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 [26] 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 [24] or Globus [9], or invocations of RPC-style services using protocols such as SOAP. Examples of workflow systems include DAGMan [24], Chimera [10], Taverna [7], and Kepler [6].

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

*This is the pre-peer reviewed version of the following article: Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Lambda Calculus as a Workflow Model. *Concurrency and Computation: Practice and Experience*, 21(16):1999-2017, July 2009. Available at: http://www.interscience.wiley.com/

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 work involving complex logic 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 [3] 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* [17]. 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 [4], 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

The main purpose of this paper is to explain how concepts from the world of functional programming are useful in the context of workflows. Existing workflow engines support only limited programming models, in which the focus is on invoking services and passing data between them, but not on other aspects like computation or data processing. We believe there are significant benefits to be gained from enhancing the capabilities of workflow systems to the point where they are just as powerful as general-purpose scripting languages. An important step towards this is consideration of the programming models upon which these systems are based.

Our philosophy is that workflow engines are simply programming language implementations which provide novel features related to concurrent and distributed programming. The notion of coordinating execution of a set of tasks is much like that of representing program logic as a set of calls to functions and operators, which is essentially the act of programming. The fact that the execution involves interaction with services over a network is largely a detail to be addressed at the implementation level. This is why programming models which are usually used only for local computation, such as the functional programming model, can also be applied to distributed programming.

Functional programming is well suited to data-oriented workflows, due to its emphasis on abstraction and side effect free computation. Both workflow engines and functional language implementations go to significant effort to shield the programmer from low-level execution details, allowing them to focus on *what* the program does rather than *how* it does it. Extensive work has been done on parallel execution of functional languages [12], leveraging the lack of side effects to safely execute different parts of the program in parallel without introducing non-determinism. Much of this work is of relevance to the implementation of workflow engines.

Lambda calculus is the basic programming model upon which functional programming languages are based. Understanding the simplicity and elegance of this model is key to realising how a very wide range of programming capabilities can be supported in a language with only a very small feature set. In this paper we discuss several common features of workflow languages that can be implemented in terms of the lambda calculus. Some of these rely on certain primitive operations being provided, while others depend on certain execution semantics, but the model itself does not require extension to support these features.

The syntax of lambda calculus is very concise, and reasonably close to what programmers are accustomed to. Many workflow languages use XML as a representation format, which we consider to be a poor choice for programming language syntax due to its verbosity. Although these formats are typically intended to be read and written by graphical editors, there is much to gain (and little to lose) by choosing a more compact syntax suitable for

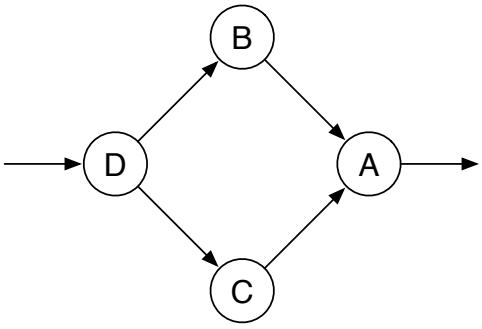


Figure 1: Task dependency DAG

editing by hand. Conciseness is valued by programmers, who are more likely to choose a language that readily facilitates program development. This issue is explored further in the next section.

It is also worth mentioning that the basic lambda calculus itself, while capable of being used to define a workflow, is usually extended by implementations with additional features. These include syntactic sugar, along with built-in operations and data types. The latter are addressed in Section 4.1.

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 4.8.

