Lambda Calculus as a Workflow Model

Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn
School of Computer Science
University of Adelaide
South Australia 5005, Australia
{pmk,paulc,andrew}@cs.adelaide.edu.au

Abstract

Data-oriented workflows are often used in scientific applications for executing a set of dependent tasks across multiple computers. We discuss how these can be modeled using lambda calculus, and how ideas from functional programming are applicable in the design of workflows. Such an approach avoids the restrictions often found in workflow languages, permitting the implementation of complex application logic and data manipulation.

This paper explains why lambda calculus is an appropriate model for workflow representation, and how a suitably efficient implementation can provide a wide range of capabilities to developers. The presented approach also permits high-level workflow features to be implemented at user level, in terms of a small set of low-level primitives provided by the language implementation.

1 Introduction

Workflow systems [17] have emerged in recent years as tools for building distributed applications involving the coordination of software components at different sites. In scientific fields, these are generally based on a data-oriented model, where a series of side effect free operations are performed on a collection of input data to produce a result. Each operation is realised as a *task* that is executed on a remote computer, and invoked by the workflow engine over the network. Tasks are generally either jobs submitted via batch queuing systems such as Condor [15] or Globus [8], or invocations of RPC-style services using protocols such as SOAP. Examples of workflow systems include DAGMan [15], Chimera [9], Taverna [6], and Kepler [5].

A *model of computation* specifies the way in which the task invocation is carried out at an abstract level. Usually, this is based on a set of data dependencies between tasks, so that a given task only executes once all of its inputs are available, and the outputs are sent to other tasks. These can be expressed as a directed acyclic graph (DAG), which can be edited graphically or textually. This model is sometimes extended with additional features for loops, conditional tests, and nested workflows.

Many of these systems are based on models of computation which only support a limited set of programming constructs. Typically, these models are either not Turing complete, or do not support fine grained computation. This means that only very simple programming logic can be implemented in the workflow language, and any complex work must be carried out by external jobs or services written in more powerful programming languages. This is a problem for complex workflows, because it sometimes means that a developer must switch to a different language to perform certain actions, such as data manipulation or intermediate computation on the values exchanged between different tasks.

This issue can be addressed by using a more flexible model of computation which is Turing complete and can be implemented in such a way that both high-level and low-level programming is equally well supported. It is preferable that such a model maintains the advantages of existing workflow models, such as explicit data dependencies, lack of side effects, and a level of abstraction above that of mainstream imperative programming languages. In this paper, we discuss the suitability of lambda calculus for expressing workflows, and how it can meet these requirements.

1.1 Lambda calculus

Lambda calculus [2] is an abstract model of computation which is the theoretical foundation of functional programming. It specifies a notation in which functions are defined as *lambda abstractions*, consisting of a set of arguments and a body, such as the following:

```
(λa.λb.λc.b (a c c))
```

When such a function is applied to arguments, its body is instantiated by replacing all variable references with the

supplied arguments. Evaluation proceeds via a sequence of *reductions*, each of which transforms the expression into a version that is closer to its *normal form*, or result value:

```
   (λa.λb.λc.b (a c c)) + sqrt 8
=> sqrt (+ 8 8)
=> 4
```

Although the basic lambda calculus model does not include any built-in operations or data types, any language implementation based on the model will provide some set of primitives. These may be coarse grained, such as operations to invoke web services with data values representing XML trees, or fine grained, such as arithmetic operations and numeric values. The set of primitives is an implementation choice which is independent of the basic model of program representation and evaluation.

The most common technique for evaluating lambda calculus is *graph reduction* [11]. In this model, the program is represented in memory as a graph, much like a syntax tree produced by a parser. Evaluation proceeds by traversing the graph to locate expressions that can be reduced, and replacing the relevant graph nodes with the result obtained from evaluation of the expression. This graph representation has similarities with DAG-based workflows, as we shall see in section 3.

