds} * \mathsf{nest}(\sigma) \\
\mathsf{ds} &= (\mathsf{time} * \mathsf{loc})\ \mathsf{set} \\
\mathsf{nest}(\tau) &= \mathsf{time} * \mathsf{loc} * (\mathsf{time\ option}) \\
\mathsf{nest}(\tau\ \mathsf{option}) &= \mathsf{time} * \mathsf{loc} * (\mathsf{time\ option}) \\
\mathsf{nest}(\sigma_1 * \sigma_2) &= \mathsf{nest}(\sigma_1) * \mathsf{nest}(\sigma_2) \\
\mathsf{nest}(\sigma_1 + \sigma_2) &= \mathsf{nest}(\sigma_1) + \mathsf{nest}(\sigma_2) \\
\mathsf{nest}(\sigma\ \mathsf{list}) &= \mathsf{nest}(\sigma)\ \mathsf{list}
\end{aligned}$$

Feed typing depends upon a standard judgment for typing lambda calculus expressions: $\Gamma \vdash e : \tau$.

Having covered these preliminaries, we can now discuss the syntax and typing for each of the feed constructs in Figure 12. Core feeds express the structure of base feeds, describing the data sources (`src`), schedule (`sched`), window (`win`), preprocessing function (`pp`) and file format (`format`). The source field describes the set of locations from which to fetch data. It may contain pseudo-locations that model the `proc` form found in the implementation. Instead of having timeouts specified as part of schedules, as we did in the surface language, the calculus separates these two concepts into distinct fields, which simplifies the semantics. If an item specified to arrive at time t by schedule e_2 fails to arrive within the window e_3, the feed pretends it received the value None. Otherwise, it wraps the received data string in an option. As a result, the preprocessor e_4 maps a $\mathsf{string\ option}$ to a $\mathsf{string\ option}$, where a result of None indicates either a network or preprocessing error. Finally, the formatting function e_5 parses the output of the preprocessor to produce a value of type $\tau\ \mathsf{option}$, where a None result indicates a network, preprocessing or formatting error. (For the sake of simplicity, we do not model the variety of error codes that the implementation supports.)

The feed `all` C selects all the data from the core feed C. The feed `any` C selects for each time in the schedule for C the first good value to arrive from any location. It returns None paired with appropriate meta-data if no such good value exists.

The empty feed ($\emptyset$) contains no elements and may be ascribed any feed type. The singleton feed $\mathtt{One}(e_v, e_t)$ constructs a feed containing a single value e_v at a single time e_t. The schedule feed $\mathtt{SchedF}(e)$ builds a feed whose elements are the times in the schedule e. The union feed merges two feeds with the same type

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

$$\frac{\Gamma \vdash C : \sigma\ \mathsf{core}}{\Gamma \vdash \mathtt{all}\ C : \sigma\ \mathsf{feed}} \ (\textit{t-all})$$

$$\frac{\Gamma \vdash C : \sigma\ \mathsf{core}}{\Gamma \vdash \mathtt{any}\ C : \sigma\ \mathsf{feed}} \ (\textit{t-any})$$

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

$$\frac{\Gamma \vdash e_v : \tau \qquad \Gamma \vdash e_t : \mathsf{time}}{\Gamma \vdash \mathtt{One}(e_v, e_t) : \tau\ \mathsf{feed}} \ (\textit{t-one})$$

$$\frac{\Gamma \vdash e : \mathsf{time\ set}}{\Gamma \vdash \mathtt{SchedF}(e) : \mathsf{time\ feed}} \ (\textit{t-schedule})$$

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

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

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

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

$$\frac{\Gamma \vdash F_1 : \sigma_1\ \mathsf{feed} \quad \Gamma, x{:}\mathsf{meta}(\sigma_1) * \sigma_1 \vdash F_2 : \sigma_2\ \mathsf{feed}}{\Gamma \vdash \{\!|F_2 \mid x \leftarrow F_1|\!\} : \sigma_2\ \mathsf{feed}} \ (\textit{t-comp})$$

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

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash F : \sigma\ \mathsf{feed}}{\Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ F : \sigma\ \mathsf{feed}} \ (\textit{t-let})$$

Figure 13. Feed Language Typing.

