WIKIPEDIA

Cuneiform (programming language)

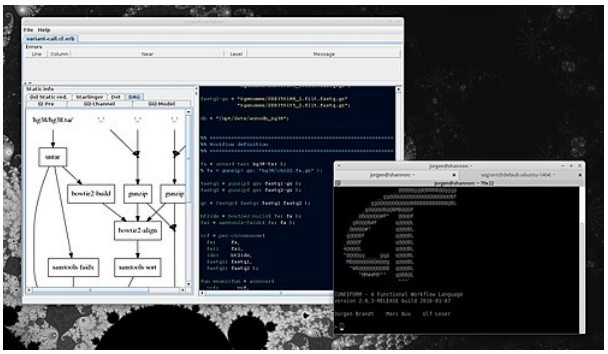
From Wikipedia, the free encyclopedia

Cuneiform is an open-source workflow language for large-scale scientific data analysis.[1][2] It is a statically typed functional programming language promoting parallel computing. It features a versatile foreign function interface allowing users to integrate software from many external programming languages. At the organizational level Cuneiform provides facilities like conditional branching and general recursion making it Turing-complete. In this, Cuneiform is the attempt to close the gap between scientific workflow systems like Taverna, KNIME, or Galaxy and large-scale data analysis programming models like MapReduce or Pig Latin while offering the generality of a functional programming language.

Cuneiform is implemented in distributed Erlang. If run in distributed mode it drives a POSIX-compliant distributed file system like Gluster or Ceph (or a FUSE integration of some other file system, e.g., HDFS). Alternatively, Cuneiform scripts can be executed on top of HTCondor or Hadoop.[3][4][5][6]

Cuneiform is influenced by the work of Peter Kelly who proposes functional programming as a model for scientific workflow execution.[7][8] In this, Cuneiform is distinct from related workflow languages based on dataflow programming like Swift.[9]

Cuneiform



Paradigm	functional, scientific workflow
Designed by	Jörgen Brandt
First appeared	2013
Stable release	3.0.4 / November 19, 2018
Typing discipline	static, simple types
Implementation language	Erlang
OS	Linux, MacOS
License	Apache License 2.0
Filename extensions	.cfl
Website	cuneiform-lang.org (http://cuneiform-lang.org/)

Influenced by

Swift (parallel scripting language)

Contents

- 1 External software integration
- 2 Type system
  - 2.1 Base data types
  - 2.2 Records and pattern matching
  - 2.3 Lists and list processing
- 3 Parallel execution
- 4 Examples
- 5 Release history

- 5.1 Version 1
- 5.2 Version 2
- 5.3 Version 3
- 6 References

External software integration

External tools and libraries (e.g., R or Python libraries) are integrated via a foreign function interface. In this it resembles, e.g., KNIME which allows the use of external software through snippet nodes, or Taverna which offers BeanShell services for integrating Java software. By defining a task in a foreign language it is possible to use the API of an external tool or library. This way, tools can be integrated directly without the need of writing a wrapper or reimplementing the tool.[10]

Currently supported foreign programming languages are:

- Bash
- Elixir
- Erlang
- Java
- JavaScript
- MATLAB
- GNU Octave
- Perl
- Python
- R
- Racket

Foreign language support for AWK and gnuplot are planned additions.

Type system

Cuneiform provides a simple, statically checked type system.[11] While Cuneiform provides lists as compound data types it omits traditional list accessors (head and tail) to avoid the possibility of runtime errors which might arise when accessing the empty list. Instead lists are accessed in an all-or-nothing fashion by only mapping or folding over them. Additionally, Cuneiform omits (at the organizational level) arithmetics which excludes the possibility of division by zero. The omission of any partially defined operation allows to guarantee that runtime errors can arise exclusively in foreign code.

Base data types

As base data types Cuneiform provides Booleans, strings, and files. Herein, files are used to exchange data in arbitrary format between foreign functions.

