Cuneiform
A Functional Workflow Language Implementation in Erlang

Jörgen Brandt

Humboldt-Universität zu Berlin

2015-12-01
Cuneiform

A Functional Language for Large Scale Scientific Data Analysis
Cuneiform

A Functional Language for Large Scale Scientific Data Analysis

- Black-box operator model
  
  Pro: Operators can be any piece of software
Cuneiform
A Functional Language for Large Scale Scientific Data Analysis

- **Black-box operator model**
  Pro: Operators can be any piece of software

- **Black-box data model**
  Pro: Input and Output data can be anything
Cuneiform

A Functional Language for Large Scale Scientific Data Analysis

- **Black-box operator model**
  
  Pro: Operators can be any piece of software

- **Black-box data model**
  
  Pro: Input and Output data can be anything

- **Features of advanced workflow languages**
  
  Abstractions, lists, operations on lists, conditions
Cuneiform

A Functional Language for Large Scale Scientific Data Analysis

- **Black-box operator model**
  
  Pro: Operators can be any piece of software

- **Black-box data model**
  
  Pro: Input and Output data can be anything

- **Features of advanced workflow languages**
  
  Abstractions, lists, operations on lists, conditions

- **Light-weight Foreign Function Interface (FFI)**
  
  Wrapping in R, Matlab, Octave, Python, Lisp, Perl, Bash
Cuneiform

A Functional Language for Large Scale Scientific Data Analysis

- **Black-box operator model**
  Pro: Operators can be any piece of software

- **Black-box data model**
  Pro: Input and Output data can be anything

- **Features of advanced workflow languages**
  Abstractions, lists, operations on lists, conditions

- **Light-weight Foreign Function Interface (FFI)**
  Wrapping in R, Matlab, Octave, Python, Lisp, Perl, Bash

- **Automatic parallelization**
  Scalability with large data sets
“New hardware is increasingly parallel, so new programming languages must support concurrency or they will die.”

Joe Armstrong
“New Hardware”
DNA Sequencing is becoming cheap
Decentralized software development
Scientific Workflow Systems
Workflows as DAGs

- Scientific Workflows are DAGs
Scientific Workflow Systems

Workflows as DAGs

- Scientific Workflows are DAGs
- Nodes are tasks
Scientific Workflow Systems

Workflows as DAGs

- Scientific Workflows are DAGs
- Nodes are tasks
- Edges are data dependencies
Example: Galaxy Workflow System

Focus on

- Usability
Example: Galaxy Workflow System

Focus on

- Usability
- Integration of tools/libraries
Example: Galaxy Workflow System

Focus on

- Usability
- Integration of tools/libraries
- Systematic documentation
Focus on

- Usability
- Integration of tools/libraries
- Systematic documentation
- Reproducibility

Example: Galaxy Workflow System
The Next Generation Sequencing use case
The Next Generation Sequencing use case

![Diagram showing the Next Generation Sequencing process]

- **Reference alignment**
  - Reference sequence: AAG GTG GC
  - Read sequences:
    - AAGG G
    - GACA C
    - CCA A
    - GGG G
    - ACATA C
    - CGGC C
    - TAA A
    - GTCC C

- **Variant calling**
  - Variants:
    - AAGG G
    - GACA C
    - CCA A
    - GGG G
    - ACATA C
    - CGGC C
    - TAA A
    - GTCC C

- **Variant characterization**
  - Characterization:
    - AAGG G
    - GACA C
    - CCA A
    - GGG G
    - ACATA C
    - CGGC C
    - TAA A
    - GTCC C

**Moore's law**

- **Cost per Mbp of DNA sequenced**
  - April 02: $100,000
  - September 02: $50,000
  - January 04: $25,000
  - May 05: $12,500
  - October 05: $6,250
  - December 05: $3,125
  - April 06: $1,562.50
  - August 06: $781.25
  - December 06: $390.63
  - June 07: $195.31
The Next Generation Sequencing use case

Jörgen Brandt (HU Berlin)
The Next Generation Sequencing use case
Desired Features in a Language

Is there a language that is . . .

Like a workflow language
So we can integrate all the tools
Like MapReduce
So we can derive parallelism and distribute the work
Like a functional programming language
So we can write arbitrary programs using lists and operations on lists
Desired Features in a Language

Is there a language that is . . .

