JFP **27**, e22, 31 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S0956796817000119

Computation semantics of the functional scientific workflow language Cuneiform[*]

JÖRGEN BRANDT, WOLFGANG REISIG and ULF LESER
Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany
(*e-mails:* brandjoe@informatik.hu-berlin.de, reisig@informatik.hu-berlin.de, leser@informatik.hu-berlin.de)

Abstract

Cuneiform is a minimal functional programming language for large-scale scientific data analysis. Implementing a strict black-box view on external operators and data, it allows the direct embedding of code in a variety of external languages like Python or R, provides data-parallel higher order operators for processing large partitioned data sets, allows conditionals and general recursion, and has a naturally parallelizable evaluation strategy suitable for multi-core servers and distributed execution environments like Hadoop, HTCondor, or distributed Erlang. Cuneiform has been applied in several data-intensive research areas including remote sensing, machine learning, and bioinformatics, all of which critically depend on the flexible assembly of pre-existing tools and libraries written in different languages into complex pipelines. This paper introduces the computation semantics for Cuneiform. It presents Cuneiform's abstract syntax, a simple type system, and the semantics of evaluation. Providing an unambiguous specification of the behavior of Cuneiform eases the implementation of interpreters which we showcase by providing a concise reference implementation in Erlang. The similarity of Cuneiform's syntax to the simply typed lambda calculus puts Cuneiform in perspective and allows a straightforward discussion of its design in the context of functional programming. Moreover, the simple type system allows the deduction of the language's safety up to black-box operators. Last, the formulation of the semantics also permits the verification of compilers to and from other workflow languages.

1 Introduction

In many current scientific projects, the need to analyze large and heterogeneous data sets in distributed compute environments is prevalent (Hey *et al.*, 2009). Systems supporting the definition, execution, and management of such data analysis programs are called scientific workflow systems (Cohen-Boulakia & Leser, 2011; Liu *et al.*, 2015). They typically bundle a scientific workflow language and a workflow execution environment. The main purpose of such workflow languages is to facilitate the formulation of assemblies made up of independent computations operating on large and heterogeneous data sets. This leads to a unique set of requirements when

* This work is funded by the EU FP7 project "Scalable, Secure Storage and Analysis of Biobank Data" under Grant Agreement no. 317871. We also acknowledge funding by the Humboldt Graduate School GRK 1651: SOAMED.

compared to general purpose programming languages or scripting languages: (i) It must be easy to specify a complex data analysis procedure as an assembly of independent subprograms that can run in parallel. (ii) It must be easy to integrate existing tools and libraries with heterogeneous programming interfaces, because the individual steps in a scientific workflow are often complex programs in themselves which are developed by different groups around the world. Finally, (iii) analysis procedures often require conditional branching and iterative processing.

Cuneiform is a programming language for the specification of scientific workflows focusing on parallel execution, integration of arbitrary external programs, and support for a rich set of control elements (Brandt *et al.*, 2015). It has been applied in a variety of scientific domains including remote sensing, machine learning, and bioinformatics (Bessani *et al.*, 2015). Cuneiform comes with an execution environment supporting the parallel execution of programs on multi-core servers, on local clusters running distributed computation frameworks like Hadoop, HTCondor, or distributed Erlang which may be hosted on cloud infrastructures like Amazon's EC2 (Bux *et al.*, 2015). Performance evaluations on systems with up to several hundreds of nodes showed the scalability of Cuneiform workflows in a number of use cases (Bux *et al.*, 2017).

In this paper, we define Cuneiform's structural operational semantics, i.e., a small-step operational semantics (Hennessy, 1990; Winskel, 1993; Harper, 2016) together with a simple type system. Defining a language's semantics has several advantages. First, it eases language implementation because, otherwise, the language's desired behavior would have to be re-enacted from a reference implementation or extracted from an informal description. Furthermore, it permits reasoning about language properties such as the assertion that evaluation is total, consistent, and safe, as well as reasoning about when termination can be guaranteed. Last, the correctness of compilers transforming programs from or to other workflow languages with defined semantics can be verified.

Our motivation to provide and use a formal semantics for Cuneiform also arose from our own bad experiences. Previously, the behavior of Cuneiform programs was defined only informally which caused a previous Java-based interpreter (of a code size of 20 kLOC) to be difficult to maintain (Brandt *et al.*, 2015). Although this approach worked satisfactorily for basic language features, it made extending the language core to support more complex features, e.g., general recursion, difficult, and error prone. Faced with this situation, we decided to re-implement the interpreter based on a formal semantics and a type system, which made the code cleaner, easier to understand and maintain, and more concise. The resulting Java-based implementation (10 kLOC) had only roughly half the code volume while offering a richer functionality. Eventually, we also implemented the interpreter a third time in Erlang (Armstrong *et al.*, 1996) which had the additional advantage that the reduction rules can be directly expressed owing to Erlang being a functional programming language with pattern matching. Using Erlang led to a further reduction of code volume by a factor of 3 (3.6 kLOC). Thus, defining Cuneiform's semantics had an apparent positive effect on code volume and enabled important features which previously were either infeasible or had brittle implementations.

Cuneiform is minimal in the sense that it provides only those language features that are absolutely necessary regarding its specialization to organizing scientific data analysis. It focuses on integrating external operators written in a variety of different languages and the parallel execution of these operators. Accordingly, it lacks a number of features common in general purpose programming languages, such as support for efficient arithmetic operations, a more involved type system, control structures like continuations or exception handling, or encapsulation. Instead, it offers features such as automatic parallelization and direct embedding of code in external languages. Its approach differs from current database-inspired distributed systems in that Cuneiform has a strict black-box operator model and a black-box data model. This means, it neither has information about the algebraic properties of operators to re-order them nor has it a generic data model that would allow for implicit partitioning of data sets. In this, it differs from some common declarative data-flow languages like Pig Latin (Olston *et al.*, 2008) or HiveQL (Thusoo *et al.*, 2009). However, the black-box approach allows the integration of arbitrary existing tools and libraries and the processing of arbitrary data formats. The flexibility resulting from being able to reuse any library and to process any data format is a necessary condition for many scientific data analysis use cases.

The main idea behind Cuneiform's semantics is the repeated evaluation of a program term which may contain external operations. When an external operation is encountered, it is scheduled to an execution environment. Evaluation continues as far as possible to be able to simultaneously schedule as many external operations as possible. The evaluation process starts with a program term which is handed to the Cuneiform interpreter connected to a (possibly distributed) execution environment. The interpreter sends external operations that are ready to execute to the execution environment and receives operation results in return. Communication between the interpreter and the execution environment is asynchronous. External operations are independently executed and, each time an external operation finishes, the result is sent back to the interpreter which re-evaluates the program term typically generating new ready external operations. This process continues until the program term has been evaluated to a value. Then, the interpreter terminates and outputs this result value. If we expect a deterministic evaluation result given the non-determinism in the order of external operation execution, we have to make a number of assumptions about external operators; in particular, all operators are assumed (i) to terminate, because only then the workflow as a whole can terminate, (ii) to be deterministic, because intermediate results are memoized and reused wherever possible, and (iii) to be independent, i.e., external operations without explicit data dependencies never influence one another. Many scientific tools and libraries naturally fulfill these properties. However, they are only assumed but not enforced by Cuneiform.

This presentation of Cuneiform's semantics sticks closely to a simply typed lambda calculus. One major benefit from adopting a lambda calculus is the ability to express (unbounded) iteration as a recursive function call. This is especially useful in research areas like machine learning where, often, an initial model is iteratively improved until a target function has reached an optimum. With general recursion, Cuneiform is not limited to applications in which a static dependency graph is known *a priori*, which is a practical limitation of many existing workflow languages like DAGMan (Deelman *et al.*, 2006; Kalayci *et al.*, 2010) or Snakemake (Köster & Rahmann, 2012). Furthermore, the relative proximity of Cuneiform to the simply typed lambda calculus facilitates the reproduction of important properties like the theorems of Church and Rosser (1936) or the language's safety (Pierce, 2002). While we do not present proofs for these properties, here we rely on the assertion that the analogs of those results are in fact valid for Cuneiform. Section 7 shows how random test generation can be used to informally assert these properties.

The scope of this paper is the Cuneiform interpreter, i.e., the component that performs evaluation, on an abstract level. A description of the concrete syntax (Brandt *et al.*, 2015) and a distributed execution environment based on Hadoop (Bux *et al.*, 2015; Bux *et al.*, 2017) have been presented in previous publications.

The remainder of this paper is structured as follows: Section 2 shows several example workflows to demonstrate Cuneiform's usability and expressiveness. We give a more detailed overview over the goals, trade-offs, assumptions, and design decisions for Cuneiform in Section 3. After introducing the notation used throughout this paper in Sections 4 and 5 introduces Cuneiform's abstract syntax, static semantics, and type system. Section 6 describes the evaluation rules in terms of a structural operational semantics. Section 7 briefly introduces a reference implementation written in Erlang which exemplifies how the rules given in this paper can be transcribed to a general purpose programming language. Section 8 reviews related work and languages with similar design goals as Cuneiform. We conclude the paper in Section 9.

2 Examples

In this section, we discuss several examples with the goal to demonstrate the features of Cuneiform. In essence, a Cuneiform program consists of function (task) definitions and function calls. Herein, a function body can be defined in Cuneiform or in any of the supported external programming languages. For example, for defining a function that calls the command line tool `gzip` to decompress a file, we would define a function in Bash that consumes a gzipped file and outputs its decompressed version.

```
deftask gunzip( out( File ) : gz( File ) ) in Bash *{
  out=unzipped_${gz%.gz}
  gzip -c -d $gz > $out
}*
```

This snippet defines the function `gunzip` producing one output named `out` and consuming one input argument named `gz`. Both are flagged to be of type `File` which prompts the (possibly distributed) execution environment to stage-in the input file stored in the argument `gz` and, after function execution, to stage-out the output file stored in the variable `out`. While we declared `Bash` to be the language for the body of this function, any high-level programming language can be driven this way, e.g., Python, R, or Perl. Thus, it is straightforward to integrate a given tool or library with an API in one of the supported languages. The first line of the function body defines the string content of the variable out. We define the output filename to be the prefix of the original filename, thereby omitting the ".gz" suffix. In addition, we prefix the output filename with the string "unzipped_" to avoid a name clash with the original filename in case it did not have a ".gz" suffix. With the output filename defined, we can start the gzip command. The -c flag tells gzip to write to the standard output while the -d flag instructs it to decompress. The third parameter to gzip is the input file stored in the variable gz piping the output to the previously specified file stored in the variable out.