Records and pattern matching

Cuneiform provides records (structs) as compound data types. The example below shows the definition of a variable `r` being a record with two fields `a1` and `a2`, the first

being a string and the second being a Boolean.

```
let r : <a1 : Str, a2 : Bool> =
  <a1 = "my string", a2 = true>;
```

Records can be accessed either via projection or via pattern matching. The example below extracts the two fields `a1` and `a2` from the record `r`.

```
let a1 : Str = ( r|a1 );

let <a2 = a2 : Bool> = r;
```

Lists and list processing

Furthermore, Cuneiform provides lists as compound data types. The example below shows the definition of a variable `xs` being a file list with three elements.

```
let xs : [File] =
  ['a.txt', 'b.txt', 'c.txt' : File];
```

Lists can be processed with the for and fold operators. Herein, the for operator can be given multiple lists to consume list element-wise (similar to `for/list` in Racket, `mapcar` in Common Lisp or `zipwith` in Erlang).

The example below shows how to map over a single list, the result being a file list.

```
for x <- xs do
  process-one( arg1 = x )
  : File
end;
```

The example below shows how to zip two lists the result also being a file list.

```
for x <- xs, y <- ys do
  process-two( arg1 = x, arg2 = y )
  : File
end;
```

Finally, lists can be aggregated by using the fold operator. The following example sums up the elements of a list.

```
  fold acc = 0, x <- xs do
    add( a = acc, b = x )
  end;
```

Parallel execution

Cuneiform is a purely functional language, i.e., it does not support mutable references. In the consequence, it can use subterm-independence to divide a program into parallelizable portions. The Cuneiform scheduler distributes these portions to worker nodes. In addition, Cuneiform uses a Call-by-Name evaluation strategy to compute values only if they contribute to the computation result. Finally,

foreign function applications are memoized to speed up computations that contain previously derived results.

For example, the following Cuneiform program allows the applications of `f` and `g` to run in parallel while `h` is dependent and can be started only when both `f` and `g` are finished.

```
let output-of-f : File = f();
let output-of-g : File = g();

h( f = output-of-f, g = output-of-g );
```

The following Cuneiform program creates three parallel applications of the function `f` by mapping `f` over a three-element list:

```
let xs : [File] =
  ['a.txt', 'b.txt', 'c.txt' : File];

for x <- xs do
  f( x = x )
  : File
end;
```

Similarly, the applications of `f` and `g` are independent in the construction of the record `r` and can, thus, be run in parallel:

```
let r : <a : File, b : File> =
  &lt;a = f(), b = g()&gt;;
```

Examples

A hello-world script:

```
def greet( person : Str ) -> <out : Str>
in Bash *{
  out="Hello $person"
}*

( greet( person = "world" )|out );
```

This script defines a task `greet` in Bash which prepends `"Hello "` to its string argument `person`. The function produces a record with a single string field `out`. Applying `greet`, binding the argument `person` to the string `"world"` produces the record `<out = "Hello world">`. Projecting this record to its field `out` evaluates the string `"Hello world"`.

Command line tools can be integrated by defining a task in Bash:

```
def samtoolsSort( bam : File ) -> <sorted : File>
in Bash *{
  sorted=sorted.bam
  samtools sort -m 2G $bam -o $sorted
}*
```

In this example a task `samtoolsSort` is defined. It calls the tool SAMtools, consuming

an input file, in BAM format, and producing a sorted output file, also in BAM format.

Release history

Version	Appearance	Implementation Language	Distribution Platform	Foreign Languages
1.0.0	May 2014	Java	Apache Hadoop	Bash, Common Lisp, GNU Octave, Perl, Python, R, Scala
2.0.x	Mar. 2015	Java	HTCondor, Apache Hadoop	Bash, BeanShell, Common Lisp, MATLAB, GNU Octave, Perl, Python, R, Scala
2.2.x	Apr. 2016	Erlang	HTCondor, Apache Hadoop	Bash, Perl, Python, R
3.0.x	Feb. 2018	Erlang	Distributed Erlang	Bash, Erlang, Java, MATLAB, GNU Octave, Perl, Python, R, Racket

In April 2016, Cuneiform's implementation language switched from Java to Erlang and, in February 2018, its major distributed execution platform changed from a Hadoop to distributed Erlang. Additionally, from 2015 to 2018 HTCondor had been maintained as an alternative execution platform.