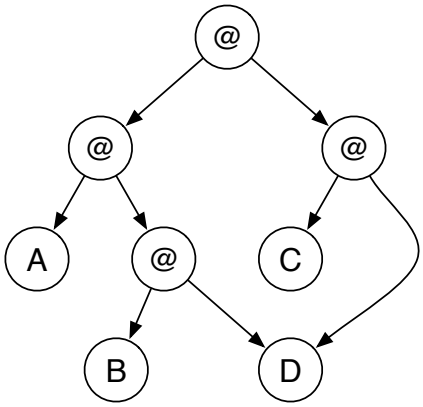


Figure 2: Function application graph

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 supported 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)
```

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

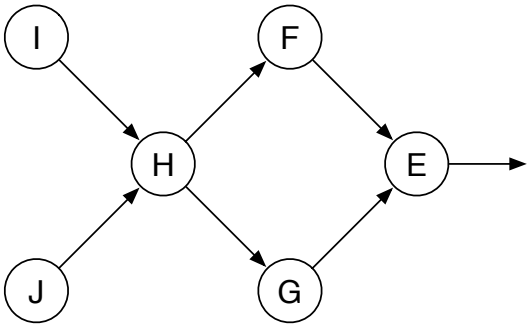


Figure 3: Graph requiring a let expression

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

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

Table 1: Primitive operations

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 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 [1].

4.2 Network connections

The `connect` primitive listed in Table 1 deserves special attention. We argue that for a data-oriented workflow model in which all remote operations are free of side effects, it makes sense to treat the network connections that invoke these operations as side effect free mappings from input to output. This can be achieved by providing a function which accepts the data to be sent across the connection as an input parameter, and returns the data received from the connection.

This approach differs from that used by other functional languages, which make the conservative assumption that all I/O operations involve side effects, and must therefore be executed in an explicitly defined sequential order. These languages typically provide finer-grained operations for dealing with connections, often separating the establishment of a connection from subsequent reads and writes. Our approach instead provides a higher-level view of a connection, which we believe is more appropriate for the functional programming model. This does however rely on the programmer guaranteeing that they will only access side effect free services.

We define `connect` as a built-in function taking three parameters: host, port, and input stream. The first two are used when establishing the connection, while the third is a cons list supplied by the program, which the runtime system sends across the connection. The result of a call to the `connect` function is itself a cons list, which can be read from 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.

4.3 Job submission

Tasks used within workflows are sometimes realised as *jobs*, which involve execution of a program installed on a remote host. Batch queuing systems like PBS [13], Condor [24], and Sun Grid Engine [11] provide a network-based interface via which specified programs can be launched. The request messages used to launch these jobs contain details of the program to run, as well as command line arguments and names of input files. Invoking these tasks from within a workflow requires the workflow engine to open a connection to the job submission system, and send appropriately-formatted request data based on the input parameters to the task. Once the job has finished, a response is sent back to the client.

The way in which the network interaction occurs depends on the protocol used by the 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

```
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

it a request with the query string formed based on supplied program name and argument list. This code assumes separately defined routines 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 shown below, where `x` and `y` represent the arguments:

```
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.

4.4 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.

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

```
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

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>")))
```

4.5 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 transformation 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* [8, 15].

The need for shims is due to the lack of support in many workflow languages for the required 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 mainstream 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 of workflow languages, such as automatic parallelisation and fault tolerance. 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.

4.6 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 [21]. 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)
```

It is possible to implement the runtime system in such a manner that code such as this is automatically parallelised. Other operations like `filter` and `reduce` can be implemented in a manner similar to the above.

4.7 Control structures

Some workflow engines provide a 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 4.6.

4.8 Abstract workflows

Some workflow engines support the concept of *abstract workflows* [5, 22], 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 can 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.

4.9 Support for higher-level workflow languages

A workflow engine based on the model of lambda calculus may be used to execute other workflow languages, by writing compilers for those languages which generate lambda calculus as output. This essentially treats lambda calculus as an intermediate language in a similar manner to Java bytecode, although at a higher level of abstraction. Both existing workflow languages and future ones could be supported in such a manner. The key requirement that input languages would need to meet are that they only use side effect free computation.

There are a variety of existing workflow languages which could be compiled into such an intermediate language. Those which are based on task dependency graphs are good candidates, since they have a straightforward translation into lambda calculus, as described in Section 3. More complex languages which provide a richer feature set but are still based on purely side effect free computation would also be suitable. We are currently developing an implementation of XQuery [2] based in this approach, for specifying workflows that utilise XML-based web services [20].