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 pair feed, written (F_1, F_2), combines the elements of the two nested feeds synchronously, matching elements that have the same *scheduled* time, regardless of when those elements actually *arrive*. The list feed $[F \mid x \leftarrow e]$, in contrast, provides n-way synchronization, where n is the length of the input list e. Each element e_i in e defines a feed $F_i = F[x \mapsto e_i]$. For each time t with a value v_i in each feed F_i, the list feed returns the list $[v_1, \ldots, v_n]$ (and appropriate meta-data). Note that if the F_i feeds share a schedule s, then each feed will have a value for every time in the schedule s, even in the presence of errors, so the synchronization will succeed at each time in the schedule s.

The feed comprehension $\{\!|F_2 \mid x \leftarrow F_1|\!\}$ creates a feed with elements $F_2[x \mapsto v]$ when v is an element of F_1. Note that the entry v is a pair of meta-data (with type $\mathsf{meta}(\sigma)$) and payload data (with type σ). The feed $\mathtt{filter}\ F\ \mathtt{with}\ e$ eliminates elements v from F when $e\ v$ is $\mathtt{false}$. Finally, let feeds $\mathtt{let}\ x = e\ \mathtt{in}\ F$ provide a convenient mechanism for binding intermediate values.

Several of the surface language constructs presented in Section 3 may be modeled as derived constructs in the calculus. For instance, (**foreach** x in F_1 **create** F_2) can be modeled as a $\{\!|F_2 \mid x \leftarrow F_1|\!\}$. Likewise, the surface language comprehension $\{\!|e_2 \mid x \leftarrow F_1,\ e_1|\!\}$ can be modeled as $\{\!|\mathtt{One}(e_2, x.1.t) \mid x \leftarrow \mathtt{filter}\ F_1\ \mathtt{with}\ e_1|\!\}$. When e_s is a schedule and e_t is a function over times, purely artificial "computed" feeds may be modeled as $\{\!|\mathtt{One}(e_t\ x.1.t, x.1.t) \mid x \leftarrow \mathtt{SchedF}(e_s)|\!\}$.

5.2 Feed Semantics

We give a denotational semantics for our formal feed language in Figure 14. 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 $U(t_s, \ell) = (t_a, \mathtt{Some}\ w)$, the interpretation is that if the run-time system requests data from location ℓ at time t_s then string data w will be returned at time t_a. The time t_a must be no earlier than t_s. When $U(t_s, \ell) = (\infty, \mathtt{None})$, networking errors have made location ℓ unreachable.

The semantic definitions for $\mathcal{C}$ and $\mathcal{F}$ use conventional set-theoretic notations. They depend 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 the set of (meta-data, payload) pairs for the feed. To construct this set, we first compute the list of source locations L, the set of times in the schedule S and the length of the window W. The `timeout` function checks whether the item arrival time x_{at} is within the window W of the scheduled time x_t, returning `None` if not. Otherwise, `timeout` returns its data argument x_s, which may be `None` because of other networking errors. Similarly, the `arrival` function returns the arrival time `Some` x_{at} if the item arrived within the window and `None` otherwise. The function `meta` uses the `arrival` function to construct the meta-data for the item, consisting of the scheduled time t, the dependency set containing the scheduled time and source location $\{(t, \ell)\}$, and the nested meta-data, which includes the scheduled time t, the source location ℓ, and the actual arrival time $\mathtt{arrival}(t, U(t, \ell))$. (This apparent redundancy goes away with non-core feeds.) Using the `timeout` function, we define an alternate universe U' that retrieves data from the outside world using the original universe U, checks for a timeout, and applies the preprocessor ($\mathcal{E}[\![e_{pp}]\!]_E$). To compute the payload, the `val` function applies the formatting function $\mathcal{E}[\![e_f]\!]_E$ to the value returned by the alternative universe U' at time t for location ℓ. Finally, the result is the set of all pairs of meta-data and payload produced for each location ℓ in the list L and time t in the schedule S.