An important property of the lambda calculus is that the result of evaluation is independent of the order in which the reductions are performed. This is known as the Church-Rosser theorem [3], and is what enables separate parts of the graph to be safely reduced in parallel. In the context of workflows, this means invoking multiple tasks at the same time. This relies on the assumption that all tasks are side effect free, which is typically the case for scientific workflows.

1.2 Motivation

There are several advantages to using lambda calculus as a model for expressing workflows. Firstly, text-based representations scale better than graphical views, which can be an issue for complex workflows. Text editors are easily capable of handling large files with hundreds of lines of code, while graphical workflow editors generally do not provide an interface that deals with such complexity easily. Expressions are also a more compact syntax than separate node and edge declarations.

Secondly, the use of lambda calculus as a computational model lends itself to the application of theory developed in the context of functional programming. Common tasks such as iteration, tail recursion, and conditional expressions can easily be expressed with this model, as can data structure manipulation.

Finally, we can apply implementation techniques from the functional programming community to execute workflows more efficiently by compiling them into executable code, which is particularly important for complex workflows including data transformation and application logic.

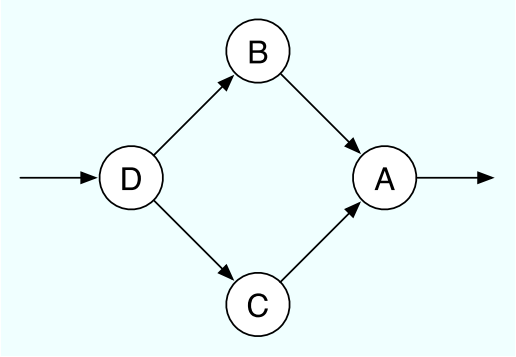

Figure 1. Task dependency DAG

2 Graph types

Task dependency graphs, sometimes referred to as directed acyclic graphs (DAGs), contain nodes specifying tasks, and edges representing data dependencies. The tasks are executed in such an order that if task A depends on task B, then B will be executed first, with its output data being transferred to A. Each task can be viewed as a "function" taking a set of input values and producing an output value. In implementation terms, function evaluation corresponds to invoking a computation on a remote resource. This model is typically referred to as *dataflow*.

Figure 1 shows an example workflow. Task A depends on B and C, which both depend on D. Thus the execution order will be D first, then B and C (either sequentially or concurrently), and finally A. The "result" of the workflow will be the output data generated by A.

An alternative model is a graph of application nodes. In this model, a task (or equivalently, function call) is modeled as a sequence of *application* nodes, one per input. The last link in the chain is a reference to a function (which may be executed remotely), and each of the input parameters are also graphs. Figure 2 shows the above workflow in this model, with application nodes represented by the character @.

The key differences with this model are that nodes can correspond to data values, and functions are treated as values. Functions can thus be passed around as parameters and returned as results from so-called *higher order* functions. This property enables a wide range of useful techniques that are common in functional programming, and is also useful for modeling abstract workflows, as described in Section 4.7.

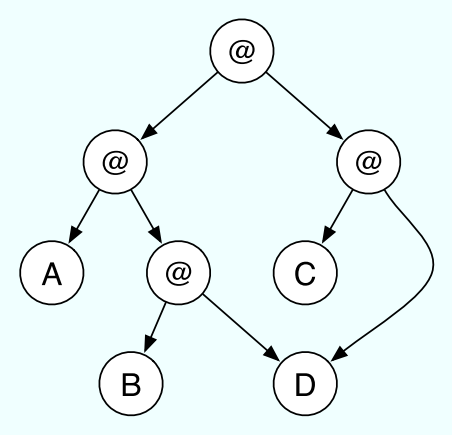

Figure 2. Function application graph

3 Workflow representation

There are several common ways of representing the information that specifies a workflow structure. One that is common in many workflow systems is a graphical representation, in which task nodes are represented as shapes, with lines and arrows indicating dependencies between them. This representation is aimed at end users who utilise a graphical editor to construct their workflow. Figure 1 is a typical example of this representation.

