Scalable Multi-Language Data Analysis on BEAM

The Erlang Experience

Jörgen Brandt

Humboldt-Universität zu Berlin

2016-09-08

Erlang User Conference 2016

About me

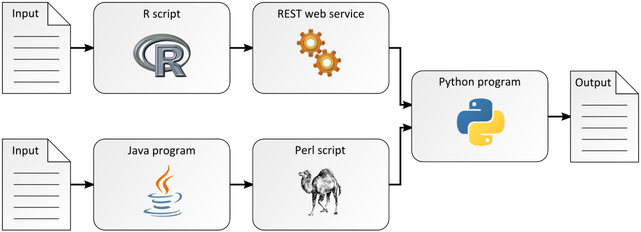
- PhD student at Humboldt (Berlin)
- Creator of Cuneiform workflow language
- Major area of application: Bioinformatics

Motivation

- Introduction to Cuneiform
- How to implement a large-scale data analysis PL
- How in Erlang?
- Why in Erlang?

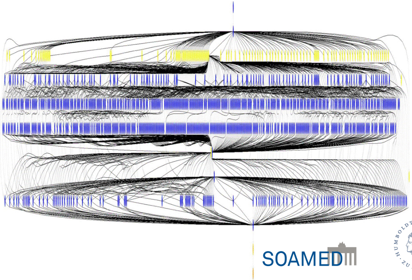
Scientific Workflow Systems

Scientific Workflow Systems

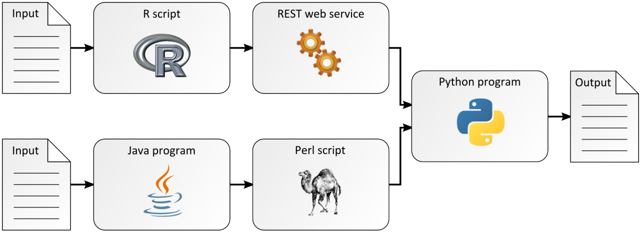


Workflows as DAGs

- Scientific Workflows are DAGs

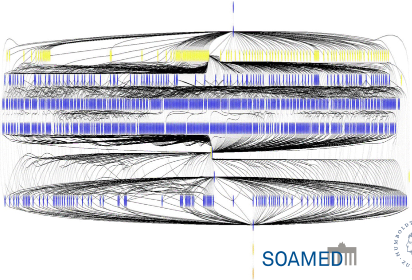


Scientific Workflow Systems

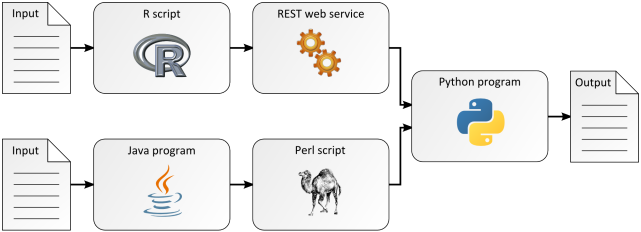


Workflows as DAGs

- Scientific Workflows are DAGs
- Nodes are tasks

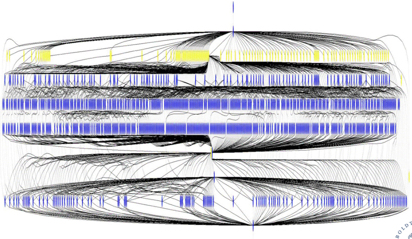


Scientific Workflow Systems



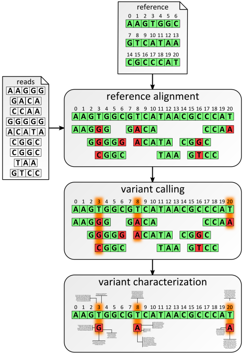
Workflows as DAGs

- Scientific Workflows are DAGs
- Nodes are tasks
- Edges are data dependencies

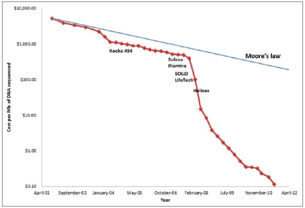


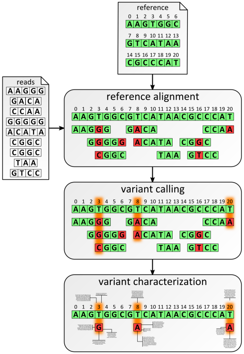
SOAMED

The Next Generation Sequencing use case

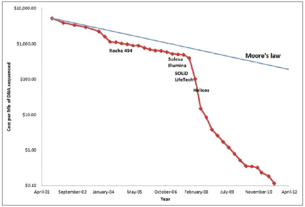


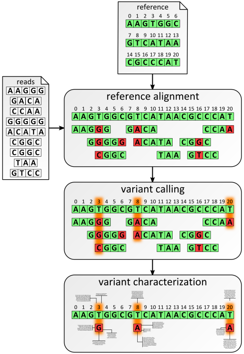
The Next Generation Sequencing use case



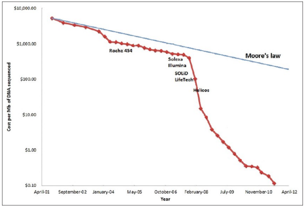


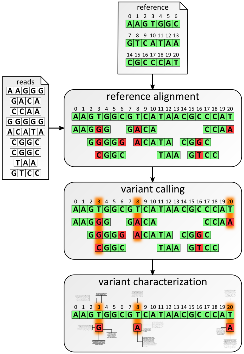
The Next Generation Sequencing use case

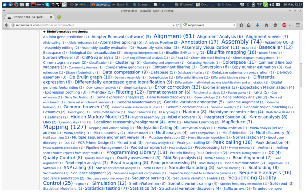




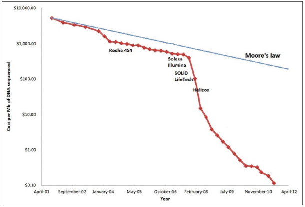
The Next Generation Sequencing use case

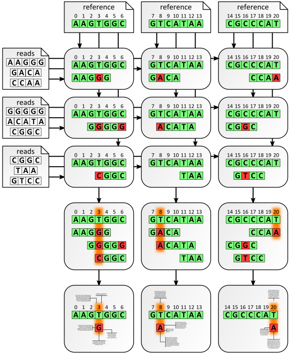


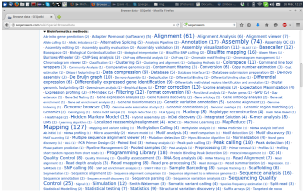




The Next Generation Sequencing use case