Cuneiform's surface syntax was revised twice, as reflected in the major version number.

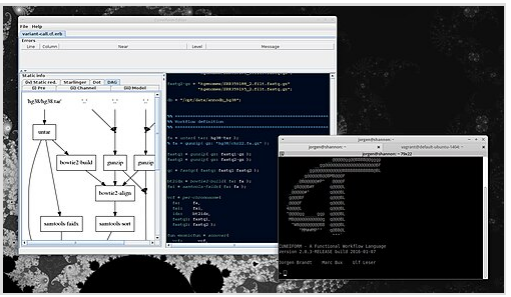
Version 1

In its first draft published in May 2014, Cuneiform was closely related to Make in that it constructed a static data dependency graph which the interpreter traversed during execution. The major difference to later versions was the lack of conditionals, recursion, or static type checking. Files were distinguished from strings by juxtaposing single-quoted string values with a tilde ~. The script's query expression was introduced with the `target` keyword. Bash was the default foreign language. Function application had to be performed using an `apply` form that took `task` as its first keyword argument. One year later, this surface syntax was replaced by a streamlined but similar version.

The following example script downloads a reference genome from an FTP server.

```
declare download-ref-genome;

deftask download-fa( fa : ~path ~id ) *{
    wget $path/$id.fa.gz
    gunzip $id.fa.gz
    mv $id.fa $fa
}*

ref-genome-path = ~'ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes';
ref-genome-id = ~'chr22';

ref-genome = apply(
    task : download-fa
    path : ref-genome-path
    id : ref-genome-id
);

target ref-genome;
```

Version 2



Swing-based editor and REPL for Cuneiform 2.0.3

The second draft of the Cuneiform surface syntax, first published in March 2015, remained in use for three years outlasting the transition from Java to Erlang as Cuneiform's implementation language. Evaluation differs from earlier approaches in that the interpreter reduces a query expression instead of traversing a static graph. During the time the surface syntax remained in use the interpreter was formalized and simplified which resulted in a first specification of Cuneiform's semantics. The syntax featured conditionals. However, Booleans were encoded as lists, recycling the empty list as Boolean false and the non-empty list as Boolean true. Recursion was added later as a byproduct of formalization. However, static type checking was introduced only in Version 3.

The following script decompresses a zipped file and splits it into evenly sized partitions.

```
deftask unzip( <out( File )> : zip( File ) ) in bash *{
  unzip -d dir $zip
  out=`ls dir | awk '{print "dir/" $0}'`
}*

deftask split( <out( File )> : file( File ) ) in bash *{
  split -l 1024 $file txt
  out=txt*
}*

sotu = "sotu/stateoftheunion1790-2014.txt.zip";
fileLst = split( file: unzip( zip: sotu ) );

fileLst;
```

Version 3

The current version of Cuneiform's surface syntax, in comparison to earlier drafts, is

an attempt to close the gap to mainstream functional programming languages. It features a simple, statically checked type system and introduces records in addition to lists as a second type of compound data structure. Booleans are a separate base data type.

The following script untars a file resulting in a file list.

```
def untar( tar : File ) -> <fileLst : [File]>
in Bash *{
  tar xf $tar
  fileLst=`tar tf $tar`
}*

let hg38Tar : File =
  'hg38/hg38.tar';

let <fileLst = faLst : [File]> =
  untar( tar = hg38Tar );

faLst;
```