We apply the function defined above by binding the function's input argument gz to a gzipped file or a list of gzipped files.

```
gunzip( gz: "archive1.gz" "archive2.gz" "archive3.gz" );
```

In the above snippet, we bind the argument gz to a three-element list of gzipped files. The gunzip function does not consume the list as a whole. By default, the function is implicitly mapped over the elements of the input file list, generating three independent applications of gunzip which are scheduled and executed by the Cuneiform execution environment.

Use cases that involve simulation experiments often incorporate sweeps over the ranges of several arguments. For example, suppose we want to integrate a simulation library having an API in Perl. It consumes a set of observations stored in a CSV file and, given a temperature, a pH value, and a water activity, performs a simulation which produces a prediction for these arguments.

```
deftask sim( predict : observ( File ) ph temp wa ) in Perl *{
  predict = sim( $observ, $ph, $temp, $wa );
}*

observ = "observ.csv";
ph     = 4 5 6 7 8 9 10;
temp   = 16 18 20 22 24;
wa     = "0.0" "0.5" "1.0";

sim( observ: observ, ph: ph, temp: temp, wa: wa );
```

The above code snippet calls the sim function for all combinations of seven pH values, five temperature values, and three water activity values. By default, Cuneiform takes the Cartesian product of all input arguments and instantiates a call to sim for each distinct combination, i.e., the sim function is called 105 times in the above example. In a similar way, bootstrapping (Efron & Tibshirani, 1994; Manly, 2006) or *n*-fold cross validation (Bishop, 2006; Haykin *et al.*, 2009; Duda *et al.*, 2012) can be expressed.

Often it is not enough to map a task to a number of data partitions. Many algorithms, based on, e.g., gradient descent or expectation maximization, iteratively improve on an initial solution terminating the iteration when a convergence criterion is met. This iteration can be expressed in Cuneiform through recursion.

```
deftask find_clusters( cls( File ) : data( File ) state( File ) ) {

  clusters = cluster( data: data, state: state );
  state1   = reevaluate( cls: clusters );

  cls = if has_converged( old: state, new: state1 )
        then
          clusters
        else
          find_clusters( data: data, state: state1 )
        end;
}
```

The above snippet implements a *k*-means algorithm by (i) associating all data points with a cluster center using the `cluster` function and (ii) re-evaluating the cluster centers using the `reevaluate` function. The function `has_converged` checks whether a convergence criterion is met and, if so, returns the current cluster centers. Otherwise, the task calls itself with an updated state argument.[1]

In Appendix A, we demonstrate a more complex workflow from the field of bioinformatics implementing a ChIP-Seq analysis (Myers *et al.*, 2013). In a ChIP-Seq experiment, the starting point is a set of sequenced DNA reads that have been previously selected via immunoprecipitation. By detecting peaks in the sequence coverage relative to a baseline coverage, it is possible to identify the active genes in a cell under a given test condition. The workflow demonstrates how sequence alignment to a reference genome, peak detection, peak annotation, and quality control can be expressed in Cuneiform.[2]

In summary, Cuneiform enables direct integration of external tools and libraries by allowing function definitions to use external programming languages, it implicitly applies appropriate second-order functions to iterate over lists, and it allows unbounded iteration through general recursion. The implicit application of second-order functions to process lists is a way to make programs type check that otherwise would not. It also relieves the programmer from using second-order functions explicitly when their expedience is clear from the context. Implicit list processing is also found in Taverna's implicit iteration (Oinn *et al.*, 2006) or in Matlab where most arithmetic operations are defined for matrices and a number is treated as a single-element matrix.

3 The Cuneiform approach

Cuneiform is an organizational language for distributed data analysis. Its goal is the easy integration of external libraries and tools by allowing to write light-weight wrappers around these tools and to parallelize them automatically. By providing

[1] A full implementation of the *k*-means workflow example is available under https://github.com/joergen7/kmeans.

[2] Detailed information about the ChIP-Seq workflow example is available at http://cuneiform-lang.org/examples/2016/04/29/chip-seq/.

a generic black-box data model, Cuneiform can process data encoded in any serializable data format. As such, Cuneiform is a hybrid of a functional programming language, because, at its heart, it is a lambda calculus with black boxes, a database query language, because its purpose is to coordinate parallel computation on data sets potentially exceeding the disk size of a single computer, and a scientific workflow language, because it integrates foreign tools and libraries producing and consuming files in various domain-specific formats.

External black-box operations often involve complex computations that would be expensive to repeat if a workflow is run more than once or if different workflows have external operations in common. Thus, a Cuneiform execution environment should memoize all external operations and reuse computation results where possible. While exhaustive memoization may be problematic in situations where many cheap computations are memoized, it becomes invaluable when the number of computations is moderate but computations themselves are costly (e.g., take hours or days to complete) as is often the case in data analysis application areas. For the same reason, it is undesirable to execute any external operation that does not, in some way, contribute to the final result of a workflow, i.e., if there is no explicit data dependency between an external operation and the result term we want to dispose of this external operation early. In functional programming languages, this can be achieved by picking a call-by-name evaluation strategy (Michaelson, 2011). However, since there is no way of substituting an unevaluated term into the body of a black-box operator, Cuneiform resorts to a mixture of a call-by-name and a call-by-value evaluation strategy. Herein, it gives precedence to call-by-name evaluation wherever possible to discard non-contributing terms early on but it is uses a call-by-value strategy when it needs to evaluate the arguments of external black-box operators.

Cuneiform is intended for the analysis of large data sets. Thus, Cuneiform introduces "files" as a separate base data type for a finite-size addressable black-box data object. As far as the Cuneiform semantics are concerned, both files and strings behave identically. However, external operators need a hint whether a value is just a string or whether it is an address referencing a data object which needs to be made available prior to execution or published prior to communicating this address to downstream operators. Herein, the Cuneiform interpreter does not need to care whether access to a data object is implemented as a database access, as an API call to a remote storage service, as the creation of a symbolic link in the local file system, or as sending and receiving from a Unix pipe, a socket, or a distributed message queue. All the interpreter needs to know is whether a string stands just for itself or whether it is an address referencing a data object and treat these cases consistently.

Language features for managing concurrency, communication, or synchronization are deliberately missing from Cuneiform. Details about the fact that computation is eventually performed in a potentially large distributed system, organized as a composition of web-services providing distributed storage and computation in the presence of failures, should not leak into the organizational language itself. While no explicit language features for managing concurrency are provided, concurrency is achieved by treating external black-box operators like pure functions: If external operations do not influence one another (i.e., if they are pure), then it is possible

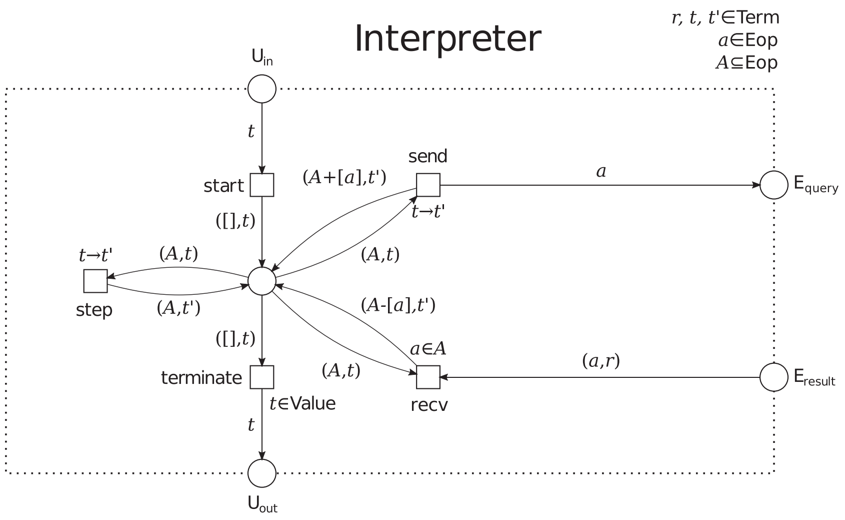

Fig. 1. Interpreter with user interface (U_{in} and $\mathsf{U}_{\mathsf{out}}$) and interface to execution environment ($\mathsf{E}_{\mathsf{query}}$ and $\mathsf{E}_{\mathsf{result}}$). Interpretation starts when a Cuneiform term t appears on place U_{in}. This term does not contain any futures yet so the multi-set of external operations A is initialized empty. When the *step* transition fires, there a reduction rule is applied to the term t reducing it to t'. When an external operation is encountered, the *send* transition fires, replacing the external operation in t with a future resulting in t' and adding the external operation a to the multi-set A and also sending a to the execution environment via the place $\mathsf{E}_{\mathsf{query}}$. When an external operation finishes, it appears together with its result term on the place $\mathsf{E}_{\mathsf{result}}$. Now, the *recv* transition can fire. It substitutes the result term r for any appearance of the future embodying the external operation a in the term t, and also a is discarded from the multi-set of external operations A. Evaluation continues until the term t is a value. In this event, it is returned to the user.

to evaluate them independently. Herein, reading or writing to disk and even communicating over the network is tolerable as long as operators are deterministic and independent. Moreover, the Church–Rosser property allows us to substitute the evaluation results in the order they arrive and continue evaluation as far as possible as early as possible. Note that this makes Cuneiform inherently non-deterministic since the order in which external operations are scheduled and results arrive is up to the execution environment (see Figure 1).

By independently running black box-operations, *task* parallelism is achieved. *Data* parallelism is a special case of task parallelism which is achieved by applying second-order functions to lists of files representing data partitions. Execution environments for Cuneiform also exploiting *pipeline* parallelism are feasible but have not been conceived to date.

Following this approach, we introduce the structural operational semantics of Cuneiform as a simply typed lambda calculus with black-box functions.