Even if a graphical editor is used, a text-based format is also present for storing the file on disk. An obvious serialisation of the graphical view is to represent the graph by a series of statements defining the set of nodes, along with another set defining the edges. The statements may be represented either as XML elements, or using some other syntax. An example of the graph from Figure 1 represented in this way is the following:

```
node A
node B
node C
node D
edge B -> A
edge C -> A
edge D -> B
edge D -> C
```

An alternative way of representing a graph is as an *expression*, in which each node is specified by writing its name, followed by a list of parenthesised sub-graphs that point to it. In fact, this corresponds exactly to the syntax used by the lambda calculus. The same graph expressed in this manner would be as follows:

```
A (B D) (C D)
```

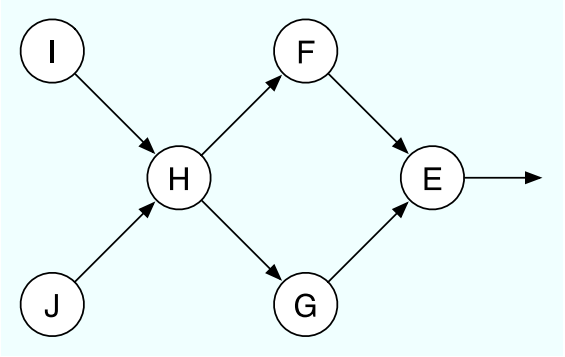

Figure 3. Graph requiring a let expression

This representation is more concise, and closer to the way in which programmers normally write expressions. However, it does not properly handle tasks whose output is sent to more than one destination, as is the case with D. The syntax is still acceptable in this example, since D takes no inputs. However, tasks which do take inputs, such as H in Figure 3, need a better solution. This problem can be solved using a lambda abstraction, by binding a symbol to the sub-graph, and referencing it where necessary:

```
(λx.E (F x) (G x)) (H I J)
```

It is straightforward to extend the syntax of lambda calculus with *let* expressions, which can be transformed at parse time into the above format, e.g:

```
let x = (H I J) in (E (F x) (G x))
```

Each of these graphical and textual formats is capable of representing the same information, and transformation from one to another is straightforward. A workflow engine that uses lambda calculus as its input language can execute workflows that have been exported from graphical editors, or translated from other DAG-based workflow languages.

Converting these workflows into lambda calculus does not alter their high-level nature. We have still described nothing about what each of the tasks actually does, how the workflow engine causes them to be invoked, or the way in which data is passed between them. These are still details which we consider to be abstracted away; the only difference is the notation in which we have expressed the workflows. In Section 4, we will discuss how these functions may be implemented in concrete terms.

4 Modeling workflow features

A model of computation specifies the abstract nature of tasks and how they are used together during execution, but does not specify how those tasks are actually implemented. Any workflow system must provide facilities for invoking a

task on a remote machine. In most cases, this invocation is provided as a primitive within the workflow language, and implemented explicitly within the execution engine. This is unavoidable for workflow languages that are not Turing complete, because the low-level interaction with the remote host often requires language facilities that are only present in more powerful languages.

Because lambda calculus is capable of expressing arbitrarily complex computations, an efficient implementation that provides a few basic data types and low-level operations can enable task invocation to be implemented in terms of the workflow language itself. Instead of coarse grained primitives for launching tasks, fine grained primitives for low-level operations like establishing TCP connections and writing binary data may be used to build up higher level abstractions for task invocation. This approach is analogous to a micro kernel based operating system, in which all high-level functionality is implemented at user level on top of basic primitives provided by the runtime environment exposed by the kernel.

The workflows discussed in the previous section represented each task as a function application, such as `(B D)`, which is a call to the task B, supplying the output of D. These functions were not defined, but assumed to be mapped by the workflow engine into remote tasks. This mapping can be achieved by making each function a built-in primitive, or defining those functions in the workflow language as expressions which call lower level networking primitives to invoke the remote computation. In this section, we describe how this latter approach can be used to provide much of the functionality that is normally implemented as primitives by existing workflow engines.