Cuneiform is ...

Functional Language for Large-Scale Data Analysis
implemented in Erlang

- Different from systems like Spark
  - Standalone syntax, no fluent API in Scala
  - Integration of foreign PLs
- Different from distributed workflow languages like Snakemake
  - Reduction of functional expression, no static dependency graph

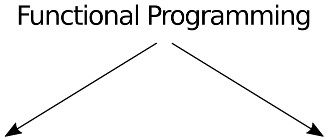
The main idea

Functional Programming

$+$

Foreign Language Interfacing

Functional Programming



- Very expressive
- Natural operations on lists (map, . . .)
- Natural iteration (recursion)

Gain

- General data analysis

Functional Programming

Functional Programming

- Very expressive
- Natural operations on lists (map, ...)
- Natural iteration (recursion)

Gain

- General data analysis

- Parallelize independent sub-expressions
- Lazy Evaluation

Gain

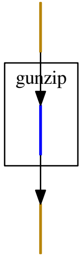
- Automatic Parallelism

Cuneiform Example

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

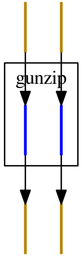
Cuneiform Example

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

gunzip(
  gz: 'myarchive1.gz'
);
```



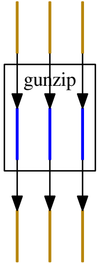
Cuneiform Example

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

gunzip(
  gz: 'myarchive1.gz' 'myarchive2.gz'
);
```



Cuneiform Example

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

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

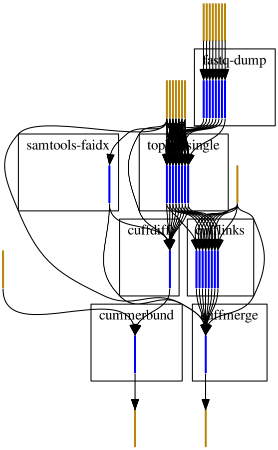


Workflow Implementations Available

`cuneiform-lang.org/examples`

Available example workflows:

- Variant Calling (Varscan)
- Methylation
- RNA-Seq
- Variant Calling (GATK)
- etc ...