4 Preliminaries

A Cuneiform program is provided by the user in the form of a sequence of characters constituting the concrete program. When execution finishes, the interpreter generates

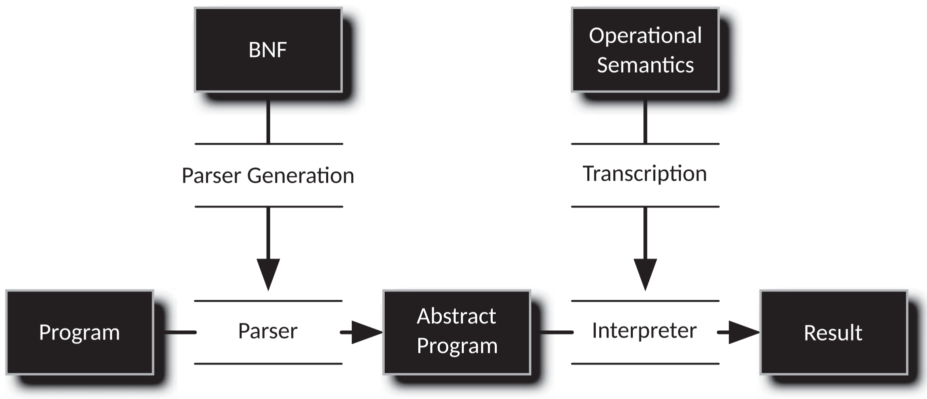

Fig. 2. Specification of a Cuneiform interpreter. A Cuneiform program is first parsed resulting in an abstract program. This abstract program is then evaluated by the interpreter producing the result value. Both the parser and the interpreter have formal specifications. While the parser can be generated automatically from the specification in BNF, the specification of the interpreter has to be manually transcribed.

the output in the form of a Cuneiform *value*. The overarching transformation process is commonly separated in a parsing step and an interpretation step (see Figure 2). First, the concrete program is parsed resulting in an abstract program having the form of a Cuneiform *term*. For Cuneiform as for many programming languages, the concrete syntax is specified in Backus–Naur Form (BNF) and a parser generator can be used to automatically generate the parser from the syntax definition in Backus–Naur Form. Herein, only part of a program string is actually to be parsed as Cuneiform code. The body of any external operator definition in, e.g., R or Python, is left unparsed. The lexer recognizes the beginning and end of a foreign operator body via a matching pair of mickeymouse-eared curly braces: `*{...}*`. where it is assumed that no such character combination is ever used inside a foreign function body. When the parser encounters external code, it stores the code snippet as is, i.e., still in its textual form, inside a foreign abstraction term. Only when the execution environment schedules the external operation to run in an actual R or Python instance, the foreign code is actually parsed (and run).

The discussion of Cuneiform's semantics starts in Section 5 by giving the abstract syntax which specifies the form that a Cuneiform term can assume and which terms are also values. In addition, this section provides the typing rules for terms. If a program adheres to the abstract syntax, it is well-formed. If it, additionally, adheres to the typing rules, it is well typed. We demand that the interpreter only accepts programs that are well typed. In Section 6, we present the rules for evaluating terms in the form of a structural operational semantics. That is, evaluation is defined as a series of small evaluation steps that are repeatedly applied until a Cuneiform term results, that cannot be evaluated any further.

Commonly, in a safe programming language, the term that results when the interpreter halts is guaranteed to be a value. In Cuneiform, there is another reason for the interpreter to suspend evaluation: Any unfinished external operation appears as a future inside the Cuneiform term under evaluation. A future is neither a value nor can it be further evaluated. Instead, it is substituted for its result value when the corresponding external operation finishes. Thus, whenever a term contains external operations, the interpreter eventually suspends evaluation while the result term is not a value. In this event, the interpreter waits until an external operation finishes. Upon substitution of a future for its result value, interpretation is resumed until the interpreter is suspended again. Finally, the interpreter evaluates the term to a value and the interpreter terminates.[3] So for Cuneiform to be safe, we have to assume that all futures are eventually substituted (i.e., we never lose external operations and they always terminate) *and* that the return values of external operations match their declared type.

In the two upcoming sections, the abstract syntax, typing rules, and evaluation rules for Cuneiform are introduced. Herein, each syntactic category, is introduced in turn, by first stating, how the set of terms and values is to be extended. For example, for introducing the syntactic category of a string literal, we write $t ::= \textbf{str}\ s$ to say that a term t is defined to be the symbol **str** followed by an instance of the meta-variable s. Together with a term definition, we also give a value definition of the form $v ::= \textbf{str}\ s$ to say that a string literal is not only a term but also a value. While we explicitly define what form the meta-variables t and v can assume, the definition of the abstract syntax contains the following other syntactic categories: We write s for some string "…", i for some positive natural number $i \in \{1, 2, \ldots\}$, and k for some future identifier $k \in K$, where K is some set from which future identifiers can be chosen. In practice, this can be a unique natural number or some hash value represented by a binary. The meta-variable x denotes a name.

When introducing the abstract syntax of types, we distinguish between types in general T and types that can be consumed or produced by external operators U which we call ground types. Whenever we introduce types, e.g., by writing $T ::= \mathsf{Bool}$, to say that the meta-variable T can be the type Bool, we also state whether we introduce it as a ground type by writing $U ::= \mathsf{Bool}$ to say that a Boolean is not only a type but also a ground type.

Typing rules are given as a three-element relation written in the form $\Gamma \vdash t : T$ meaning that in the context Γ, the term t is element of the set uniquely represented by the type T. The context Γ itself is a two-element relation, associating some name x with some type T. Herein, we use the fact that a two-element relation can be represented as a set of pairs to construct it. We write $\Gamma = \emptyset$ to express that the context relation Γ never holds. To extend Γ so that it additionally holds for the pair $x_1 : T_1$, we write $\Gamma, x_1 : T_1$. To extend Γ with the set of pairs $x_i : T_i^{i \in 1..n}$, we write $\Gamma, (x_i : T_i^{i \in 1..n})$. We write $x_1 : T_1 \in \Gamma$ to say that the context relation Γ holds for a name x_1 of type T_1. Function types play a special role Cuneiform. A function type is written in the form $\left(x_i : T_i^{i \in 1..n}\right) \rightarrow^{\tau} T_{\text{ret}}$ meaning the type of a function taking n arguments with the names x_1 to x_n having the types T_1 to T_n and returning a value

[3] A Cuneiform term may diverge so termination is, in general, not guaranteed even if the language is safe.

of type T_{ret}. Herein, τ is either "ntv," if it is a native function, or "frn" if it is an external operator, i.e., a foreign function.

After the abstract syntax and typing rules have been introduced, we give the evaluation rules in Section 6. We write down the small-step evaluation rules in the following way: To say that a term t can take an evaluation step to t', we write $t \longrightarrow t'$. To perform substitution, we write $[x \mapsto t_1]t_2$ to say that all occurrences of the variable with the name x in the term t_2 should be substituted for the term t_1 up to α-renaming. Thus, substitution is suspended for shadowed variables and is capture avoiding.

5 Abstract syntax, static semantics, and types

This section introduces the abstract syntax and typing rules of Cuneiform. When presenting the abstract syntax, we distinguish between terms t and values v. Any valid Cuneiform program has the form of a term. Values are those terms that can be the output of interpretation. They cannot be further evaluated. If a term already is a value, then it is returned to the user as it is. Otherwise, it is evaluated until a value results. In this section, we define both terms and values separately. Moreover, we give typing rules to uniquely associate any term or value with its corresponding type, given that the types of its sub-terms are known. Since external operators are an integral part of the Cuneiform semantics, they are introduced as a syntactic category in their own right. The arguments consumed by external operators as well as the results they produce are always values, e.g., Booleans or string lists. However, while native functions and the external operators themselves are also values, they cannot be consumed (or produced) by external operations. Thus, we have to distinguish between types in general T which include function types and those types that can be arguments or results of external operators U which we call ground types. Both types and ground types are separately defined.

We start the discussion of Cuneiform's abstract syntax by giving an overview over all syntactic categories:

$$
\begin{array}{lll}
t ::= & & \\
 & \mathbf{str}\ s & \mathbf{fix}\ t \\
 & \mathbf{file}\ s & \mathbf{nil}[T] \\
 & x & \mathbf{cons}[T]\ t\ t \\
 & \lambda_{\text{ntv}}\ (x_i : T_i^{\,i\in 1..n})\ .\ t & \mathbf{isnil}[T]\ t \\
 & \lambda_{\text{frn}}\ (x_i : U_i^{\,i\in 1..n}) \rightarrow^{\text{frn}} U\ \mathbf{in}\ l\ .\ s & \mathbf{zip}[x_i^{\,i\in 1..n} | T_{\text{ret}}]\ t \\
 & t\ \left(x_i = t_i^{\,i\in 1..n}\right) & \{t_i^{\,i\in 1..n}\} \\
 & \mathbf{fut}[U]\ k & \pi_i\ t \\
 & \mathbf{true} & \Pi_i\ t \\
 & \mathbf{false} & \mathbf{let}\ x : T = t\ \mathbf{in}\ t \\
 & \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t &
\end{array}
$$

In the following, we introduce each of these syntactic categories in turn. We start with the base types which are strings and files. Then we introduce the lambda calculus with black boxes and continue by introducing Booleans and conditions, general recursion, and, eventually, compound data types like lists and tuples and their respective constructors and accessors.

5.1 Files and string literals

String literals and files are important basic data types in Cuneiform. For example, the arguments of external libraries can be encoded as a string data type. As an example, consider the significance level of a p-value which can be encoded as **str** “0.99.” Similarly, if a tool consumes a file, the filename can be encoded as a file data type. As an example, consider a partition of a DNA sample which may be stored in a file encoded as **file** “sample01.fastq.”

$$\begin{array}{llll} t ::= & & v ::= & \\ & \textbf{str}\ s & & \textbf{str}\ s \\ & \textbf{file}\ s & & \textbf{file}\ s \end{array} \tag{1}$$

While not supported in the currently available Cuneiform implementations, it is possible to include integers or floating point numbers.[4] Care must be taken to take into account differences in the definition ranges of these data types among different external languages. To keep the notation concise, we do not extend the abstract syntax to this level of detail here. Strings and files are not only terms but also values, i.e., they can be the result of a Cuneiform program. Next, we introduce a string type Str and a file type File. Both types Str and File are also ground types, i.e., they can be arguments or the result of external operations.

$$\begin{array}{llll} T ::= & & U ::= & \\ & \mathsf{Str} & & \mathsf{Str} \\ & \mathsf{File} & & \mathsf{File} \end{array} \tag{2}$$

The typing rules state that any term that has the form of a string (or file) is of the type Str (or File) independent of the context Γ.

$$\Gamma \vdash \textbf{str}\ s : \mathsf{Str} \tag{3}$$

$$\Gamma \vdash \textbf{file}\ s : \mathsf{File} \tag{4}$$

5.2 Lambda calculus with black boxes

Cuneiform is presented here as a lambda calculus. However, to attribute for the language modularity described earlier, Cuneiform includes, next to native functions, the syntactic category of a foreign functions (i.e., an external black-box operator). Herein, we need to account for the fact that the arguments of a foreign function cannot be applied one argument at a time like it is done in the canonical lambda

[4] Currently, numbers are distinct in the concrete syntax but are internally represented as strings and handled accordingly.