4.1 Primitives

In order to implement high-level workflow functionality in terms of low-level operations, it is necessary for the language implementation to provide a small number of basic primitives. These should include capabilities for manipulating data at the byte level, performing arithmetic operations, and manipulating data structures. Table 1 lists a possible set of operations. The built-in data types should include at least numeric values, cons pairs, and nil values.

While the basic lambda calculus can theoretically be used to express any computation, primitives like these are necessary in order to achieve reasonable performance. Additionally, many optimisations can be made within the language implementation to gain efficiency, such as storing cons lists internally as arrays, and compiling expressions into executable code. As we are interested primarily in the computational model, we will not go into further details of these optimisations here.

Many abstractions can be built up from these primitives

`+, -, *, /, %`	Arithmetic functions
`==, !=, <, <=, >, >=`	Numeric comparison
`and, or, not, if`	Logical & conditional
`cons, head, tail`	List operations
`connect`	Network connections

Table 1. Primitive operations

that can be used to support high-level workflow functionality. Cons lists can be used to support a wide range of data structures, such as strings (lists of character codes), binary data (lists of byte values), and XML trees (lists of element objects containing strings and other elements). Conditionals are supported by the `if` function; iteration can be achieved using tail recursion, and map, filter, and reduction operations can be defined in terms of the built-in list operations. These techniques are well known in the functional programming literature.

The `connect` primitive deserves special attention. We define it as a built-in function taking three parameters: host, port, and output stream. The first two are used when establishing a connection, while the third is a cons list supplied by the program, which the runtime system writes out to the connection. The result of a call to the `connect` function is itself a cons list, which can be read from lazily to obtain data from the connection. It can be treated as a string by interpreting each value in the list as a character code, which is useful for implementing text-based protocols such as HTTP.

4.2 Job submission

A batch queuing system accepts a request from a client specifying details of a job to run, and then notifies the client once the job has completed. The way in which the network interaction occurs depends on the protocol used by job submission mechanism. In some cases, separate network connections may be used to submit the request and then later retrieve the result; in others, a single connection may be kept open until the job completes, and the results sent back directly. The logic to handle this interaction may be implemented as a set of support routines provided by the workflow system, using lambda calculus combined with the primitives given above.

As a simple example, consider a job queue that accepts submissions via HTTP GET requests, and sends the job output directly back to the client in the HTTP response. A script on the web server named `runjob.pl` accepts a `program` parameter specifying the name of the executable file, and an `args` parameter containing the command line arguments to be passed to the program. The code in Figure 4 invokes this script by sending it a request with the query string formed based on supplied program name and argument list. This code assumes separately defined routines

```
submit_job = (λhost.λprog.λargs.
(parse_response
  (connect host 80
    (++ "GET /runjob.pl?program="
    (++ (urlencode prog)
    (++ "&args="
    (++ (urlencode (join " " args))
        " HTTP/1.0\r\n\r\n"))))))))
```

Figure 4. Support routine for job submission

are available for string concatenation, list joins, parameter encoding, and HTTP response parsing. It also assumes that the parser converts strings into cons lists of characters.

Individual tasks can be be implemented as functions which call this routine, supplying concrete details about the job to be executed. For example, a task A that takes two string arguments could be defined as follows:

```
A = (λx.λy.submit_job "hydra"
"/home/pmk/myprog" (cons x (cons y
nil)))
```

This task may then be called from a workflow, in this case taking as input the results of tasks B and C:

```
A (B D) (C D)
```

Implementing tasks in such a manner preserves our ability to deal with them at a high level. The workflow specification remains the same as the example given in Section 3; here we have gone into one more level of detail to specify how it is realised in actual terms. The concrete implementations may be arbitrarily complex, involving all of the necessary mechanisms to invoke a job, including support actions such as staging data in or out of the remote host, and monitoring for task completion.

4.3 Services

Services can be accessed in a similar manner to the above. RPC mechanisms are generally implemented by having the client establish a connection to the server, send it a request containing the operation name and input parameters, and then having the server send back the result. For example, web services utilise the HTTP request/response mechanism and encode the parameters and results using XML and SOAP.