The semantics of the $\mathtt{all}\ C$ feed is simply the semantics of the underlying core feed. The semantics of the $\mathtt{any}\ C$ feed selects for each time t in the schedule S of the core feed C the earliest good payload value from any location if one exists, or `None` otherwise. It returns the set of all such values v_t, paired with the appropriate meta-data. To compute this set, the function first computes the meaning A of the core feed C. It extracts the schedule S from the meta-data in A. For each time t in the schedule, it computes

$$\begin{array}{lll}
\mathcal{C}\llbracket\{\texttt{src} = e_{src}; & = & \{(\texttt{meta}(t,\ell),\texttt{payload}(t,\ell)) \mid \ell \in L \text{ and } t \in S\}\\
\quad \texttt{sched} = e_{sched}; & & \quad\text{where}\\
\quad \texttt{win} = e_{win}; & & \qquad L = \mathcal{E}\llbracket e_{src}\rrbracket_E\\
\quad \texttt{pp} = e_{pp}; & & \qquad S = \mathcal{E}\llbracket e_{sched}\rrbracket_E\\
\quad \texttt{format} = e_f;\}\rrbracket_{E\,U} & & \qquad W = \mathcal{E}\llbracket e_{win}\rrbracket_E\\
& & \qquad \texttt{timeout} = \lambda(x_t,(x_{at},x_s)).\texttt{if}\ x_{at} \le x_t + W\ \texttt{then}\ x_s\ \texttt{else}\ \texttt{None}\\
& & \qquad \texttt{arrival} = \lambda(x_t,(x_{at},x_s)).\texttt{if}\ x_{at} \le x_t + W\ \texttt{then}\ \texttt{Some}\ x_{at}\ \texttt{else}\ \texttt{None}\\
& & \qquad \texttt{meta} = \lambda(t,\ell).(t,\{(t,\ell)\},(t,\ell,\texttt{arrival}(t,U(t,\ell))))\\
& & \qquad U' = \lambda(t,\ell).\mathcal{E}\llbracket e_{pp}\rrbracket_E\,(\texttt{timeout}\,(t,U(t,\ell)))\\
& & \qquad \texttt{payload} = \lambda(t,\ell).\mathcal{E}\llbracket e_f\rrbracket_E\,(U'(t,\ell))
\end{array}$$

$$\mathcal{F}\llbracket\texttt{all}\ C\rrbracket_{E\,U} = \mathcal{C}\llbracket C\rrbracket_{E\,U}$$

$$\begin{array}{lll}
\mathcal{F}\llbracket\texttt{any}\ C\rrbracket_{E\,U} & = & \{((t,DS_t,nest_t),v_t) \mid t \in S\}\\
& & \quad\text{where}\ \begin{array}{lll}
A & = & \mathcal{C}\llbracket C\rrbracket_{E\,U}\\
S & = & \{m.t \mid (m,v) \in A\}\\
A_t & = & \{(m,v) \mid (m,v) \in A \text{ and } m.t = t\}\\
DS_t & = & \bigcup_{(m,v)\in A_t} m.ds\\
G_t & = & \{(m,\texttt{Some}\ v) \mid (m,\texttt{Some}\ v) \in A_t\}\\
(nest_t,v_t) & = & \begin{cases} (m.nest,v) \text{ where } (m,v) = \mathsf{earliest}(G_t) & \text{if} \quad |G_t| > 0\\ ((t,\texttt{nowhere},\texttt{None}),\texttt{None}) & \text{if} \quad |G_t| = 0 \end{cases}
\end{array}
\end{array}$$

$$\mathcal{F}\llbracket\emptyset\rrbracket_{E\,U} = \{\ \}$$

$$\mathcal{F}\llbracket\texttt{One}(e_v,e_t)\rrbracket_{E\,U} = \{((\mathcal{E}\llbracket e_t\rrbracket_E,\{\ \},(\mathcal{E}\llbracket e_t\rrbracket_E,\texttt{nowhere},\texttt{Some}\ \mathcal{E}\llbracket e_t\rrbracket_E)),\mathcal{E}\llbracket e_v\rrbracket_E)\}$$

$$\mathcal{F}\llbracket\texttt{SchedF}(e)\rrbracket_{E\,U} = \{((t,\{\ \},(t,\texttt{nowhere},\texttt{Some}\ t)),t) \mid t \in \mathcal{E}\llbracket e\rrbracket_E\}$$

$$\mathcal{F}\llbracket F_1 \cup F_2\rrbracket_{E\,U} = \mathcal{F}\llbracket F_1\rrbracket_{E\,U} \bigcup \mathcal{F}\llbracket F_2\rrbracket_{E\,U}$$