Example: RNA-Seq

`cuneiform-lang.org/examples`

Cuneiform — About | Blog | Download | Examples | Code | Documentation

RNA-Seq

Feb 26, 2016

This workflow exemplifies the comparison of DNA expression levels in two experimental conditions from RNA-Seq data. It reimplements a study by Trapnell et al. 2012.

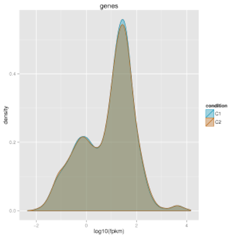
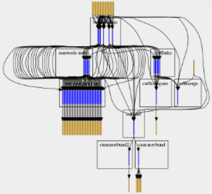


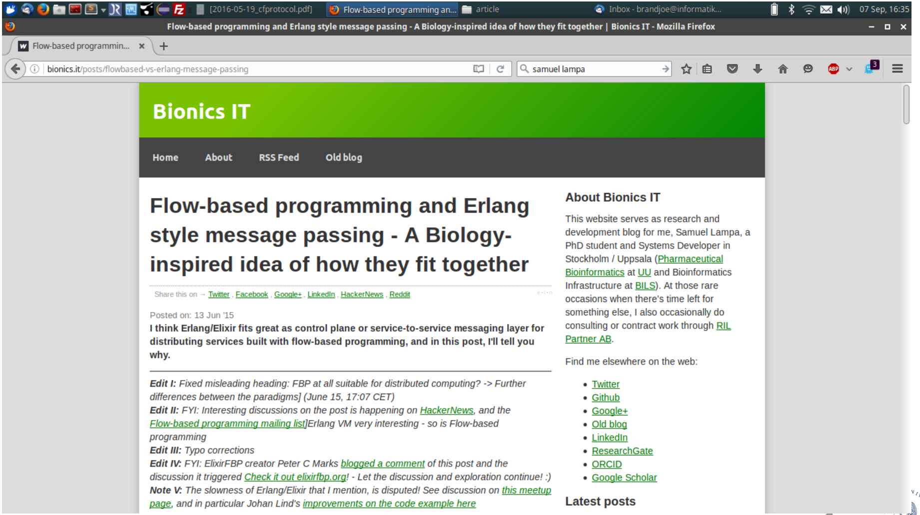
Figure 1: Probability density of expression levels of

RNA-Seq



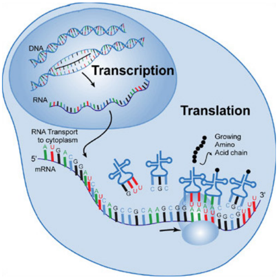
Cookbook	https://github.com/joergen7/rna-seq
Script	Cuneiform script
Publication	Trapnell et al. 2012
Input data	Drosophila reference genome SAMtools reference index Bowtie index FastQ samples (2 conditions)

Biology-Inspired Analogy



Biology-Inspired Analogy

- Large-scale data analysis systems as 2-tier system:
  - Algorithm-level (fast, local)
  - Workflow-level (organizational, distributed)
- Analogy to the cell:
  - DNA transcription (fast, local)
  - Cell-to-cell signaling (organizational, distributed)



Erlang good fit for organizational part

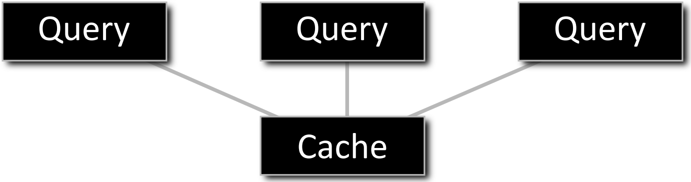
Distributed Workflow System Architecture