A workflow task corresponding to a web service invocation can be defined as a function which makes calls to support routines that submit a SOAP request to a web server. Depending on how the workflow system is implemented, the values passed between tasks could be represented as plain strings, or more complex data structures such as XML trees. It is the responsibility of the support routines to convert between these representations and the on-the-wire format.

```
postreq = (λhost.λpath.λreq.
(connect host 80
  (++ "POST "
  (++ path
  (++ " HTTP/1.0\r\n"
  (++ "Content-Type: text/xml\r\n"
  (++ "Content-Length: "
  (++ (numtostring (len req))
  (++ "\r\n\r\n" req))))))))

soapcall = (λhost.λpath.λbody.
(parse_response
  (postreq host path
    (++ "<Envelope><Body>"
    (++ body
        "</Body></Envelope>")))))
```

Figure 5. Support routines for web services

As an example, the code in Figure 5 implements a simplified version of SOAP over HTTP. The `postreq` function accepts a host and URL path, as well as a request body. It makes a HTTP POST request containing the appropriate headers. The `soapcall` function accepts the host and path, as well as the arguments to be passed to the service, which are assumed to be already encoded in XML. It wraps these in a SOAP envelope and posts the request to the server. The response is then parsed to extract the result value.

Consider a workflow task B that calls a stock quote service. It takes a single parameter specifying the symbol name, and submits a SOAP request to http://stockexchange/getquote using the above routines:

```
B = (λsym.soapcall "stockexchange"
"/getquote" (++ "<symbol>" (++ sym
"</symbol>")))
```

4.4 Shims

A common problem encountered when developing workflows is that tasks sometimes use different input and output formats. These formats may contain the same information but with a different syntax, or similar information that requires both syntactic and semantic transformations to achieve compatibility. When such conversion is needed, additional components must be added to the workflow to perform the necessary conversion. These are known as *shims* or *adapters* [7, 10].

The need for shims is due to the lack of support in many workflow languages for data manipulation. Instead of being able to access fields of an object or perform basic string

manipulation using built-in language constructs or APIs, a developer must create separate components or services to perform these tasks. Features like these, which are common in regular programming languages, involve significant effort and complexity in workflow languages. While this cost is sometimes worth paying in order to get the benefits provided by workflow languages, there are many cases where the tradeoff is questionable.

Of course, these problems could be avoided by writing the workflow entirely in a language like Java or C, but this would involve giving up other benefits like automatic parallelisation and fault tolerance that are generally provided by workflow languages. Instead, it is preferable to use a language which meets both types of requirements. Lambda calculus, combined with basic primitives such as those suggested in Section 4.1, meets this criterion.

4.5 Data parallelism

Workflows can often benefit from data parallelism, where each item in a list of values is processed separately, potentially on different computers. This sort of processing is common in task farming middleware, but is not well supported by DAG-based workflow languages. The reason is that the latter typically assume a fixed number of tasks, thus requiring sequential iteration to process lists [14]. A solution is to use the `map` construct common in functional programming languages, which can easily be expressed in lambda calculus using the `cons`, `head`, and `tail` primitives:

```
map = (λf.λlst.
if lst
  (cons (f (head lst))
        (map f (tail lst)))
  nil)
```

Such a call can be automatically parallelised by the runtime system in the case of strict evaluation, but may require explicit annotations [16] in the case of lazy evaluation. Other operations like `filter` and `reduce` can be implemented in a manner similar to the above.

4.6 Control structures

Some workflow engines provide limited ability to control the flow of execution based on data computed during the workflow. These constructs are standard features present in all major programming languages, and their implementation in functional programming languages is well known. For example, conditionals are implemented using the `if` function, which takes a boolean expression as well as true and false branches. When evaluated, it evaluates the conditional parameter and returns either the second or third argument, depending on the result.