- Like a workflow language
  
  So we can integrate all the tools
Desired Features in a Language

Is there a language that is . . .

- Like a workflow language
  So we can integrate all the tools

- Like MapReduce
  So we can derive parallelism and distribute the work
Is there a language that is . . .

- Like a workflow language
  So we can integrate all the tools

- Like MapReduce
  So we can derive parallelism and distribute the work

- Like a functional programming language
  So we can write arbitrary programs using lists and operations on lists
deftask gunzip( out( File ) : gz( File ) ) in bash *
  gzip -c -d $gz > $out
}*

Jørgen Brandt  (HU Berlin)
deftask gunzip( out( File ) : gz( File ) )

in bash *

  gzip -c -d $gz > $out

}*

gunzip(  
gz: 'myarchive1.gz'
);

Jørgen Brandt (HU Berlin)
deftask gunzip( out( File ) : gz( File ) ) in bash *
  gzip -c -d $gz > $out
}*

gunzip(  
  gz: 'myarchive1.gz' 'myarchive2.gz'  
);
**Cuneiform example**

```
deftask gunzip( out( File ) : gz( File ) ) in bash {
  gzip -c -d $gz > $out
}*

gunzip(
  gz: 'myarchive1.gz' 'myarchive2.gz' 'myarchive3.gz'
);
```
Workflow Implementations Available

Available example workflows:

- Variant calling
  https://www.github.com/joergen7/variant-call

- Methylation
  https://www.github.com/joergen7/methylation

- RNA-Seq
  https://www.github.com/joergen7/rna-seq

- etc (ChIP-Seq, miRNA detection, consensus prediction, ...)

- samtools-faidx
- fastq-dump
- cufflinks
- cummerbund
- tophat-single
- cuffdiff
- cuffmerge
Workflow Implementations Available

Available example workflows:

- Variant calling
  https://www.github.com/joergen7/variant-call

- Methylation
  https://www.github.com/joergen7/methylation
Available example workflows:

- Variant calling
  https://www.github.com/joergen7/variant-call

- Methylation
  https://www.github.com/joergen7/methylation

- RNA-Seq https://www.github.com/joergen7/rna-seq
Workflow Implementations Available

Available example workflows:

- Variant calling
  https://www.github.com/joergen7/variant-call
- Methylation
  https://www.github.com/joergen7/methylation
- RNA-Seq https://www.github.com/joergen7/rna-seq
- etc (ChIP-Seq, miRNA detection, consensus prediction, ...)

Jørgen Brandt (HU Berlin)
Cuneiform
2015-12-01
Cuneiform Operational Semantics
Current design in Java implementation:

Program → Scanner/Parser → Parse tree → Transcription Visitor → Abstract Program → Interpreter → Result
Designated design but never implemented:
Designated design in Erlang:

- Program Interpretation in Cuneiform
- Program
- Scanner/Parser
- Parse tree
- Transcription Visitor
- Abstract Program
- Interpreter
- Result
- EBNF
- Parser Generator
  (ANTLR)
- Operational Semantics
- Program Extraction
  (?)
The eval Function

fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr
fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr

Expr The expression to be evaluated
fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr

Expr The expression to be evaluated
ρ Current scope
The eval Function

fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr

Expr The expression to be evaluated
ρ Current scope
GetFuture A function returning a future for a foreign task application
The eval Function

fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr

Expr The expression to be evaluated
ρ Current scope
GetFuture A function returning a future for a foreign task application
Global Task definitions
The eval Function

\[
\textbf{fun} \ \text{eval}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \rightarrow \text{Result when}
\]

\[
\begin{align*}
\text{Expr} & : \text{expr} \\
\rho & : \text{string} \Rightarrow \text{expr} \\
\text{GetFuture} & : \text{fun} \\
\text{Global} & : \text{string} \Rightarrow \text{lam} \\
\text{Fin} & : \text{id} \Rightarrow \text{expr} \\
\text{Result} & : \text{expr}
\end{align*}
\]