Query

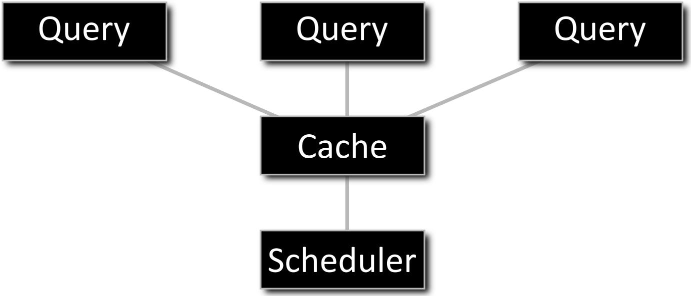
Query

Query

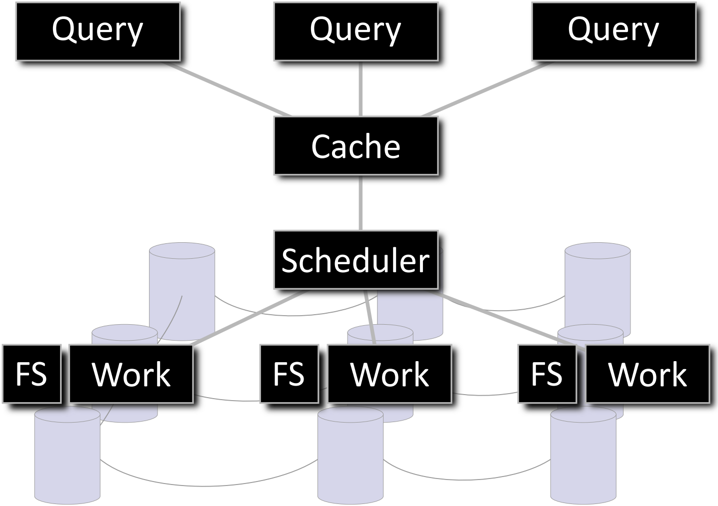
Distributed Workflow System Architecture



Distributed Workflow System Architecture



Distributed Workflow System Architecture



Two Modeling Challenges (i)

Interpreter

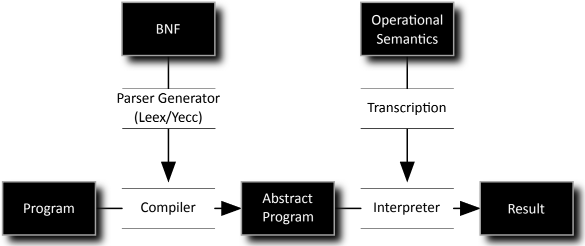
- Reduction of query expression

Execution Environment

- Distributed System

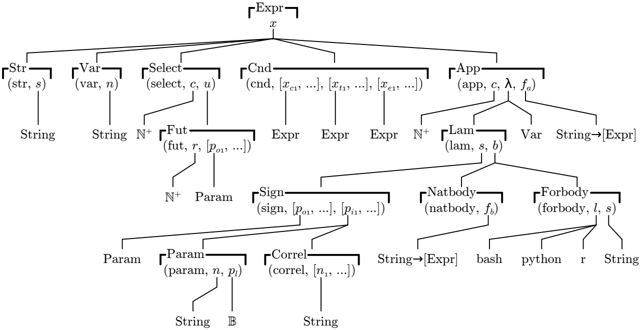
How to model programming languages?