calculus but have to be applied all at once. To be as consistent as possible between native and foreign functions, we allow both function types to bind multiple arguments at once. Together with variables and function applications, we obtain the following syntax for a lambda calculus with black boxes:

$$\begin{array}{lll} t ::= & \ldots & \qquad v ::= \ldots \\ & x & \\ & \lambda_{\text{ntv}}\ (x_i : T_i{}^{i \in 1..n})\ .\ t & \qquad \quad \lambda_{\text{ntv}}\ (x_i : T_i{}^{i \in 1..n})\ .\ t \\ & \lambda_{\text{frn}}\ (x_i : U_i{}^{i \in 1..n}) \rightarrow^{\text{frn}} U\ \textbf{in}\ l\ .\ s & \qquad \quad \lambda_{\text{frn}}\ (x_i : U_i{}^{i \in 1..n}) \rightarrow^{\text{frn}} U\ \textbf{in}\ l\ .\ s \\ & t\ \left(x_i = t_i{}^{i \in 1..n}\right) & \end{array} \tag{5}$$

Variables are represented just with their variable name x. We introduce function values for both the native and the foreign case. A function binds a (possibly empty) set of arguments x_1 to x_n having the types T_1 to T_n. The return type of a native function can be obtained by applying the typing rules presented in this section. The body of a native function is a term. In the foreign case, the return type cannot be inferred by applying the typing rules. Accordingly, the full function type (arguments and return type) has to be part of the definition of a foreign function. Last, we need to be told the body of the foreign function which is represented as an arbitrary string s as well as the information in which external programming language l it is written. Important external languages are

$$\begin{array}{ll} l ::= & \\ & \textbf{Bash} \\ & \textbf{Octave} \\ & \textbf{Perl} \\ & \textbf{Python} \\ & \textbf{R} \\ & \ldots \end{array} \tag{6}$$

The only types that need to be added are the two function types: the native function type and the foreign function type, while the foreign function type can assume only ground types for each of its argument types and the return type.

$$\begin{array}{ll} T ::= & \ldots \\ & (x_i : T_i{}^{i \in 1..n}) \rightarrow^{\text{ntv}} T \\ & (x_i : U_i{}^{i \in 1..n}) \rightarrow^{\text{frn}} U \end{array} \tag{7}$$

The function types are not added to the set of ground types since functions cannot be an argument or the result of an external operation.

We, now, give typing rules for the four syntactic categories introduced: Variables, native and foreign abstractions, and function applications. The type of a variable x is looked up in the context Γ in which the variable term is located.

$$\frac{x_1 : T_1 \in \Gamma}{\Gamma \vdash x : T_1} \tag{8}$$

The function type of a native abstraction results from the argument types and the inferred return type, while the foreign function type is part of the foreign abstraction's definition.

$$\frac{\Gamma, (x_i : T_i^{i\in 1..n}) \vdash t_2 : T_{\text{ret}}}{\Gamma \vdash \lambda_{\text{ntv}}\ (x_i : T_i^{i\in 1..n})\ .\ t_2 : (x_i : T_i^{i\in 1..n}) \rightarrow^{\text{ntv}} T_{\text{ret}}} \tag{9}$$

$$\Gamma \vdash \lambda_{\text{frn}}\ (x_i : U_i^{i\in 1..n}) \rightarrow^{\text{frn}} U_{\text{ret}}\ \mathbf{in}\ l\ .\ s : (x_i : U_i^{i\in 1..n}) \rightarrow^{\text{frn}} U_{\text{ret}} \tag{10}$$

The type of an application is the return type of its left-hand term t_f which needs to be of a function type, native or foreign.

$$\frac{\Gamma \vdash t_f : \left(y_i : T_i^{i\in 1..n}\right) \rightarrow^{\tau} T_{\text{ret}} \forall i\ \ (x_i = y_i \wedge \Gamma \vdash t_i : T_i)}{\Gamma \vdash t_f\ \ \left(x_i = t_i^{i\in 1..n}\right) : T_{\text{ret}}} \tag{11}$$

5.3 Futures

Futures are temporary placeholders for unfinished external operations. A future denotes the return type U of the applied external operator (which must be a ground type) as well as a future identifier $k \in K$ from some set of possible future identifiers. A typical choice for K would be the set of natural numbers $\mathbb{N}$. Note that two future names can be allowed to coincide only when they represent exactly the same external operation. This can be achieved by assigning increasing numbers to futures as they are encountered. We, thus, define the syntactic category of futures:

$$\begin{aligned} t ::=\ & \ldots \\ & \mathbf{fut}[U]\ k \end{aligned} \tag{12}$$

Note that a future is not a value. The typing rules state that a future is always of the ground type it denotes, so no additional types need to be introduced.

$$\Gamma \vdash \mathbf{fut}[U_1]\ k_1 : U_1 \tag{13}$$

5.4 Booleans and conditions

The only base data type in Cuneiform in addition to Str and File is the Boolean type Bool. Boolean values are the symbols **true** and **false**. We use Booleans in conditional terms which branch Cuneiform programs depending on a condition known only at run time, e.g., to express the exit condition of a recursive function. For example, the term

$$\mathbf{if\ true\ then}\ (\mathbf{str}\ \text{“foo”})\ \mathbf{else}\ (\mathbf{str}\ \text{“bar”})$$

evaluates to the string value **str** “foo.”

$$\begin{aligned} t ::=\ & \ldots & \qquad v ::=\ & \ldots \\ & \mathbf{true} & & \mathbf{true} \\ & \mathbf{false} & & \mathbf{false} \\ & \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t & & \end{aligned} \tag{14}$$

We extend the set of types for the type Bool which is also a ground type, i.e., it can be an argument or the result of an external operation.

$$\begin{array}{rl} T ::= & \ldots \\ & \mathsf{Bool} \end{array} \qquad \begin{array}{rl} U ::= & \ldots \\ & \mathsf{Bool} \end{array} \tag{15}$$

The typing rules state that the values **true** and **false** are of type Bool and that a conditional term is well typed if its condition t_1 is a Boolean and the then-term t_2 has the same type as the else-term t_3. The conditional term itself then has the type of the then- and else-terms.

$$\Gamma \vdash \textbf{true} : \mathsf{Bool} \tag{16}$$

$$\Gamma \vdash \textbf{false} : \mathsf{Bool} \tag{17}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \Gamma \vdash t_2 : T_{23} \Gamma \vdash t_3 : T_{23}}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T_{23}} \tag{18}$$

5.5 General recursion

Another important feature of Cuneiform is the possibility to define recursive functions. This is achieved via the introduction of a fixpoint operator. To attain the typability of such an operator, we do not introduce it as a derived form but as a syntactic category in its own right. For example, say we have defined a function

$$f^* := \lambda_{\text{ntv}} (\mathsf{f} : (\mathsf{x} : \mathsf{File}) \to^{\text{ntv}} \mathsf{Bool}, \mathsf{x} : \mathsf{File}) . \ldots$$

with two arguments: "f," a function that consumes a file and "x," a file. The function f^* returns a value of type Bool and its body may contain f as a free variable. We can construct a recursive function from f^* by using it in the following expression:

$$f := \textbf{fix } f^*$$

This way we construct a recursive function $f : (\mathsf{x} : \mathsf{File}) \to^{\text{ntv}} \mathsf{Bool}$. Accordingly, we extend the set of terms in the following way:

$$\begin{array}{rl} t ::= & \ldots \\ & \textbf{fix } t \end{array} \tag{19}$$

Since a fixpoint term can always be evaluated, we do not need to add fixpoint terms to the set of values. The typing rule requires the first argument of a function term t_1 to be the name of the recursive function x_f. Then, the remaining arguments and the return type of t_1 make up the type of **fix** t_1.

$$\frac{\Gamma \vdash t_1 : (x_f : (x_i : T_i^{i \in 1..n}) \to^{\text{ntv}} T_{\text{ret}}, x_i : T_i^{i \in 1..n}) \to^{\text{ntv}} T_{\text{ret}}}{\Gamma \vdash \textbf{fix } t_1 : (x_i : T_i^{i \in 1..n}) \to^{\text{ntv}} T_{\text{ret}}} \tag{20}$$

5.6 Lists

In Cuneiform, data parallelism is achieved by partitioning a data set and applying functions to lists of partitions. Thus, the list is an important compound data type.

We introduce the empty list of type T **nil**$[T]$ and the list constructor **cons**$[T]$ t t as list values. The **isnil**$[T]$ t predicate returns **true** if a list is empty. For example, the list value **nil**[File] denotes the empty list of files, while the list value

$$\textbf{cons}[\mathsf{File}]\ (\textbf{file}\ \text{"foo1.txt"})\ (\textbf{cons}[\mathsf{File}]\ (\textbf{file}\ \text{"foo2.txt"})\ (\textbf{nil}[\mathsf{File}]))$$

denotes the two-element list containing the files “foo1.txt” and “foo2.txt.” Thus, the set of terms is extended to contain the empty list, the list constructor, and the isnil predicate. Of these, the empty list is a value as well as the list constructor whose head and tail are values.

$$\begin{array}{rlcrl} t ::= & \ldots & \quad & v ::= & \ldots \\ & \textbf{nil}[T] & & & \textbf{nil}[T] \\ & \textbf{cons}[T]\ t\ t & & & \textbf{cons}[T]\ v\ v \\ & \textbf{isnil}[T]\ t & & & \end{array} \tag{21}$$

The list type is added to the set of types which is also a ground type. That is, lists can be the arguments or return values of external operators.

$$\begin{array}{rlcrl} T ::= & \ldots & \quad & U ::= & \ldots \\ & \mathsf{List}\ T & & & \mathsf{List}\ U \end{array} \tag{22}$$

We give typing rules for the empty list, the list constructor, and the isnil operator: The empty list is of the list type it specifies.

$$\Gamma \vdash \textbf{nil}[T_1] : \mathsf{List}\ T_1 \tag{23}$$

If the list constructor’s head is of the specified type and the tail is of the specified list type, the constructor is also of this list type.

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : \mathsf{List}\ T_1}{\Gamma \vdash \textbf{cons}[T_1]\ t_1\ t_2 : \mathsf{List}\ T_1} \tag{24}$$

The isnil operator is of type Boolean if its argument is of the specified list type.

$$\frac{\Gamma \vdash t_1 : \mathsf{List}\ T_1}{\Gamma \vdash \textbf{isnil}[T_1]\ t_1 : \mathsf{Bool}} \tag{25}$$

5.7 Accessing lists using map, zip, or the Cartesian product

In the previous section, we have introduced constructors for lists (**nil**$[T]$ and **cons**$[T]$ t t), but we have refrained from introducing list accessors like head or tail. The reason for this restraint is that these accessors can be defined only as partial functions over lists since they are undefined for the empty list **nil**$[T]$. There are several ways to deal with this partiality, e.g., to emit an error at run time. However, since many data-intensive workloads run for hours or days, it is impractical to detect an undefined list access only at run time when the program may already be running for several hours. Cuneiform avoids this problem by constraining the user to either apply an external operator that consumes the list as a whole or to access the list via its zip operator, which implements list comprehensions. The zip operator closely resembles Common Lisp’s `mapcar` or Racket’s `for/list` form. However, the interpreter does not suspend evaluation when an external operation is encountered. Instead, all list elements are processed right away to detect and start as many external operations as possible.