- \textbf{Expr} The expression to be evaluated
- \textbf{\rho} Current scope
- \textbf{GetFuture} A function returning a future for a foreign task application
- \textbf{Global} Task definitions
- \textbf{Fin} Results of foreign task applications
fun eval(Expr, ρ, GetFuture, Global, Fin) → Result when

Expr :: expr
ρ :: string => expr
GetFuture :: fun
Global :: string => lam
Fin :: id => expr
Result :: expr

Expr The expression to be evaluated
ρ Current scope
GetFuture A function returning a future for a foreign task application
Global Task definitions
Fin Results of foreign task applications
Result The result of evaluation (may contain futures)
\begin{align*}
\text{eval}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \rightarrow \\
\text{Next} = \text{step}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \\
\text{case} \text{Next} \text{ of} \\
\hspace{1em} \text{Expr} \rightarrow \text{Expr} \\
\hspace{1em} - \rightarrow \text{eval}(\text{Next}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \\
\text{end}
\end{align*}
Computational Semantics

\[ \text{eval}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \rightarrow \]
\[ \text{Next} = \text{step}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \]
\[ \textbf{case Next of} \]
\[ \text{Expr} \rightarrow \text{Expr} \]
\[ - \rightarrow \text{eval}(\text{Next}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \]
\[ \textbf{end} \]

- single step is computed
Computational Semantics

\[
\text{eval}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \rightarrow \\
\text{Next} = \text{step}(\text{Expr}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \\
\text{case Next of} \\
\text{Expr} \rightarrow \text{Expr} \\
\_ \rightarrow \text{eval}(\text{Next}, \rho, \text{GetFuture}, \text{Global}, \text{Fin}) \\
\text{end}
\]

- single step is computed
- If the step has no effect evaluation terminates
fun eval(Expr, ρ, GetFuture, Global, Fin) →
Next = step(Expr, ρ, GetFuture, Global, Fin)
case Next of
    Expr → Expr
    _ → eval(Next, ρ, GetFuture, Global, Fin)
end

- single step is computed
- If the step has no effect evaluation terminates
- Otherwise eval is called recursively
Choosing a language

- Functional Language

Languages Suitable for Operational Semantics

- Common Lisp
- Haskell
- ML
- Scala
- Erlang
Formalisms Suitable for Operational Semantics

Choosing a language

- Functional Language
- With Pattern Matching

Diagram:
- Common Lisp
- Haskell
- ML
- Scala
- Erlang
Formalisms Suitable for Operational Semantics

Choosing a language
- Functional Language
- With Pattern Matching
- Concurrency Orientation
Program Interpretation in Cuneiform

New design:

- **Program** → **Scanner/Parser** → **Abstract Program** → **Result**
  - EBNF
  - Parser Generator (Leex/Yecc)
  - Compiler (Erlang)
  - Operational Semantics
  - Interpreter

---

Jörgen Brandt  (HU Berlin)
New design:

Program Interpretation in Cuneiform

Parser Generator
(Leex/Yecc)

Scanner/
Parser

Abstract
Program

Interprete

Result

Compiler
(Erlang)

Operational
Semantics

EBNF
Fault Tolerance
Distributed applications can be complex in two independent ways:

- In the number of processes involved to compute one task: the more system components, the more likely one component fails.
- In the number of tasks contributing to a workflow: the more tasks, the more likely one task fails.
Distributed applications can be complex in two independent ways:

- in the number of processes involved to compute one task
  - the more system components the more likely one component fails

- in the number of tasks contributing to a workflow
  - the more tasks the more likely one task fails
Two types of complexity

Distributed applications can be complex in two independent ways:

- in the number of processes involved to compute one task, the more system components the more likely one component fails.
- in the number of tasks contributing to a workflow, the more tasks the more likely one task fails.
Two types of complexity

Distributed applications can be complex in two independent ways:

- in the number of processes involved to compute one task, the more system components the more likely one component fails.
- in the number of tasks contributing to a workflow, the more tasks the more likely one task fails.
Distributed applications can be complex in two independent ways:

- **in the number of processes involved to compute one task**: the more system components the more likely one component fails.