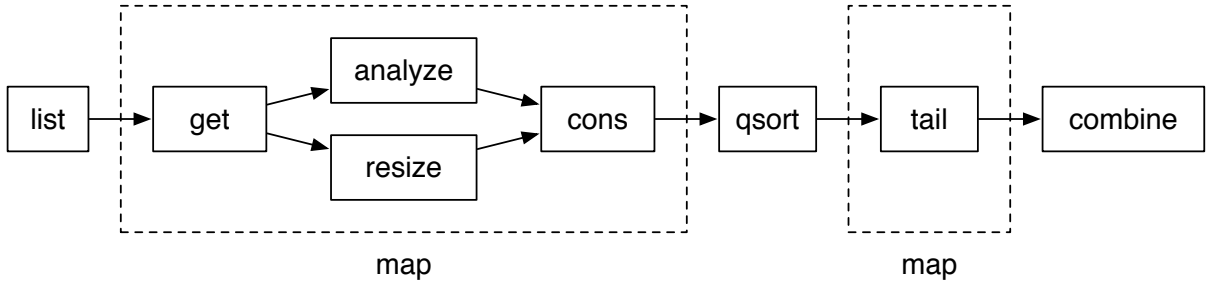
5 Example

Figure 6 shows an example of an image processing workflow expressed using the approach described above, which has been successfully run on the implementation described in [20]. The workflow does the following:

1. Obtain a list of images available from a service (using the `list` task)
2. Retrieve each image (using the `get` task)
3. Analyse each to find the average hue, saturation, and value (using the `analyze` task)
4. Resize each image to produce a 75×75 thumbnail (using the `resize` task)
5. Sort all the thumbnails by their hue, achieving a spectrum-like effect
6. Combine all the thumbnails into one large, tiled image (using the `combine` task)

The first part of the code specifies the basic structure of the workflow, which is mirrored in the diagram shown above the code. This uses several common functional programming idioms such as `map`, to apply a function (lambda abstraction) to a list of items, and `cons`, to combine two values into a single data structure. The cons pairs are pulled apart during sorting, when it is necessary to compare the hue values of the images, and prior to the `combine` function, where it is necessary to extract just the thumbnails from the (then sorted) list. The syntax used here is a slightly modified version of that described above, in which `!` is used in place of the λ symbol (for use in ASCII text), and function definitions $f = \lambda x_1 \ldots \lambda x_n.e$ can also be written as $f\ x_1 \ldots x_n = e$.

Below the main workflow definition is the implementation of each task. The service invoked by this workflow uses a simple command invocation protocol in which a request




```
/* Workflow */
main = (letrec
   analyzed = (map (!name.
    (letrec img = (get name)
     in (cons (analyze img)
              (resize 75 75 img))))
    list)
   sorted = (qsort huecmp analyzed)
   thumbnails = (map tail sorted)
in (combine thumbnails))

/* Service operations */
list = (split
(connect "localhost" 5000 "list\n"))

get name = (connect "localhost" 5000
(++ "get " (++ name "\n")))

analyze img = (map ston (split
(connect "localhost" 5000
(++ "analyze\n" (++ (ntos (len img))
(++ "\n" img))))))

resize width height img =
(connect "localhost" 5000
(++ "resize " (++ (ntos width)
(++ " " (++ (ntos height)
(++ "\n" (++ (ntos (len img))
(++ "\n" img))))))))

combine imgs =
(connect "localhost" 5000
(++ "combine " (++ (ntos (len imgs))
(++ "\n" (foldr ++ nil
  (map (!img.++ (ntos (len img))
                (++ "\n" img))
       imgs))))))

/* Sorting comparators */
huecmp x y =
(- (hue (head x)) (hue (head y)))

/* Data structure accessors */
hue x        = (item 0 x)
saturation x = (item 1 x)
value x      = (item 2 x)

/* Response parsing (list/analyze) */
split s = (split1 s "")

split1 s got =
(if s
    (if (isspace (head s))
        (cons got
              (split1 (tail s) ""))
        (split1 (tail s)
                (++ got (cons (head s)
                              nil))))
    nil)
```

Figure 6: Image processing workflow

consists of a space-separated sequence of tokens, specifying the operation name and arguments in the same manner as the UNIX command line. Some operations accept additional binary image data, which is encoded as a length string followed by the data itself. The `get`, `resize`, and `combine` operations return binary image data, and the `list` and `analyze` functions return a space-separated sequence of tokens, corresponding to the list of images and analysed image properties, respectively. Each of the task implementation functions composes the request and parses the responses as appropriate.