For example, say we have two equally long lists of Booleans and we want to perform pairwise Boolean conjunction over both lists. First, we define $f^* : (a : \mathsf{Bool}, b : \mathsf{Bool}) \rightarrow^{\mathrm{ntv}} \mathsf{Bool}$ consuming two arguments with the names a and b, both of which are of type Bool.

$$f^* := \lambda_{\mathrm{ntv}}\ (\mathsf{a} : \mathsf{Bool}, \mathsf{b} : \mathsf{Bool})\ .\ (\textbf{if}\ \mathsf{a}\ \textbf{then}\ \mathsf{b}\ \textbf{else}\ \textbf{false})$$

From the function f^*, we create a new function f also consuming two arguments a and b but of type $\mathsf{List\ Bool}$ by applying the zip operator. The new function returns a term of type $\mathsf{List\ Bool}$.

$$f := \textbf{zip}[\mathsf{a}, \mathsf{b}|\mathsf{Bool}]\ f^*$$

If the above term is applied to two Boolean lists of equal length, the original function f^* is applied to each pair of elements from both lists, generating a new list of the return type of f^*. As an example, we apply the function f to two lists of size two:

$$l_1 := \textbf{cons}[\mathsf{Bool}]\ \textbf{true}\ (\textbf{cons}[\mathsf{Bool}]\ \textbf{false}\ \textbf{nil}[\mathsf{Bool}])$$

$$l_2 := \textbf{cons}[\mathsf{Bool}]\ \textbf{true}\ (\textbf{cons}[\mathsf{Bool}]\ \textbf{true}\ \textbf{nil}[\mathsf{Bool}])$$

$$f\ \ (\mathsf{a} = l_1, \mathsf{b} = l_2)$$

This term evaluates to the two-element list $\textbf{cons}[\mathsf{Bool}]\ \textbf{true}\ (\textbf{cons}[\mathsf{Bool}]\ \textbf{false}\ \textbf{nil}[\mathsf{Bool}])$.

In the following, we extend the term definition with the zip operator. Since its evaluation rules are explained only in the context of function application, we need to include the zip operator in the set of values to account for the possibility that it is never applied.

$$\begin{array}{rll} t ::= & \ldots & \\ & \textbf{zip}[x_i^{i\in 1..n}|T]\ t & \end{array} \qquad \begin{array}{rll} v ::= & \ldots & \\ & \textbf{zip}[x_i^{i\in 1..n}|T]\ t & \end{array} \tag{26}$$

The typing rules for the zip operator state that the zip operator alters the type of its original function to return $\mathsf{List}\ T_{\mathrm{ret}}$ if the original function returns T_{ret} and that it alters the type T_i of each argument of the original function to be $\mathsf{List}\ T_i$ for each argument x_i that appears in the argument name list of the operator. Herein, the zip operator has to specify at least one argument name.

$$\frac{m \geqslant 1 \qquad \forall j\ \exists i\ \ y_j = x_i \qquad \Gamma \vdash t_f : \left(x_i : T_i^{i\in 1..n}\right) \rightarrow^{\tau} T_{\mathrm{ret}}}{\Gamma \vdash \textbf{zip}[y_j^{j\in 1..m}|T_{\mathrm{ret}}]\ t_f : \left(x_i : \begin{cases} \mathsf{List}\ T_i & , \exists j\ \ y_j = x_i \\ T_i & , \text{else} \end{cases}^{i\in 1..n}\right) \rightarrow^{\tau} \mathsf{List}\ T_{\mathrm{ret}}} \tag{27}$$

Note that in the case of a single argument, the zip operator degenerates to a map. Furthermore, by chaining map operators, we can achieve the effect of a Cartesian product. With zip, we have, thus, introduced a powerful higher order operator that can be used in a variety of ways to process lists. Of course, it would also be possible to introduce zip as a derived form instead of giving it the rank of a distinct syntactic category. But not only is the zip operator an important feature in Cuneiform, it is also a way to introduce some higher order constructs without making use of the fixpoint operator (introduced in Section 5.5). In the absence of general recursion, the assertion of progress (which is part of the safety argument for Cuneiform) also is an assertion of termination. That is, Cuneiform programs not using the fixpoint operator are guaranteed to terminate (Pierce, 2002) under the assumption that all external operations terminate. By introducing the zip operator this way, termination can be argued for all programs that use the zip operator but not the fixpoint operator.

5.8 Tuples

External operators can have multiple outputs. To account for this, Cuneiform introduces tuples and the projection operation allowing the extraction of the ith tuple element. So an external operator producing n outputs returns an n-ary tuple in Cuneiform. Since external operators are often mapped over lists of data partitions, a very common situation is that we have lists of tuples from which we need to obtain a list comprising only the ith element of each tuple. This is why we also introduce a projection through lists (which we call a list projection). For example, say that the application of an external operator that yields two output files has been mapped over a list of input files resulting in the following output list of type $\mathsf{List}\ \{\mathsf{File},\mathsf{File}\}$:

$$\begin{aligned}\mathbf{cons}[\{\mathsf{File},\mathsf{File}\}]\ \ &\{(\mathbf{file}\ \text{"foo1.txt"}),(\mathbf{file}\ \text{"bar1.txt"})\}\\ &(\mathbf{cons}[\{\mathsf{File},\mathsf{File}\}]\ \{(\mathbf{file}\ \text{"foo2.txt"}),(\mathbf{file}\ \text{"bar2.txt"})\}\\ &\qquad\qquad\mathbf{nil}[\{\mathsf{File},\mathsf{File}\}])\end{aligned}$$

If we are interested in a list containing only the files of the first operator output, we can use the list projection Π_1 $(\mathbf{cons}[\{\mathsf{File},\mathsf{File}\}]\ \dots)$ to obtain a list of type $\mathsf{List}\ \mathsf{File}$. The set of terms is, thus, extended for the syntactic category of tuples, projections, and list projections where only tuples are values.

$$\begin{array}{rll} t ::= & \dots & \qquad v ::= \ \dots \\ & \{t_i^{i\in 1..n}\} & \qquad\qquad \{v_i^{i\in 1..n}\} \\ & \pi_i\ t & \\ & \Pi_i\ t & \end{array} \tag{28}$$

We add the tuple type to the set of types. Herein, a tuple type containing only ground types is itself a ground type. This way tuples can be the input or output of external operators.

$$\begin{array}{rll} T ::= & \dots & \qquad U ::= \ \dots \\ & \{T_i^{i\in 1..n}\} & \qquad\qquad \{U_i^{i\in 1..n}\} \end{array} \tag{29}$$

The typing rules state that the tuple type enumerates the types of all its elements.

$$\frac{\forall i\ \ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i^{i\in 1..n}\} : \{T_i^{i\in 1..n}\}} \tag{30}$$

Furthermore, if a projection is applied to a tuple, this projection has the type of the corresponding tuple element.

$$\frac{\Gamma \vdash t_1 : \{T_i^{i \in 1..n}\}}{\Gamma \vdash \pi_j\, t_1 : T_j} \tag{31}$$

Accordingly, if a list projection is applied to a list of tuples, this list projection has the list type of the corresponding tuple element.

$$\frac{\Gamma \vdash t_1 : \mathsf{List}\,\{T_i^{i \in 1..n}\}}{\Gamma \vdash \Pi_j\, t_1 : \mathsf{List}\ T_j} \tag{32}$$

5.9 Let bindings

The last syntactic category we need to define are let bindings.

$$\begin{aligned} t ::= \quad & \dots \\ & \textbf{let}\ x : T = t\ \textbf{in}\ t \end{aligned} \tag{33}$$

and give their typing rule:

$$\frac{\Gamma \vdash t_1 : T_1 \Gamma, (x : T_1) \vdash t_2 : T_2}{\Gamma \vdash \textbf{let}\ x : T_1 = t_1\ \textbf{in}\ t_2 : T_2} \tag{34}$$

6 Evaluation

Having defined the abstract syntax and typing rules, we can define the evaluation rules for Cuneiform. We give a formal definition of term evaluation in the form of a structural operational semantics (Plotkin, 1981). In contrast to the alternative natural semantics (Kahn, 1987) (big-step semantics) in which evaluation is defined as a single recursive transformation, structural operational semantics is defined as the repeated application of a standard reduction relation. This relation defines a reduction step that transforms a term to a (possibly) simpler equivalent term. To express that the standard reduction relation holds for a pair of terms t and t', we write $t \longrightarrow t'$. We repeatedly apply the standard reduction relation until none of the reduction rules can be applied. The term we obtain by applying the standard reduction relation n times to an initial term t_1 is the evaluation result t^*.

$$t_1 \longrightarrow t_2 \longrightarrow \dots \longrightarrow t_{n-1} \longrightarrow t^*$$

The safety argument for Cuneiform's typing rules asserts that the term t^* for which no evaluation rules apply, either is a value or still contains futures acting as placeholders for external operations to become available later in time. If t^* is a value, it is returned to the user. Otherwise, evaluation is suspended until the result of some external operation becomes available, in which event the corresponding future is substituted for its result and evaluation is resumed.

This section introduces the standard reduction relation, i.e., the small-step evaluation rules for well-typed terms by extending a notion of reduction with an evaluation strategy. First, we give the evaluation rules for a lambda calculus with black boxes and show how external operations are represented with futures. Then, we introduce conditionals. After giving the evaluation rules for the fixpoint operator, we state how list and tuple accessors are evaluated. Last, let bindings are discussed. These rules make up the notion of reduction for Cuneiform.

Cuneiform's evaluation strategy is a mixture of call-by-name and a non-deterministic variant of call-by-value. Many general purpose programming languages fix the order in which sub-terms are evaluated to ensure that side effects of sub-terms are not only guaranteed to occur but also that they occur in a deterministic order. This ordering is important since operations in such languages usually dependent on the order of side effects. In contrast, Cuneiform's external operations must be independent (i.e., they must not influence the result of other external operations). Since operators are black boxes, independence can only be assumed but not enforced. Independence allows us to discard unneeded sub-terms and to reorder, or restart sub-term evaluation where most programming languages need to take great care for side effects to occur in a foreseeable order and at most once.

Moreover, independence allows us to relax the evaluation strategy mixing the call-by-name strategy (for early discarding of unnecessary computations) and a non-deterministic variant of the call-by-value strategy (for evaluating the arguments of external operators in parallel). This mixed evaluation strategy is defined by the congruence rules we give in Section 6.8 which extend the previously given notion of reduction to form the standard reduction relation.