$$\begin{array}{lll}
\mathcal{F}\llbracket F_1 + F_2\rrbracket_{E\,U} & = & \{((m.t,m.ds,\texttt{inl}\ m.nest),\texttt{inl}\ v) \mid (m,v) \in \mathcal{F}\llbracket F_1\rrbracket_{E\,U}\} \bigcup\\
& & \{((m.t,m.ds,\texttt{inr}\ m.nest),\texttt{inr}\ v) \mid (m,v) \in \mathcal{F}\llbracket F_2\rrbracket_{E\,U}\}
\end{array}$$

$$\begin{array}{lll}
\mathcal{F}\llbracket(F_1,F_2)\rrbracket_{E\,U} & = & \{((m_1.t,m_1.ds \cup m_2.ds,(m_1.nest,m_2.nest)),(v_1,v_2)) \mid\\
& & \quad (m_1,v_1) \in \mathcal{F}\llbracket F_1\rrbracket_{E\,U} \text{ and } (m_2,v_2) \in \mathcal{F}\llbracket F_2\rrbracket_{E\,U} \text{ and } m_1.t = m_2.t\}
\end{array}$$

$$\begin{array}{lll}
\mathcal{F}\llbracket[F \mid x \leftarrow e]\rrbracket_{E\,U} & = & \{((t,\bigcup_{i=1\ldots k} m_i.ds,[m_1.nest,\ldots,m_k.nest]),[v_1,\ldots,v_k]) \mid\\
& & \quad \forall i : 1\ldots k.(m_i,v_i) \in \mathcal{F}\llbracket F\rrbracket_{(E,x\mapsto z_i)\,U} \text{ and } m_i.t = t\}\\
& & \quad \text{where} \quad [z_1,\ldots,z_k] = \mathcal{E}\llbracket e\rrbracket_E
\end{array}$$

$$\mathcal{F}\llbracket\{|F_2 \mid x \leftarrow F_1|\}\rrbracket_{E\,U} = \{((m_2.t,m_1.ds \cup m_2.ds,m_2.nest),v_2) \mid (m_1,v_1) \in \mathcal{F}\llbracket F_1\rrbracket_{E\,U} \text{ and } (m_2,v_2) \in \mathcal{F}\llbracket F_2\rrbracket_{(E,x\mapsto(m_1,v_1))\,U}\}$$

$$\mathcal{F}\llbracket\texttt{filter}\ F\ \texttt{with}\ e\rrbracket_{E\,U} = \{(m,v) \mid (m,v) \in \mathcal{F}\llbracket F\rrbracket_{E\,U} \text{ and } \mathcal{E}\llbracket e\ (m,v)\rrbracket_E = \texttt{true}\}$$

$$\mathcal{F}\llbracket\texttt{let}\ x = e\ \texttt{in}\ F\rrbracket_{E\,U} = \mathcal{F}\llbracket F\rrbracket_{(E,x\mapsto\mathcal{E}\llbracket e\rrbracket_E)\,U}$$

Figure 14. Feed Language Semantics.

the set A_t of (meta-data, payload) pairs fetched at time t. For each such set, it computes the dependency set DS_t, which collects the dependencies of all the items fetched at time t. The set G_t collects all the good items from A_t. If this set is non-empty, we use the function earliest to choose the (meta-data, payload) pair (m,v) with the earliest arrival time from G_t. (We assume that there is always one such earliest item.) In this case, we set the nested meta-data $nest_t$ to be the nested meta-data of m, and the payload value v_t to be v. If the set of good values is empty, then we set the nested meta-data to indicate that at time t, we created (location = `nowhere`) a payload value that had no actual arrival time. In this case, the payload value v_t is just `None`.

The meaning of the empty feed is the empty set. The meaning of the singleton feed $\texttt{One}(e_v,e_t)$ is a single pair, the payload portion of which is the meaning of e_v. The meta-data indicates the scheduled time is the meaning of e_t, the dependency set is empty, the data came from `nowhere` (a dummy location indicating the value was generated internally), and the arrival time matched the scheduled time. A schedule feed $\texttt{SchedF}(e)$ yields a feed with one payload value for each t in the meaning of the schedule e. The corresponding meta-data follows the same pattern as for the singleton feed. The union feed is the set-theoretic union of its constituent feeds. The sum feed injects the elements of its constituent feeds into a sum and likewise takes their union. It constructs compound meta-data from the meta-data of the constituent feeds in the obvious way.