References

1. "Joergen7/Cuneiform" (https://github.com/joergen7/cuneiform). *GitHub*. 14 October 2021.
2. Brandt, Jörgen; Bux, Marc N.; Leser, Ulf (2015). "Cuneiform: A functional language for large scale scientific data analysis" (http://ceur-ws.org/Vol-1330/paper-03.pdf) (PDF). *Proceedings of the Workshops of the EDBT/ICDT*. **1330**: 17–26.
3. "Scalable Multi-Language Data Analysis on Beam: The Cuneiform Experience by Jörgen Brandt" (https://web.archive.org/web/20161002222350/http://beta.erlangcentral.org/videos/scalable-multi-language-data-analysis-on-beam-the-cuneiform-experience-by-jorgen-brandt/#.WBLlE2hNzIU). *Erlang Central*. Archived from the original (http://beta.erlangcentral.org/videos/scalable-multi-language-data-analysis-on-beam-the-cuneiform-experience-by-jorgen-brandt/#.WBLlE2hNzIU) on 2 October 2016. Retrieved 28 October 2016.
4. Bux, Marc; Brandt, Jörgen; Lipka, Carsten; Hakimzadeh, Kamal; Dowling, Jim; Leser, Ulf (2015). "SAASFEE: scalable scientific workflow execution engine" (http://www.vldb.org/pvldb/vol8/p1892-bux.pdf) (PDF). *Proceedings of the VLDB Endowment*. **8** (12): 1892–1895. doi:10.14778/2824032.2824094 (https://doi.org/10.14778%2F2824032.2824094).
5. Bessani, Alysson; Brandt, Jörgen; Bux, Marc; Cogo, Vinicius; Dimitrova, Lora; Dowling, Jim; Gholami, Ali; Hakimzadeh, Kamal; Hummel, Michael; Ismail, Mahmoud; Laure, Erwin; Leser, Ulf; Litton, Jan-Eric; Martinez, Roxanna; Niazi, Salman; Reichel, Jane; Zimmermann, Karin (2015). "Biobankcloud: a platform for the secure storage, sharing, and processing of large biomedical data sets" (http://www.di.fc.ul.pt/~bessani/publications/dmah15-bbc.pdf) (PDF). *The First International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2015)*.
6. "Scalable Multi-Language Data Analysis on Beam: The Cuneiform Experience" (http://www.erlang-factory.com/euc2016/jorgen-brandt). *Erlang-factory.com*. Retrieved 28 October 2016.

7. Kelly, Peter M.; Coddington, Paul D.; Wendelborn, Andrew L. (2009). "Lambda calculus as a workflow model". *Concurrency and Computation: Practice and Experience*. **21** (16): 1999–2017. doi:10.1002/cpe.1448 (https://doi.org/10.1002%2Fcpe.1448). S2CID 10833434 (https://api.semanticscholar.org/CorpusID:10833434).
8. Barseghian, Derik; Altintas, Ilkay; Jones, Matthew B.; Crawl, Daniel; Potter, Nathan; Gallagher, James; Cornillon, Peter; Schildhauer, Mark; Borer, Elizabeth T.; Seabloom, Eric W. (2010). "Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis" (https://escholarship.org/content/qt2q46n1tp/qt2q46n1tp.pdf?t=nivnuu) (PDF). *Ecological Informatics*. **5** (1): 42–50. doi:10.1016/j.ecoinf.2009.08.008 (https://doi.org/10.1016%2Fj.ecoinf.2009.08.008). S2CID 16392118 (https://api.semanticscholar.org/CorpusID:16392118).
9. Di Tommaso, Paolo; Chatzou, Maria; Floden, Evan W; Barja, Pablo Prieto; Palumbo, Emilio; Notredame, Cedric (2017). "Nextflow enables reproducible computational workflows". *Nature Biotechnology*. **35** (4): 316–319. doi:10.1038/nbt.3820 (https://doi.org/10.1038%2Fnbt.3820). PMID 28398311 (https://pubmed.ncbi.nlm.nih.gov/28398311). S2CID 9690740 (https://api.semanticscholar.org/CorpusID:9690740).
10. "A Functional Workflow Language Implementation in Erlang" (http://www.erlang-factory.com/static/upload/media/1448992381831050cuneiformberlinefl2015.pdf) (PDF). Retrieved 28 October 2016.
11. Brandt, Jörgen; Reisig, Wolfgang; Leser, Ulf (2017). "Computation semantics of the functional scientific workflow language Cuneiform". *Journal of Functional Programming*. **27**. doi:10.1017/S0956796817000119 (https://doi.org/10.1017%2FS0956796817000119). S2CID 6128299 (https://api.semanticscholar.org/CorpusID:6128299).

Retrieved from "https://en.wikipedia.org/w/index.php?title=Cuneiform_(programming_language)&oldid=1283887502"