The remaining parts of the code contain helper functions used by the workflow. `huecmp` is a comparator function used when sorting, to determine the relative ordering of images. The `hue`, `saturation`, and `value` functions access the corresponding fields of the data structure returned by `analyze` (represented as a cons list). The `split` function is used to parse the space-separated token strings returned by `list` and `analyze` into cons lists. Other standard helper routines such as `map`, `foldr`, and `qsort` are not shown here for space reasons. However, they are defined in using the same notation used for the workflow itself. `ntos` and `ston` are built-in functions which convert between numbers and strings.

6 Implementation

Our discussion so far has addressed only the *definition* of workflows. In order to *execute* workflows in an appropriate manner, it is necessary for the language implementation to meet certain requirements. In designing such an implementation, it is useful to consider the approaches that have been used for executing existing functional languages, which also use the lambda calculus as their underlying programming model. In particular, parallel implementations of functional languages are relevant to workflows, since there are often performance benefits to be gained from having multiple external tasks running in parallel.

In this section, we discuss properties we consider useful for implementing the model described in this paper, based on experience gained from building our own prototype. Our approach is not necessarily the only way of implementing the model, however we believe the features listed here are important for appropriately treating workflows as functional programs. The specifics of our prototype implementation are outside the scope of this paper, but are discussed in [20].

6.1 Execution efficiency

Section 4 introduced the idea that support features required by workflows, such as the ability to invoke remote tasks, could be implemented directly in the workflow language, rather than being part of the execution engine. This relies on the presence of built-in operations for performing numeric and string manipulation operations, as well as the ability to create and manipulate data structures. These language features can be used to perform light-weight computation and data transformation on the results and input values passed between tasks.

A workflow engine designed to support internal computation should aim to achieve reasonable execution speeds, ideally comparable to that of mainstream scripting languages. Automatic parallelisation introduces overheads, and it is necessary for implementors to

consider the tradeoffs in performance it involves. However, we have found that it is possible to keep these overheads low enough to avoid making local computation prohibitively expensive. Workflows generally delegate compute-intensive work to external tasks, so the overheads of parallelism are much less of a factor in overall performance.

Many functional language implementations work by compiling source code into an intermediate representation, which consists of a sequence of instructions that can be executed by an abstract machine. One of the first such examples of this was the G-machine [16], which provided an abstract instruction set that modeled the operations performed on a graph during graph reduction. Source code compiled into G-machine bytecode can then be executed by this abstract machine, which can be implemented either as an interpreter or a native code compiler. This approach provides significant performance advantages over direct manipulation of the graph, since it avoids many of the runtime checks and intermediate transformation steps that would otherwise be necessary. A detailed guide to implementing functional languages in this manner is given in [17].

6.2 Automatic parallelisation

Parallel task invocation is also important for workflows. Since the tasks are executed by remote machines, it is often beneficial to have several of them in progress at the same time, provided all the relevant data dependencies are respected. When an invocation request is made to a service, the workflow execution should not block, but should instead continue on with another part of the graph which does not depend on the requested operation. This is in contrast to traditional RPC toolkits which expose remote service operations as blocking calls, forcing the client to remain idle while the request is in progress.

A graph representation of a lambda calculus expression contains explicit data dependencies, making it possible for automated analysis to determine which expressions can safely be evaluated in parallel. In the context of workflows, evaluation of an expression can cause invocation of a remote task. Since at any point in time there may be multiple expressions eligible for evaluation, it is possible to exploit parallelism by having several external tasks running at the same time. This "external parallelism" can be achieved even when the workflow engine itself is running on a single processor, because the ability to context switch between evaluation of different parts of the graph enables the use of multiple machines across the network to exploit parallelism.

Ideally, the cases in which multiple expressions can be evaluated in parallel should be automatically determined by the language implementation. This relieves the programmer from having to manually work out which parts of the code should run in parallel, and maintains the ease of use present in existing workflow languages. As discussed in Section 6.4, existing functional languages usually do not perform automatic parallelisation, due to efficiency problems with highly compute-intensive code. However, since the workflows we are concerned with delegate most of their computation to external programs or services, some cost to local execution performance is acceptable. Automatic parallelisation is thus practical for workflows, even when they involve some degree of internal computation.

6.3 Access to network connections

Section 4.1 discussed the representation of network connections as pairs of data streams, with a `connect` function providing a mapping from input data to output data. This is conceptually a "pure" function, assuming that the service being accessed is itself free of side effects. Implementation of this function, however, involves dealing with state changes relating to connection establishment, read/write calls, and buffering.