6.1 Beta reduction

Here, we describe how the beta reduction of the canonical lambda calculus translates to Cuneiform's lambda calculus with black boxes. First, a native function application not binding any arguments trivially evaluates to the unaltered function body.

$$(\lambda_{\text{ntv}}\ ()\ .\ t_b)\ ()\longrightarrow t_b \tag{35}$$

A native function application binding one or more arguments substitutes the first bound argument into the function body, thereby removing the argument from the function term as well as the argument binding on the right-hand side of the application.

$$\begin{aligned} &(\lambda_{\text{ntv}}\ (x_1 : T_1, x_i : T_i{}^{x_i\in 2..n})\ .\ t_b)\ \left(x_1 = t_{21}, x_i = t_{2i}{}^{i\in 2..n}\right)\\ \longrightarrow\ &(\lambda_{\text{ntv}}\ (x_i : T_i{}^{x_i\in 2..n})\ .\ ([x_1 \mapsto t_{21}]t_b))\ \left(x_i = t_{2i}{}^{i\in 2..n}\right)\end{aligned} \tag{36}$$

Note that this is an instance of an evaluation rule where unnecessary terms may be disposed since the bound variable x_1 may not actually appear free in the body t_b of the native function. Thus, we prioritize this evaluation rule over the other ones.

The previous two computation rules describe how native function applications are evaluated. In the case of a foreign function, we have to make sure that both the left-hand term as well as all the argument terms of the application are values. If this is the case, we can schedule the external operation and replace the application with a future term.

$$(\lambda_{\text{frn}}\ (x_i : U_i{}^{i\in 1..n}) \rightarrow^{\text{frn}} U_{\text{ret}}\ \textbf{in}\ l\ .\ s)\ \left(x_i = v_i{}^{i\in 1..n}\right)\longrightarrow \textbf{fut}[U_{\text{ret}}]\ k_1 \tag{37}$$

Scheduling of an external operator should be done only if absolutely necessary. If there is any other evaluation rule, it should take precedence over the above rule to account for the possibility that an unnecessary computation can be disposed of.

6.2 Conditionals

The evaluation rules for conditionals are straightforward. In case the condition-term evaluates to **true** the conditional evaluates to the then-term, otherwise it evaluates to the else-term.

$$\textbf{if true then } t_2 \textbf{ else } t_3 \longrightarrow t_2 \tag{38}$$

$$\textbf{if false then } t_2 \textbf{ else } t_3 \longrightarrow t_3 \tag{39}$$

6.3 General recursion

We achieve general recursion in the presence of a simple type system by explicitly adding a fixpoint operator as a syntactic category. Here, we give its evaluation rules. Evaluation of the fixpoint operator replaces all occurrences of the name of the function x_f in the body term t_b with the fixpoint operator itself, resulting in a non-fixpointed term possibly containing the fixpoint operator again.

$$\begin{array}{ll} & \textbf{fix } \lambda_{\text{ntv}}\ (x_f : T_f, x_i : T_i^{i\in 1..n})\ .\ t_b \\ \longrightarrow & \lambda_{\text{ntv}}\ (x_i : T_i^{i\in 1..n})\ .\ [x_f \mapsto (\textbf{fix } \lambda_{\text{ntv}}\ (x_f : T_f, x_i : T_i^{i\in 1..n})\ .\ t_b)]t_b \end{array} \tag{40}$$

6.4 The isnil operator

The isnil operator determines whether a list is empty. Its evaluation rules are straightforward.

$$\textbf{isnil}[S]\ (\textbf{nil}[T]) \longrightarrow \textbf{true} \tag{41}$$

$$\textbf{isnil}[S]\ (\textbf{cons}[T]\ t_1\ t_2) \longrightarrow \textbf{false} \tag{42}$$

Note that in the last rule, we do not need to evaluate the head t_1 and tail t_2 of the list. Since this rule is also one that potentially disposes of unnecessary computation it should be given precedence over the other rules.

6.5 The zip operator

Here, we give evaluation rules for the case that the left-hand side of an application is the zip operator. The typing rules of the zip operator (see Section 5.7) guarantee us that any zipped argument has to be of list type. The first rule states that if any of the zipped arguments is the empty list **nil**$[S]$, then the whole application evaluates to the empty list **nil**$[T_{\text{ret}}]$. Thus, if the zipped argument lists have different lengths, the shortest of the argument lists determines the length of the result list.

$$\frac{\exists j, i\ \ (y_j = x_i \wedge t_i = \textbf{nil}[S])}{(\textbf{zip}[y_j^{j\in 1..m} | T_{\text{ret}}]\ t_f)\ \left(x_i = t_i^{i\in 1..n}\right) \longrightarrow \textbf{nil}[T_{\text{ret}}]} \tag{43}$$

Now the only other possibility is that each of the zipped arguments is the non-empty list. In this case, the function term t_f is applied to the head of each list (together with any remaining non-zipped arguments). This way, a list of applications is constructed.

$$\frac{\forall j\ \ \exists i\ \ (y_j = x_i \wedge t_i = \mathbf{cons}[S]\ t_{i\mathrm{hd}}\ t_{i\mathrm{tl}})}{\begin{array}{l}(\mathbf{zip}[y_j^{j\in 1..m} | T_{\mathrm{ret}}]\ t_f)\ \left(x_i = t_i{}^{i\in 1..n}\right) \\ \longrightarrow \mathbf{cons}[T_{\mathrm{ret}}]\quad (t_f\ \left(x_i = \begin{cases} t_{i\mathrm{hd}} & ,\exists j\ \ y_j = x_i \\ t_i & ,\mathrm{else}\end{cases}{}^{i\in 1..n}\right)) \\ \qquad\qquad\qquad ((\mathbf{zip}[y_j^{j\in 1..m} | T_{\mathrm{ret}}]\ t_f)\ \left(x_i = \begin{cases} t_{i\mathrm{tl}} & ,\exists j\ \ y_j = x_i \\ t_i & ,\mathrm{else}\end{cases}{}^{i\in 1..n}\right))\end{array}} \tag{44}$$

The zip operator is important because it allows the processing of lists without directly decomposing them with head or tail and has the map operator as a special case. Its evaluation is recursively defined but on the level of the meta-language instead of the target language. This lets us avoid the use of the fixpoint operator. In the consequence, we can create data-parallel programs without making explicit use of general recursion.

6.6 Tuples

Like lists tuples are a compound data type in Cuneiform. Again, the following evaluation rule rids us of unnecessary computation and, thus, should be given precedence over other rules.

$$\pi_j\ \{t_i^{i\in 1..n}\} \longrightarrow t_j \tag{45}$$

The remaining rules show how a list projection is transformed into a list of ordinary projections.

$$\Pi_j\ (\mathbf{nil}[\{T_i^{i\in 1..n}\}]) \longrightarrow \mathbf{nil}[T_j] \tag{46}$$

$$\Pi_j\ (\mathbf{cons}[\{T_i^{i\in 1..n}\}]\ t_{11}\ t_{12}) \longrightarrow \mathbf{cons}[T_j]\ (\pi_j\ t_{11})\ (\Pi_j\ t_{12}) \tag{47}$$

6.7 Let bindings

The evaluation of let bindings is straightforward.

$$\mathbf{let}\ x : T_1 = t_1\ \mathbf{in}\ t_2 \longrightarrow [x \mapsto t_1]t_2 \tag{48}$$

6.8 Congruence rules

The reduction rules given in the previous sections establish a notion of reduction. To turn this notion of reduction into a standard reduction relation, we need to determine how sub-terms should be evaluated. This is done by augmenting the notion of reduction with the following congruence rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \left(x_i = t_{2i}{}^{i\in 1..n}\right) \longrightarrow t_1'\ \left(x_i = t_{2i}{}^{i\in 1..n}\right)} \tag{49}$$

$$\frac{t_j \longrightarrow t'_j}{\begin{array}{ll} & (\lambda_{\text{frn}}\ (x_i : U_i^{i\in 1..n}) \to^{\text{frn}} U_{\text{ret}}\ \textbf{in}\ l\ .\ s)\ \left(x_i = t_i^{i\in 1..j-1}, x_j = t_j, x_i = t_i^{i\in j+1..n}\right) \\ \longrightarrow & (\lambda_{\text{frn}}\ (x_i : U_i^{i\in 1..n}) \to^{\text{frn}} U_{\text{ret}}\ \textbf{in}\ l\ .\ s)\ \left(x_i = t_i^{i\in 1..j-1}, x_j = t'_j, x_i = t_i^{i\in j+1..n}\right) \end{array}} \tag{50}$$

$$\frac{t_1 \longrightarrow t'_1}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \longrightarrow \textbf{if}\ t'_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3} \tag{51}$$

$$\frac{t_1 \longrightarrow t'_1}{\textbf{cons}[T]\ t_1\ t_2 \longrightarrow \textbf{cons}[T]\ t'_1\ t_2} \tag{52}$$

$$\frac{t_2 \longrightarrow t'_2}{\textbf{cons}[T]\ t_1\ t_2 \longrightarrow \textbf{cons}[T]\ t_1\ t'_2} \tag{53}$$

$$\frac{t_1 \longrightarrow t'_1}{\textbf{isnil}[T]\ t_1 \longrightarrow \textbf{isnil}[T]\ t'_1} \tag{54}$$

$$\frac{t_j \longrightarrow t'_j}{\{t_i^{i\in 1..j-1}, t_j, t_k^{k\in j+1..n}\} \longrightarrow \{t_i^{i\in 1..j-1}, t'_j, t_k^{k\in j+1..n}\}} \tag{55}$$

$$\frac{t_1 \longrightarrow t'_1}{\pi_i\ t_1 \longrightarrow \pi_i\ t'_1} \tag{56}$$

$$\frac{t_1 \longrightarrow t'_1}{\Pi_i\ t_1 \longrightarrow \Pi_i\ t'_1} \tag{57}$$

The standard reduction relation does not traverse all possible sub-terms: Native function bodies and the then- and else-branches of conditionals are left unevaluated. Reduction also does not traverse inside the fixpoint operator or the zip operator. Function arguments are evaluated only if the function term is a foreign function.

Note that the congruence rules leave open, which function argument to evaluate first. Also, list or tuple elements may be evaluated in any order. This non-determinism reflects the frankness with which we look for relevant foreign operations where they first appear in the program term.