Sub-workflows can be modeled as functions. Wherever they are used, each input link corresponds to an actual parameter, and the output link is the result of the function call. Multiple outputs can be handled by wrapping them in lists. Iteration can be modeled using tail recursion, which can be performed in constant space by most functional language implementations. Each iteration is simply another call to the same function but with updated values passed as parameters. A portion of the workflow that needs to be executed multiple times for different inputs would be expressed as an application of the `map` function to a lambda abstraction, as described in Section 4.5.

4.7 Abstract workflows

Some workflow engines support the concept of *abstract workflows*, which are workflow specifications that do not explicitly specify which resources are to be used. *Concrete workflows* are constructed by instantiating an abstract workflow with a specific set of resources. In lambda calculus, abstract workflows can be modeled as higher order functions which take parameters specifying the concrete implementations of tasks. For example, the workflow from Section 3 can be parameterised as follows:

```
(λA.λB.λC.λD.A (B D) (C D))
```

This function can then be applied to a set of function parameters which implement tasks A-D. If none of the parameters contain any free variables, it is a concrete workflow which can be directly executed. Otherwise, it is a "less abstract" workflow that contains some implementation details but is still parameterised. For example, the following abstract workflow specifies each task as a web service call to a specific operation, but is parameterised by the service URLs:

```
AWF =
(λurl1.λurl2.λurl3.λurl4.
  (λA.λB.λC.λD.A (B D) (C D))
    (λx.y.wscall url1 "a" ...)
    (λx.wscall url2 "b" ...)
    (λx.wscall url3 "c" ...)
    (wscall url4 "d" ...))
```

This abstract specification can then be instantiated into a concrete workflow by applying the function to a set of parameters specifying a specific set of services to be accessed:

```
(AWF "http://a.org/analyse"
     "http://b.org/filter"
     "http://c.org/process"
     "http://d.org/query")
```

One application of the abstract workflow concept is to provide *quality of service* (QoS) mechanisms, whereby services are chosen at runtime based on certain requirements.

For example, a user of the above workflow may want to specify that they want to use the cheapest service available that implements A, the fastest available versions of services B and C, and the most reliable version of service D. Assuming the workflow engine provides functions to determine the best choice in each case, this could be expressed as follows:

```
(AWF (find_cheapest "a")
     (find_fastest "b")
     (find_fastest "c")
     (find_most_reliable "d"))
```

Abstract workflows defined using these techniques are a flexible way of achieving reuse. A scientist may develop a workflow parameterised by service addresses, and then run it in their local environment by supplying the necessary set of URLs. The abstract workflow could subsequently be shared with other scientists, who may run it with the local versions of those services hosted at their own institution, or with a different set of input data. Such usage is equivalent to a script which uses a configuration file or command-line parameters, rather than using hard-coded information.

The ability to express abstract workflows in this manner does not require explicit support from the workflow engine. It comes about as a natural consequence of the inherent support for higher order functions that is present in lambda calculus. It is one of the many examples of how powerful functionality can be built up from a minimal set of language constructs, instead of extending a monolithic infrastructure.

5 Implementation

Although the lambda calculus model is suitable for *representing* workflows, additional requirements must be met for these representations to be *executed* in a manner that achieves the advantages typically associated with workflow engines. This combination of representation and execution requirements together defines the semantics and pragmatics of the computational environment in which workflows are executed. Such an implementation effectively constitutes an implementation of a functional programming language.

Parallel evaluation of multiple sub-expressions must be provided in order for more than one task to be in progress at the same time. Techniques such as parallel graph reduction [1] may be used to achieve this, based either on strict or lazy evaluation. In the latter case, strictness analysis [4] and/or manual annotations [16] can be used to indicate which expressions may be safely reduced in parallel.

Network connections should be supported in such a manner that they can be mapped onto data streams, and preferably exposed using a cons list abstraction. This enables the data sent to or received from a service to be treated as data, and for all of the programming techniques that are applicable to list processing to be used in the context of data streams. This is particularly useful when combined with lazy evaluation, because it enables results from tasks to be processed incrementally.