- **in the number of tasks contributing to a workflow**: the more tasks the more likely one task fails.
Distributed Application: Workflow System

- Query
- Query
- Query
- Cache

Jörgen Brandt (HU Berlin)
Distributed Application: Workflow System

- Query
- Cache
- Scheduler

Jürgen Brandt (HU Berlin)
Distributed Application: Workflow System

- Query
- Cache
- Scheduler
- FS Work
- FS Work
- FS Work

Jörgen Brandt (HU Berlin)
How to achieve fault tolerance when

Workflow systems are complex in both

- Number of processes involved in computing one task
- Number of tasks in one workflow
  so failures are likely

All components need to maintain state
so plain restarting of components is not enough

Restarting of workflow helps only if workflows are small and system
has few components
How to achieve fault tolerance when

- Workflow systems are complex in both
  - Number of processes involved in computing one task
  - Number of tasks in one workflow

so failures are likely
How to achieve fault tolerance when

- Workflow systems are complex in both
  - Number of processes involved in computing one task
  - Number of tasks in one workflow

so failures are likely

- All components need to maintain state
  so plain restarting of components is not enough
How to achieve fault tolerance when

- Workflow systems are complex in both
  - Number of processes involved in computing one task
  - Number of tasks in one workflow

  so failures are likely

- All components need to maintain state

  so plain restarting of components is not enough

- Restarting of workflow helps only if workflows are small and system has few components
Generic process behaviour

Golden path:
- $P_1$ sends request to $P_2$
Generic process behaviour

Golden path:
- $P_1$ sends request to $P_2$
- Asynchronously $P_1$ receives reply
$P_2$ may fail:

- $P_1$ sends request to $P_2$
- $P_2$ fails
$P_2$ may fail:

- $P_1$ sends request to $P_2$
- $P_1$ creates monitor on $P_2$
Generic process behaviour

$P_2$ may fail:

- $P_1$ sends request to $P_2$
- $P_1$ creates monitor on $P_2$
- $P_1$ memorizes request
Generic process behaviour

$P_2$ may fail:
- $P_1$ sends request to $P_2$
- $P_1$ creates monitor on $P_2$
- $P_1$ memorizes request
- $P_2$ fails
Generic process behaviour

$P_2$ may fail:
- $P_1$ sends request to $P_2$
- $P_1$ creates monitor on $P_2$
- $P_1$ memorizes request
- $P_2$ fails
- $P_2$ supervisor restarts $P_2$
$P_2$ may fail:
- $P_1$ sends request to $P_2$
- $P_1$ creates monitor on $P_2$
- $P_1$ memorizes request
- $P_2$ fails
- $P_2$ supervisor restarts $P_2$
- request is replayed to $P_2$
- monitor is recreated
$P_1$ may fail:
- $P_1$ sends request to $P_2$
- $P_1$ fails
$P_1$ may fail:
- $P_1$ sends request to $P_2$
- $P_2$ creates monitor on $P_1$
Generic process behaviour

\( P_1 \) may fail:
- \( P_1 \) sends request to \( P_2 \)
- \( P_2 \) creates monitor on \( P_1 \)
- \( P_1 \) fails
Generic process behaviour

$P_1$ may fail:

- $P_1$ sends request to $P_2$
- $P_2$ creates monitor on $P_1$
- $P_1$ fails
- request is canceled
- supervisor restarts $P_1$
Conclusion

- Cuneiform:
Conclusion

- Cuneiform:
  - Functional
- Cuneiform:
  - Functional
  - Integrate anything
Cuneiform:
- Functional
- Integrate anything
- Parallelism

# Conclusion
Conclusion

- **Cuneiform:**
  - Functional
  - Integrate anything
  - Parallelism

- Runs on Hadoop
Conclusion

- **Cuneiform**:  
  - Functional  
  - Integrate anything  
  - Parallelism  
- **Runs on Hadoop**  
- **Implementation in Erlang**: 

![Diagram](https://github.com/joergen7/cuneiform)
Conclusion

- **Cuneiform:**
  - Functional
  - Integrate anything
  - Parallelism

- **Runs on Hadoop**

- **Implementation in Erlang:**
  - Concise stateless semantics
Conclusion

- Cuneiform:
  - Functional
  - Integrate anything
  - Parallelism

- Runs on Hadoop

- Implementation in Erlang:
  - Concise stateless semantics
  - Fine-grained fault tolerance