Ideally, the programmer should be able to access the data streams in the same manner as normal cons lists. Once a data stream has been materialised in memory, it should be possible to traverse the stream and access portions of it just like any other list. For data streams that have not yet been fully read, the tail of the list can refer to a suspended expression which, when accessed, will cause the calling expression to block until that data becomes available[1].

Blocking should be handled in such a way that when an attempt is made to read data that is not yet available, only the expression(s) that depend on that data are blocked. Other parts of the graph may still be eligible for execution if they have local computation to do or are accessing other connections. Effective support for parallelism relies on this, since workflows will often need to have multiple service invocations in progress at the same time.

It may be necessary for the language implementation to provide certain internal functions which do actually utilise side effects, but expose an interface to user code that makes them appear as pure functions. This is acceptable, as long as the programmer is only given access to the abstraction of network connections as side effect free transformations from input to output.

6.4 Comparison with existing functional languages

Since we are proposing the use of a functional programming model for workflows, the question arises as to whether it would be sufficient to simply use an existing functional language implementation instead. Ideally this would be the case; however, certain limitations of existing functional languages mean that they do not fully meet the requirements given above. We argue that it is instead better to study these implementations and use ideas from them to design new execution engines that cater for the specific needs of workflows.

The potential for parallel execution is one of the most attractive aspects of functional programming, and this area has seen a great deal of attention over the years. The suitability of these languages for parallel programming follows from the Church-Rosser theorem [4], which states that the result of evaluation of an expression is independent of the order in which the reductions are performed[2]. Many functional language implementations have exploited this idea, resulting in parallel programming capabilities that are significantly easier to use than in imperative languages, due to the fact that the programmer does not have to explicitly control synchronisation between threads.

[1] This makes use of *suspended evaluation*, a feature common to many functional programming languages.

[2] Assuming that all possible orderings lead to termination

Automatic parallelisation has been attempted by several language implementors, but it is now generally viewed as an impractical goal [19]. This is because for fine-grained computation, the per-expression overheads of managing and scheduling parallelism can sometimes *decrease* overall performance when running on multiple processors. Current approaches to mitigating these issues involve explicit annotations that must be supplied by the programmer [25], but this increases the difficulty of writing parallel programs. However, since workflows delegate most of their computation to coarse-grained external tasks, the overheads of managing parallelism can be kept much lower. Existing functional language research has not explored this idea however, since it has focused primarily on programs that perform only internal computation. Existing workflow engines have already demonstrated the practicality of automatic parallelisation for the case of coarse grained tasks.

I/O is another important implementation concern. Workflows require the ability to have multiple I/O operations executing in parallel, to take advantage of parallel execution of remote tasks. However, existing functional languages treat I/O as "impure", since it may involve side effects. Mechanisms such as streams [23], continuations [14], and monads [18] all require I/O operations to be executed in a strictly defined order, which precludes the possibility of invoking multiple remote calls in parallel. A workflow language based on our proposed model should permit access to network connections from within pure expressions, and enable concurrent execution of I/O operations on different connections. The deterministic behaviour of the workflow then relies on the programmer guaranteeing that all of the services accessed are indeed free of side effects.

7 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] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, July 1996. Second edition.

[2] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language. W3C recommendation, World Wide Web Consortium, January 2007.

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

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

[5] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[12] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 1999.

[13] Robert L. Henderson. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg, 1995.

[14] Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely-functional I/O systems. Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, March 1989.

[15] 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).

[16] Thomas Johnsson. Efficient compilation of lazy evaluation. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 58–69, New York, NY, USA, 1984. ACM Press.

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

[18] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[19] Simon Peyton Jones. Foreword. In Kevin Hammond and Greg Michelson, editors, *Research Directions in Parallel Functional Programming*, pages xiii–xv. Springer-Verlag, London, UK, 1999.

[20] 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.

[21] 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.

[22] Stephen McGough, Laurie Young, Ali Afzal, Steven Newhouse, and John Darlington. Workflow enactment in ICENI. In *UK e-Science All Hands Meeting (AHM 2004)*, Nottingham, UK, August 2004.

[23] John T. O'Donnell. Dialogues: A basis for constructing programming environments. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 19–27, New York, NY, USA, 1985. ACM.

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

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

[26] 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.