The pair feed (F_1,F_2) is formed by finding for each time t all elements of F_1 at a time t (including erroneous elements) and all elements of F_2 at time t (again including erroneous elements) and generating their Cartesian product. Notice that if the schedules do not intersect, the pair feed will empty. The meta-data is constructed by combining the meta-data for the paired feeds. The semantics of the list feed $[F \mid x \leftarrow e]$ is similar to that of the pair feed except the synchronization is n-way instead of pairwise, where n is the length of the list e.

The feed comprehension $\{|F_2 \mid x \leftarrow F_1|\}$ contains payload values v_2 taken from the meaning of feed F_2 when x is mapped to (meta-data, payload) pairs drawn from the meaning of feed F_1. The dependency set for the feed comprehension includes the dependency sets of *both* F_1 and F_2. The filter feed `filter` F `with` e selects those (meta-data, payload) pairs from the meaning of F that satisfy the predicate e. Finally, the let feed `let` $x = e$ `in` F returns the meaning of feed F when x is mapped to the meaning of e.

5.3 Feed Properties

We have used our semantics to prove two key properties of our calculus. The first property, *Type Soundness*, serves as an important check on the basic structure of our definitions: Do the sets of values given by the denotational semantics have the types ascribed by our typing rules? The second property, *Dependency Correctness*, guarantees the semantics adequately maintains provenance meta-data. To be more specific, it demonstrates that a feed item depends exclusively on the locations and times mentioned in its dependency set. This theorem is crucial for users who need to track down problems in their distributed system – when they find their incoming data is bad, they need to know exactly where (and when) to look to find malfunctioning equipment or software.

Type Soundness. The type soundness theorem states that values contained in the semantics of each feed are (meta-data, payload) pairs 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 $\mathsf{meta}(\sigma)$. A similar statement is true of core feeds.

Theorem 1 (Type Soundness)

- *If* $\Gamma \vdash C : \sigma$ core *and for all* x *in* $\mathsf{dom}(\Gamma)$, $\vdash E(x) : \Gamma(x)$ *and* $\vdash U : \mathsf{time} * \mathsf{loc} \to \mathsf{time} * (\mathsf{string\ option})$ *then for all* $(m, v) \in \mathcal{C}[\![C]\!]_{E\,U}$, $\vdash (m, v) : \mathsf{meta}(\sigma) * \sigma$.
- *If* $\Gamma \vdash F : \sigma$ feed *and for all* x *in* $\mathsf{dom}(\Gamma)$, $\vdash E(x) : \Gamma(x)$ *and* $\vdash U : \mathsf{time} * \mathsf{loc} \to \mathsf{time} * (\mathsf{string\ option})$ *then for all* $(m, v) \in \mathcal{F}[\![F]\!]_{E\,U}$, $\vdash (m, v) : \mathsf{meta}(\sigma) * \sigma$.

We have proven the theorem by induction on the structure of feeds.

Dependency Correctness. In order to make the principle of Dependency Correctness precise, we must define what it means for two universes to be equal relative to a dependency set ds. Intuitively, this definition simply states that the universes are equal at the times and locations in ds and unconstrained elsewhere.

Definition 2 (Equal Universes Relative to a Dependency Set)
$U_1 =_{ds} U_2$ *if and only if for all* $(t, \ell) \in ds$, $U_1(t, \ell) = U_2(t, \ell)$.

Now, we need a similar definition of feed equality. In the following definitions, let S_1, S_2 range over denotations of core feeds and feeds.

Definition 3 (Feed Subset Relative to a Dependency Set)
$S_1 \subseteq_{ds} S_2$ *if and only if for all* $(m, v) \in S_1$ *such that* $m.ds \subseteq ds$, $(m, v) \in S_2$.

Definition 4 (Feed Equality Relative to a Dependency Set)
$S_1 =_{ds} S_2$ *if and only if* $S_1 \subseteq_{ds} S_2$ *and* $S_2 \subseteq_{ds} S_1$

Finally, *Dependency Correctness* states that if two universes U_1 and U_2 are identical at locations and times in ds (but arbitrarily different elsewhere) then the elements of any feed F that depend upon the locations and times in ds do not change when F is interpreted in universe U_1 as opposed to in U_2. We prove Dependency Correctness by induction on the structure of feeds.