7 Implementations

The primary purpose of defining the computation semantics of Cuneiform is to ease the implementation of Cuneiform interpreters and to ensure consistency among implementations, thus improving portability. Ideally, each of the inference rules given in this paper corresponds to one statement in whatever programming language is chosen as a host language for a Cuneiform interpreter. The more the host language lends itself to the step-wise premise–conclusion style of notation used in the definitions of operational semantics, the more directly a semantics can be transcribed into this host language. Functional programming languages with support for pattern matching like Standard ML, OCaml, Haskell, pattern matching-enabled Lisps, or Erlang are particularly suitable for unlabored transcription although any general purpose programming language can do. For the sake of demonstration, we present a reference implementation of a Cuneiform interpreter in Erlang plainly implementing the abstract syntax, typing rules, and reduction rules.[5] However, this reference implementation lacks a number of components for making it actually usable. It has neither a parser to produce the in-memory representation of the program term from an input string constituting the concrete program nor an execution environment that could actually perform foreign language computations. Its sole purpose is to show how the rules given in this paper can be transcribed in a general purpose programming language. In addition to demonstrating how an interpreter can be created from the reduction rules shown in this paper, the reference implementation served as a test bed in which we tried out simple programs and refined the reduction rules according to the experiences we have made. A collection of unit tests solidifies these experiences. Last, we used Erlang QuickCheck (Arts *et al.*, 2006; Hughes, 2007) to informally convince ourselves of the validity of important language properties like progress, preservation, or the assertion that evaluation is total for well-typed non-value terms. A formal treatment of these properties is, however, left for future work.

A complete, yet more convoluted, implementation of Cuneiform, including a Leex/Yecc-based parser, a local execution environment as well as an execution environment running on HTCondor are also available.[6] Finally, Hi-WAY[7] is an execution environment based on Hadoop. It supports Cuneiform and a number of other workflow languages (Bux *et al.*, 2017).

8 Related work

A formal specification of syntax or semantics is missing for the majority of workflow languages in use today. However, for a number of languages, formal models have been developed, notably Pegasus (Budiu & Goldstein, 2002) and Kepler (McPhillips *et al.*, 2006; Goderis *et al.*, 2007; Zinn *et al.*, 2009). A scientific workflow language with an extensive body of work regarding its semantics is Taverna (Hull *et al.*, 2006). Taverna is a graphical scientific workflow language targeting users in bioinformatics and other life sciences. It focuses on the integration of heterogeneous software and web-services. The original formulation of Taverna's semantics (Turi *et al.*, 2007) is formulated as a natural semantics based on computational lambda calculus (Moggi, 1991). It was succeeded by a number of refinements formulated as state transitions with trace semantics (Hidders & Sroka, 2008; Sroka & Hidders, 2009a; Sroka & Hidders, 2009b; Sroka *et al.*, 2010) pushing the understanding of Taverna's semantics in a process-oriented direction. The publications characterizing Taverna's semantics emphasize the fact that Taverna services can have side effects or be non-deterministic, i.e., that the order in which services are invoked potentially influences the workflow result.

The idea that data dependencies in scientific workflows can be expressed in lambda calculus has been formulated several times. Ludäscher and Altintas presented a way

[5] https://github.com/joergen7/cf_reference
[6] https://github.com/joergen7/cuneiform
[7] https://github.com/marcbux/Hi-WAY

to express scientific workflows in Haskell syntax (Ludäscher & Altintas, 2003) and observed that parallelization is directly derivable. Kelly *et al.* (2009; 2011) have defined data dependencies among web-services directly in untyped lambda calculus. Cuneiform differs from Kelly's approach in that we make a minor modification to the canonical presentation of the simply typed lambda calculus allowing the uniform notation of abstractions (native functions) and external operators (foreign functions).

Apart from this modification, we have stuck as closely as possible to a simply typed lambda calculus. However, existing scientific workflow languages are rooted in various formalisms. For example, Pig Latin is inspired by SQL and is, thus, rooted in relational algebra. Taverna started out with a functional formulation but turned in the direction of trace semantics. Kepler emphasizes its relationship with process networks and the actor model but its orchestrator concept makes execution behavior actually exchangeable. Nextflow (Di Tommaso *et al.*, 2017) has been designed around the concept of channels and is, therefore, closely related to process calculi like CSP. Finally, some workflow languages have been designed around Petri Nets, e.g., Grid-Flow (Guan *et al.*, 2006).

Similar to Turi et al., we introduce Cuneiform's semantics by first introducing its syntax and typing rules and then discuss its evaluation rules. In contrast, we introduced those evaluation rules in the form of a *structural operational semantics* (Plotkin, 1981) which defines evaluation as the repeated application of small-step evaluation rules. An alternative way to present an operational semantics is the form of *natural semantics*, which defines evaluation of a program in a single big step (Kahn, 1987). Yet, another candidate would have been a denotational semantics which transforms an expression until it is composed only of symbols and operations with an intuitive interpretation (Tennent, 1976).

Cuneiform's approach and parallel execution is inspired by distributed functional programming languages like Eden (Breitinger *et al.*, 1998; Loogen *et al.*, 2005) or the distributed Haskell implementation GDH (Pointon *et al.*, 2001). Its take on large-scale data analysis on top of a distributed file system is inspired by MapReduce (Dean & Ghemawat, 2008; White, 2012) and Spark (Zaharia *et al.*, 2010; Zaharia *et al.*, 2012). Eventually, Cuneiform's integration of external software is inspired by scientific workflow languages like Taverna (Hull *et al.*, 2006) or Galaxy (Goecks *et al.*, 2010). However, a language combining these advantages in a large-scale functional language that is agnostic about a function body's implementation language has, to our knowledge, not been otherwise conceived.

9 Conclusion

We have introduced the formal semantics of Cuneiform,[8] a minimal language for large-scale scientific data analysis by establishing its abstract syntax, type system, and evaluation rules and exemplified their implementation in Erlang. The definition

[8] https://cuneiform-lang.org/

of Cuneiform's semantics reduces the time and effort necessary to create a consensual language interpreter and, thus, improves the language's portability. At the same time, the type system is the foundation for the deduction of the language's safety up to black-box operators. Cuneiform is rooted in functional programming and, to be comparable, sticks as closely as possible to the simply typed lambda calculus.

Being an organizational language, Cuneiform focuses on expressing data dependencies among external operations while the actual computation is deferred to an external language. In contrast, general purpose programming languages take control not only over the management of data dependencies but also over the low-level computation at the operator level. This distinction between the organizational part, the programming in the large, and the computational part, the programming in the small (DeRemer & Kron, 1976), allows us to consider the feature set, of the organizational language independent from the feature set of the operator-level programming language. Design decisions about type systems, error handling, or state can be decoupled. Independence, which enables parallelism, can be emphasized on the organizational level at the price of determinism while performance can be emphasized on the operator level. This separation also allows Cuneiform to be liberal toward the language, an external operator is implemented in. All that matters is that there is a defined interface between both the organizational and the operator level.

For future work, it may be desirable to extend Cuneiform with more sophisticated base data types like integers and floating point numbers and compound data types like maps. Another possibility is to develop the language features shown here to more sophistication. For example, pattern matching may be added or a more versatile and extensible type system with subtyping may replace the simple type system to better account for the meaning and compatibility of user-specified data formats. Finally, it should be possible to use Cuneiform as a compilation target for a more derived language with a different focus or application scenario, adding syntactic sugar to ease the expression of common use cases in different research areas. Cuneiform may be seen as a kernel language for a family of distributed programming languages just like the lambda calculus is a suitable kernel language for a large variety of programming languages beyond the functional programming paradigm.

References

Armstrong, J., Virding, R., Wikström, C. & Williams, M. (1996) Concurrent Programming in ERLANG (2nd Ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Arts, T., Hughes, J., Johansson, J. & Wiger, U. (2006) Testing telecoms software with quviq quickcheck. In Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ERLANG '06. New York, NY, USA: ACM.

Bessani, A., Brandt, J., Bux, M., Cogo, V., Dimitrova, L., Dowling, J., Gholami, A., Hakimzadeh, K., Hummel, M., Ismail, M., Laure, E., Leser, U., Litton, J.-E., Martinez, R., Niazi, S., Reichel, J. & Zimmermann, K. (2015) Biobankcloud: A platform for the secure storage, sharing, and processing of large biomedical data sets. In Proceedings of 1st International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2015).

Bishop, C. M. (2006) Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Brandt, J., Bux, M. & Leser, U. (2015 March) Cuneiform: A functional language for large scale scientific data analysis. In Proceedings of the Workshops of the EDBT/ICDT, vol. 1330, pp. 17–26.

Breitinger, S., Klusik, U. & Loogen, R. (1998) *From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View*. Berlin, Heidelberg: Springer, pp. 318–334.

Budiu, M. & Goldstein, S. C. (2002) *Pegasus: An Efficient Intermediate Representation*. Technical Report. DTIC Document.

Bux, M., Brandt, J., Lipka, C., Hakimzadeh, K., Dowling, J. & Leser, U. (2015 September) Saasfee: Scalable scientific workflow execution engine. In Proceedings of the VLDB Endowment, vol. 8, pp. 1892–1895.

Bux, M., Brandt, J., Witt, C., Dowling, J. & Leser, U. (2017) Hi-way: Execution of scientific workflows on hadoop yarn. In Proceedings of the 20th International Conference on Extending Database Technology (EDBT).

Church, A. & Rosser, J. B. (1936) Some properties of conversion. *Trans. Am. Math. Soc.* **39**(3), 472–482.

Cohen-Boulakia, S. & Leser, U. (2011) Search, adapt, and reuse: The future of scientific workflows. *Sigmod Rec.* **40**(2), 6–16.

Dean, J. & Ghemawat, S. (2008) Mapreduce: Simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113.

Deelman, E., Livny, M., Mehta, G., Pavlo, A., Singh, G., Su, M.-H., Vahi, K. & Wenger, R. K. (2006) Pegasus and dagman from concept to execution: Mapping scientific workflows onto today's cyberinfrastructure. In *High Performance Computing Workshop*, pp. 56–74.

DeRemer, F. L. & Kron, H. H. (1976) *Programming-in-the-Large versus Programming-in-the-Small*. Berlin, Heidelberg: Springer, pp. 80–89.

Di Tommaso Paolo, Chatzou Maria, Floden Evan W., Barja Pablo Prieto, Palumbo Emilio & Notredame Cedric (2017). Nextflow enables reproducible computational workflows. *Nat Biotech*, **35**(4), 316–319.

Duda, R. O., Hart, P. E. & Stork, D. G. (2012) *Pattern Classification*. John Wiley & Sons.

Efron, B. & Tibshirani, R. J. (1994) *An Introduction to the Bootstrap*. CRC Press.

Goderis, A., Brooks, C., Altintas, I., Lee, E. A. & Goble, C. (2007) *Composing Different Models of Computation in Kepler and Ptolemy ii*. Berlin, Heidelberg: Springer, pp. 182–190.