Ideally, some measure of fault tolerance should be provided, so that if a remote job or service fails, then it can be retried on another host. The side effect free nature of the lambda calculus model means that it is safe to re-execute a task, even if it was already part way through execution at the time that it failed. This property is relied on by many distributed computing systems to provide correctness even in the face of partial failures.

Finally, the execution engine should be efficient enough to support fine grained computation. This is important due to the fact that low-level primitives are used by remote task invocation mechanisms, and also to support data manipulation and intermediate computation on the data values produced and consumed by external tasks. Techniques for implementing functional languages efficiently are given in [11].

In [13], we presented an example of a workflow engine that implements many of these ideas. It supports parallelism, native code compilation, and an array-based cons list representation for efficient stream processing. We have also implemented an XSLT compiler designed for web service-based workflows [12], using the lambda calculus as an intermediate target. Our current research explores the application of these ideas. In many ways, the ideas put forward in this paper are a reduction of our experiences developing and applying the system reported in [12] and [13].

The issues addressed in this paper are solely related to the modeling and execution of workflows. The development of visual programming environments and higher level workflow languages are separate concerns that can be addressed within the overall framework we have presented here. Editors or compilers which output lambda calculus can utilise an execution engine meeting the above requirements, while exposing a different (but compatible) interface or syntax to developers.

6 Conclusion

Lambda calculus is a simple, abstract model of computation. It is independent of the granularity of operations, and can thus be applied to coarse grained workflows as well as fine grained computation. The side effect free nature of the model and incorporation of explicit data dependencies enables techniques such as parallel graph reduction to be used to coordinate the concurrent invocation of operations. When applied to workflows, this enables multiple remote operations to be in progress at the same time.

Previous research on functional programming has produced techniques for efficiently executing languages based on the lambda calculus model. These techniques can be used to build workflow enactment engines in such a way that supports features necessary for workflows, such as parallelism, but also permits arbitrarily complex, fine grained computation to be performed within the workflow language itself. This fine grained computation can be used to implement support functionality such as network protocols for launching remote tasks, and transforming data between the representations used by different tasks.

The lambda calculus model provides a solid foundation for functional programming and, by extension, data-oriented workflows. We have shown how common workflow constructs can be expressed in terms of lambda calculus, both at an abstract level and with regards to concrete implementation details. This approach to designing workflow engines achieves power through simplicity, and permits the requirements of concurrency and distribution to be met without sacrificing expressiveness.

References

[1] Lennart Augustsson and Thomas Johnsson. Parallel graph reduction with the (v , g)-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 202–213, New York, NY, USA, 1989. ACM Press.

[2] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics, 2nd Ser.*, 33(2):346–366, April 1932.

[3] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[4] Chris Clack and Simon L. Peyton-Jones. Strictness analysis – a practical approach. In J. P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 35–39, Berlin, 1985. Springer-Verlag.

[5] Bertram Ludascher et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2006.

[6] Tom Oinn et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006. Special Issue on Grid Workflow.

[7] U Radetzki et al. Adapters, shims, and glue–service interoperability for in silico experiments. *Bioinformatics*, 22(9):1137–43, May 2006.

[8] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, 2002.

[10] Duncan Hull, Robert Stevens, Phillip Lord, Chris Wroe, and Carole Goble. Treating shimantic web syndrome with ontologies. In *First Advanced Knowledge Technologies workshop on Semantic Web Services (AKT-SWS04)*. KMi, The Open University, Milton Keynes, UK, 2004. (See Workshop proceedings CEUR-WS.org (ISSN:16130073) Volume 122 - AKT-SWS04).

[11] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[12] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Distributed, parallel web service orchestration using XSLT. In *1st IEEE International Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.

[13] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. A distributed virtual machine for parallel graph reduction. In *8th International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT '07)*, Adelaide, Australia, December 2007.

[14] B. Ludäscher and I. Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks. Technical Note SciDAC-SPA-TN-2003-01, 2003.

[15] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[16] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[17] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3–4):171–200, September 2005.