Theorem 5 (Dependency Correctness)

- *If* $U_1 =_{ds} U_2$ *then* $\mathcal{C}[\![C]\!]_{E\,U_1} =_{ds} \mathcal{C}[\![C]\!]_{E\,U_2}$.
- *If* $U_1 =_{ds} U_2$ *then* $\mathcal{F}[\![F]\!]_{E\,U_1} =_{ds} \mathcal{F}[\![F]\!]_{E\,U_2}$.

6. GLOVES Implementation and Evaluation

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

The Compiler. The GLOVES 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.

The Runtime System. We implement each GLOVES feed as a lazy list of feed items. Following the semantics in Section 5, a feed item is a (meta-data, payload) pair, although the implementation has a more refined notion of meta-data that includes more detailed error information.

The GLOVES 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 meta-data, 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 [28] 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.

Tools Library. As explained in Section 4, we implemented the GLOVES 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 using the PADS/ML generic tool framework [11] for base feeds. Other tools depend upon external libraries. For instance, the `feed2rrd` tool requires the RRD round-robin database [22] and the `feed2rss` tool uses the XML-Light package [19] for parsing and printing XML.

Experiments. To assess performance, we measure the average time to fetch a data item (termed *network latency*), the average time to prepare the data item for consumption after fetching it from the network (termed *system latency*), and the *throughput* of the system for the CoMon feed description in Figure 4. The throughput measures the average number of items fetched and processed per second.

All the experiments were conducted on a Mac Powerbook G4 computer with a 1.67GHz CPU and 2GB memory running Mac OS

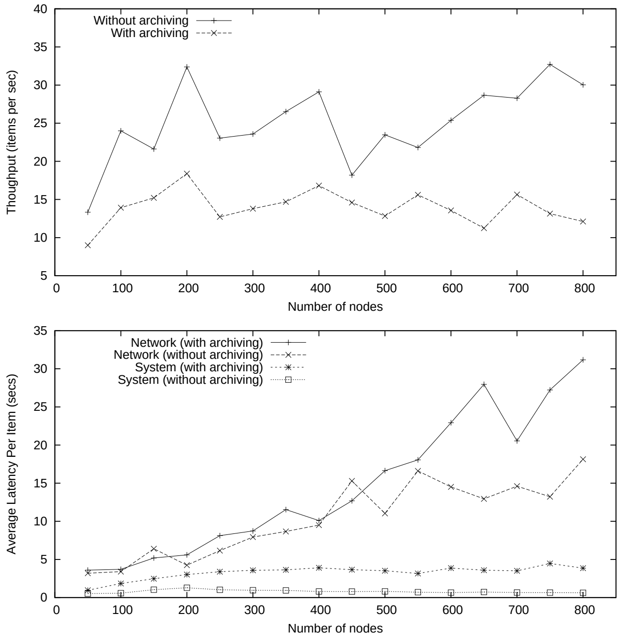

Figure 15. Average throughput and latencies per node

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, once without archiving and once 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 15 shows the average throughput and the average network and system latencies. The throughput is maximized when fetching from 200 nodes because the system supports up to 200 concurrent fetches. Archiving adds to the overhead of the system and hence reduces the throughput and increases network and system latencies. Note that while network latency increases with the number of nodes, system latency remains almost constant and relatively low, showing that the GLOVES runtime system adds little overhead to the inevitable network fetching cost. Despite the random network delays in these experiments, the network latency is generally linear in the number of nodes. The system, which we have not tried to optimize, was able to fetch data from 800 nodes and archive the results in under 70 seconds, well under the 5 minute turnaround time currently supported by CoMon. Taken together, these results suggest that GLOVES is capable of supporting PlanetLab-scale monitoring.

Language or Library. A natural question that frequently arises for domain-specific languages is whether the system is better implemented as a library or as a language extension. The strongest reason for us to implement our system as a language extension is that O'Caml (and C, and SML, and, in fact, most functional and imperative languages) have poor support for generic, type-directed programming. Unfortunately, many of our key tools, including our parsers, printers, database loaders, selectors, etc, are generic programs defined over the types of the feeds that our specifications generate. By defining a language extension, we are free to invoke a compiler to assemble the code fragments comprising the needed applications in a type-correct way.