Goecks, J., Nekrutenko, A. & Taylor, J. (2010) Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol.* **11**(8), 1.

Guan, Z., Hernandez, F., Bangalore, P., Gray, J., Skjellum, A., Velusamy, V. & Liu, Y. (2006) Grid-flow: A grid-enabled scientific workflow system with a petri-net-based interface. *Concurr. Comput.: Pract. Exp.* **18**(10), 1115–1140.

Harper, R. (2016) *Practical Foundations for Programming Languages*. Cambridge University Press.

Haykin, S. S., Haykin, S. S., Haykin, S. S. & Haykin, S. S. (2009) *Neural Networks and Learning Machines*, vol. 3. Upper Saddle River, NJ, USA: Pearson.

Hennessy, M. (1990) *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley & Sons.

Hey, T., *et al.* (2009) *The Fourth Paradigm: Data-Intensive Scientific Discovery*, vol. 1. Microsoft research Redmond, WA.

Hidders, J. & Sroka, J. (2008) *Towards a Calculus for Collection-Oriented Scientific Workflows with Side Effects*. Berlin, Heidelberg: Springer, pp. 374–391.

Hughes, J. (2007) *Quickcheck Testing for Fun and Profit*. Berlin, Heidelberg: Springer, pp. 1–32.

Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P. & Oinn, T. (2006) Taverna: A tool for building and running workflows of services. *Nucleic Acids Res*. **34**(suppl 2), W729–W732.

Kahn, G. (1987) *Natural Semantics*. Berlin, Heidelberg: Springer, pp. 22–39.

Kalayci, S., Dasgupta, G., Fong, L., Ezenwoye, O. & Sadjadi, S. M. (2010) Distributed and adaptive execution of condor dagman workflows. In SEKE, pp. 587–590.

Kelly, P. M. (2011) Applying functional programming theory to the design of workflow engines. PhD thesis, University of Adelaide.

Kelly, P. M., Coddington, P. D. & Wendelborn, A. L. (2009) Lambda calculus as a workflow model. *Concurr. Comput.: Pract. Exp*. **21**(16), 1999–2017.

Köster, J. & Rahmann, S. (2012) SnakemakeâĂŤa scalable bioinformatics workflow engine. *Bioinformatics* **28**(19), 2520–2522.

Liu, J., Pacitti, E., Valduriez, P. & Mattoso, M. (2015) A survey of data-intensive scientific workflow management. *J. Grid Comput*. **13**(4), 457–493.

Loogen, R., Ortega-Mallén, Y. & Peña-Marí, R. (2005) Parallel functional programming in eden. *J. Funct. Program*. **15**(03), 431–475.

Ludäscher, B. & Altintas, I. (2003) On providing declarative design and programming constructs for scientific workflows based on process networks. San Diego Supercomputer Center.

Manly, B. F. J. (2006) *Randomization, Bootstrap and Monte Carlo Methods in Biology*, vol. 70. CRC Press.

McPhillips, T., Bowers, S. & Ludäscher, B. (2006) *Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data*. Berlin, Heidelberg: Springer, pp. 248–263.

Michaelson, G. (2011) *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation.

Moggi, E. (1991) Notions of computation and monads. *Inform. Comput*. **93**(1), 55–92.

Myers, K. S., Yan, H., Ong, I. M., Chung, D., Liang, K., Tran, F, Keleş, S., Landick, R. & Kiley, P. J. (2013) Genome-scale analysis of escherichia coli fnr reveals complex features of transcription factor binding. *Plos Genet* **9**(6), e1003565.

Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A. & Wroe, C. (2006) Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, **18**(10), 1067–1100.

Olston, C., Reed, B., Srivastava, U., Kumar, R. & Tomkins, A. (2008) Pig latin: A not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08. New York, NY, USA: ACM, pp. 1099–1110.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT press.

Plotkin, G. D. (1981) A structural approach to operational semantics. Computer Science Department, Aarhus University Aarhus, Denmark.

Pointon, R. F., Trinder, P. W. & Loidl, H.-W. (2001) *The Design and Implementation of Glasgow Distributed Haskell*. Berlin, Heidelberg: Springer, pp. 53–70.

Sroka, J. & Hidders, J. (2009a) Towards a formal semantics for the process model of the taverna workbench. Part i. *Fundam. Inform*. **92**(3), 279–299.

Sroka, J. & Hidders, J. (2009b) Towards a formal semantics for the process model of the taverna workbench. Part ii. *Fundam. Inform*. **92**(4), 373–396.

Sroka, J., Hidders, J., Missier, P. & Goble, C. (2010) A formal semantics for the taverna 2 workflow model. *J. Comput. Syst. Sci.* **76**(6), 490–508.

Tennent, R. D. (1976) The denotational semantics of programming languages. *Commun. ACM* **19**(8), 437–453.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P. & Murthy, R. (2009) Hive: A warehousing solution over a map-reduce framework. *Proc. Vldb Endowment* **2**(2), 1626–1629.

Turi, D., Missier, P., Goble, C., De Roure, D. & Oinn, T. (2007) Taverna workflows: Syntax and semantics. In Proceedings of IEEE International Conference on e-Science and Grid Computing. IEEE, pp. 441–448.

White, T. (2012) *Hadoop: The Definitive Guide*. O'Reilly Media, Inc..

Winskel, G. (1993) *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010) Spark: Cluster computing with working sets. *Hotcloud* **10**(10–10), 95.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., Franklin, M., Shenker, S. & Stoica, I. (2012) Fast and interactive analytics over hadoop data with spark. *Usenix Login* **37**(4), 45–51.

Zinn, D., Bowers, S., McPhillips, T. & Ludäscher, B. (2009) Scientific workflow design with data assembly lines. In Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, WORKS '09. New York, NY, USA: ACM, pp. 14:1–14:10.

A ChIP-Seq workflow

```
%% TASK DEFINITIONS

% sra-tools
deftask fastq-dump( fastq( File ) : sra( File ) )in bash *{
  fastq=$sra.fastq
  fastq-dump -Z $sra > $fastq
}*

% FastQC
deftask fastqc( zip( File ) : fq( File ) )in bash *{
  fastqc -f fastq --noextract -o ./ $fq
  zip=‘ls *.zip‘
}*

deftask bowtie-build( idx( File ) : fa( File ) )in bash *{
  bowtie-build $fa btidx
  idx=idx.tar
  tar cf $idx btidx.* --remove-files
}*

deftask bowtie-align( sam( File ) : idx( File ) fq( File ) )in bash *{
  tar xf $idx
  sam=$fq.sam
  bowtie btidx -q $fq -v 2 -m 1 -3 1 -S -p 2 > $sam
}*

deftask macs(
    peaks( File ) summits( File ) <xls( File )>
    bedgraph_tag( File ) bedgraph_ctl( File )
  : tag_sam( File ) ctl_sam( File ) )in bash *{

  macs14 -t $tag_sam -c $ctl_sam --format SAM --gsize 4639675 --name "macs14" \
    --bw 400 --keep-dup 1 --bdg --single-profile --diag

  peaks=macs14_peaks.bed
  summits=macs14_summits.bed
  xls=(macs14_diag.xls macs14_negative_peaks.xls)
  bedgraph_tag=macs14_MACS_bedGraph/treat/macs14_treat_afterfiting_all.bdg.gz
  bedgraph_ctl=macs14_MACS_bedGraph/control/macs14_control_afterfiting_all.bdg.gz
}*

deftask samtools-sort( sorted_bam( File ) : sam( File ) )in bash *{
  sorted_bam=sorted.bam
  samtools view -bS $sam | samtools sort -o $sorted_bam -
}*

deftask samtools-rmdup( dedup_bam( File ) : bam( File ) )in bash *{
  dedup_bam=dedup.bam
  samtools rmdup -s $bam $dedup_bam
}*

deftask samtools-index( bai( File ) : bam( File ) )in bash *{
  bai=$bam.bai
  samtools index $bam $bai
}*

deftask samtools-faidx( fai( File ) : fa( File ) )in bash *{
  fai=$fa.fai
  samtools faidx $fa
}*

deftask bamcoverage( bedgraph( File ) : bam( File ) bai( File ) )in bash *{
  bedgraph=$bam.bedgraph
  ln -sf $bai $bam.bai
  bamCoverage --bam $bam --outFileName $bedgraph --normalizeTo1x 4639675 \
  --outFileFormat bedgraph
}*

deftask deeptools( bedgraph( File ) : sam( File ) ) {
  sorted_bam         = samtools-sort( sam: sam );
  dedup_sorted_bam   = samtools-rmdup( bam: sorted_bam );
  dedup_sorted_bai   = samtools-index( bam: dedup_sorted_bam );
  bedgraph = bamcoverage( bam: dedup_sorted_bam, bai: dedup_sorted_bai );
}

deftask bedtools-getfasta(
    bed_fa( File )
  : fa( File ) fai( File ) bed( File ) )in bash *{

  bed_fa=$bed.fa
  ln -sf $fai $fa.fai
  bedtools getfasta -fi $fa -bed $bed -fo $bed_fa
}*

deftask restrict-peaks( restricted_bed( File ) : bed( File ) )in bash *{
  restricted_bed=$bed.100.bed
  perl -lane '$start=$F[1]+100; $end = $F[2]-100 ; print "$F[0]\t$start\t$end"' \
    $bed > $restricted_bed
}*

%% INPUT DATA

tag_sra = "sra/SRR576933.sra";
ctl_sra = "sra/SRR576938.sra";
fa      = "ref/Escherichia_coli_K_12_MG1655.fasta";

%% WORKFLOW DEFINITION

tag_fq = fastq-dump( sra: tag_sra );
ctl_fq = fastq-dump( sra: ctl_sra );

qc = fastqc( fq: tag_fq ctl_fq );

fai = samtools-faidx( fa: fa );

idx     = bowtie-build( fa: fa );
tag_sam = bowtie-align( idx: idx, fq: tag_fq );
ctl_sam = bowtie-align( idx: idx, fq: ctl_fq );
```

```
peaks summits xls tag_macs_bedgraph ctl_macs_bedgraph = macs(
  tag_sam: tag_sam, ctl_sam: ctl_sam );

tag_deeptools_bedgraph = deeptools( sam: tag_sam );
ctl_deeptools_bedgraph = deeptools( sam: ctl_sam );

peaks_100    = restrict-peaks( bed: peaks );

peaks_fa     = bedtools-getfasta( fa: fa, fai: fai, bed: peaks );
peaks_100_fa = bedtools-getfasta( fa: fa, fai: fai, bed: peaks_100 );

%% QUERY

qc peaks peaks_fa peaks_100_fa
tag_macs_bedgraph ctl_macs_bedgraph
tag_deeptools_bedgraph ctl_deeptools_bedgraph;
```