Now, in theory, the compiler is not 100% essential to the generation of our generic programs, but in practice, it is an enormous advantage to the average programmer. After spending months studying this issue, the best alternative we have devised that does not use compiler support is to require that programmers write their specification code inside of functors parameterized in the appropriate way. These functors can then be passed off to other functors implementing appropriate tool interfaces. However this functor programming style is extremely hard to learn, to use and to explain. Avoiding these complications by creating a language-level interface seems to be a good, practical solution to the problem. For more insight into the precise issues at hand, we recommend reading related work on the construction of the PADS/ML infrastructure [11] as well as Hinze's work [15] on generic programming.

Two secondary issues influencing our choice of language over library are that (1) we could choose a pleasing and concise syntax for both our feed and tool specifications and (2) this approach allows smooth integration with PADS/ML, which itself is a successful language extension. On the latter point, developing a system in which data locality, temporal availability, format and properties are all specified in one place and in one seemlessly integrated syntax was an important goal. We believe it improves the user programming experience significantly.

7. Related Work

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

Provenance. GLOVES meta-data can be seen as an instance of provenance information, a topic of increasing interest in the database community. Cheney *et al* [8] showed how the programming language idea of dependency analysis leads to a formal theory for tracking provenance. Indeed, our Dependency Correctness Theorem reuses the definition of dependency correctness developed in their work. Our system differs from theirs in several ways, however. They treat provenance abstractly, as a collection of colors; we treat it concretely, as attestation of time, source location and error-freeness. They track provenance at the level of individual tuples in a relational calculus; we track it at the level of files, leading to reduced overhead. They simply track the provenance information; we permit programmer code to view and respond to such information.

Stream Processing. There has been a large body of work in data stream processing and work flow management [14]. For instance, languages such as Lustre [7], SIGNAL [4] and Functional Reactive Programming (FRP) [9, 29] are designed to implement synchronous systems that react to continuous or discrete signals. These signals are time-indexed values that can be composed or decomposed using various combinators. Our work on GLOVES is complementary to these efforts in that the primary purpose of GLOVES is to bridge between such systems and the messy, outside world. GLOVES provides a way to robustly internalize external, distributed data while tracking error conditions and maintaining provenance in a comprehensive manner so that programmers can subsequently use, for example, the elegant abstractions of events, behaviors and signals from FRP.

Web Mashups. Web Mashup languages such as MashMaker [10] and Yahoo Pipes [33] allow 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 GLOVES, 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, GLOVES and mashup languages also have the potential to be complementary, with GLOVES descriptions serving to define new ad hoc data sources for mashups. In fact, this idea motivated the design and implementation of the GLOVES ad hoc-to-RSS conversion tool.

Systems monitoring. One early and widely-used protocol for system monitoring is SNMP, the simple network management protocol [6], which is supported by commercial tools such as HP's OpenView [2] and free tools such as MRTG [23]. 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 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 GLOVES 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.

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 GLOVES 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 that this semantics is dependency correct. Finally, we demonstrate experimentally 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] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language SIGNAL. In *PLDI*, pages 163–173, 1995.

[5] 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.

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

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[8] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Database Programming Languages*, volume 4797, pages 138–152. Lecture Notes in Computer Science, 2007.

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

[14] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[15] R. Hinze. Generics for the masses. In *ICFP*, pages 19–22, 1998.

[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] C. Myers, D. Barrett, M. Hibbs, C. Huttenhower, and O. Troyanskaya. Finding function: evaluation methods for functional genomic data. *BMC Genomics*, 7:187, 2006.

[21] C. Myers, D. Robson, A. Wible, M. Hibbs, C. Chiriac, C. Theesfeld, K. Dolinski, and O. Troyanskaya. Discovery of biological networks from diverse functional genomic data. *Genome Biology*, 6(13), 2005.

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

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

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

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

[26] R. Sealfon, M. Hibbs, C. Huttenhower, C. Myers, and O. Troyanskaya. GOLEM: An interactive graph-based gene ontology navigation and analysis tool. *BMC Bioinformatics*, 7:443, 2006.

[27] 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.

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

[29] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI*, pages 242–252, 2000.

[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.