Modeling the Cuneiform Interpreter



- Parser is generated from BNF
- Interpreter is transcribed from Operational Semantics

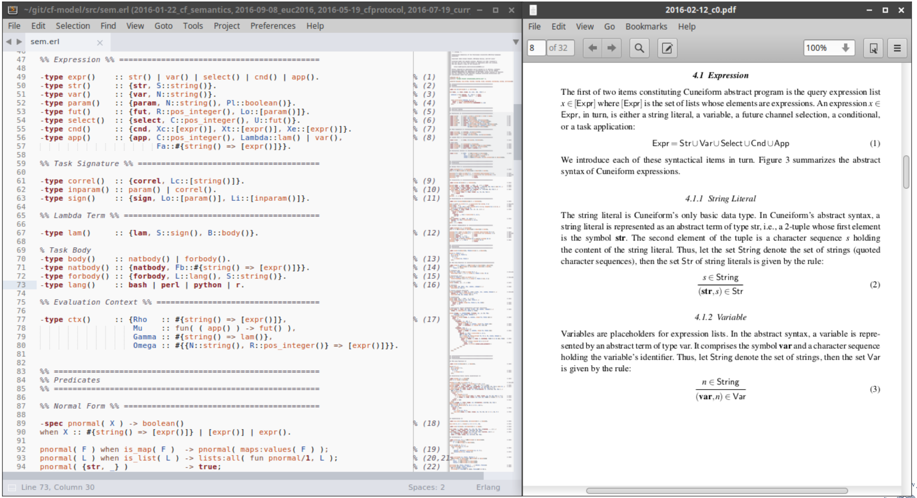
An expression in Cuneiform is . . .



- Expressions can contain other expressions
- Semantics define how expressions are reduced

$$x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x^*$$

Implementing an Operational Semantics in Erlang



~/git/cf-model/src/sem.erl (2016-01-22_cf_semantics, 2016-09-08_euc2016, 2016-05-19_cfprotocol, 2016-07-19_curr

File Edit Selection Find View Goto Tools Project Preferences Help

sem.erl

```erlang
%% Expression %% ==========================================

-type expr()    :: str() | var() | select() | cnd() | app().                 % (1)
-type str()     :: {str, S::string()}.                                       % (2)
-type var()     :: {var, N::string()}.                                       % (3)
-type param()   :: {param, N::string(), Pl::boolean()}.                      % (4)
-type fut()     :: {fut, R::pos_integer(), Lo::[param()]}.                   % (5)
-type select()  :: {select, C::pos_integer(), U::fut()}.                     % (6)
-type cnd()     :: {cnd, Xc::[expr()], Xt::[expr()], Xe::[expr()]}.          % (7)
-type app()     :: {app, C::pos_integer(), Lambda::lam() | var(),            % (8)
                         Fa::#{string() => [expr()]}}.

%% Task Signature %% ======================================

-type correl()  :: {correl, Lc::[string()]}.                                 % (9)
-type inparam() :: param() | correl().                                       % (10)
-type sign()    :: {sign, Lo::[param()], Li::[inparam()]}.                   % (11)

%% Lambda Term %% =========================================

-type lam()     :: {lam, S::sign(), B::body()}.                              % (12)

% Task Body
-type body()    :: natbody() | forbody().                                    % (13)
-type natbody() :: {natbody, Fb::#{string() => [expr()]}}.                   % (14)
-type forbody() :: {forbody, L::lang(), S::string()}.                        % (15)
-type lang()    :: bash | perl | python | r.                                 % (16)

%% Evaluation Context %% ==================================

-type ctx()     :: {Rho   :: #{string() => [expr()]},                        % (17)
                    Mu    :: fun( ( app() ) -> fut() ),
                    Gamma :: #{string() => lam()},
                    Omega :: #{{N::string(), R::pos_integer()} => [expr()]}}.


%% ========================================================
%% Predicates
%% ========================================================

%% Normal Form %% =========================================

-spec pnormal( X ) -> boolean()                                              % (18)
when X :: #{string() => [expr()]} | [expr()] | expr().

pnormal( F ) when is_map( F )  -> pnormal( maps:values( F ) );               % (19)
pnormal( L ) when is_list( L ) -> lists:all( fun pnormal/1, L );             % (20,21
pnormal( {str, _} )            -> true;                                      % (22)
```

Line 73, Column 30 Spaces: 2 Erlang

2016-02-12_c0.pdf

File Edit View Go Bookmarks Help

8 of 32 100%

4.1 Expression

The first of two items constituting Cuneiform abstract program is the query expression list $x \in [\mathsf{Expr}]$ where $[\mathsf{Expr}]$ is the set of lists whose elements are expressions. An expression $x \in \mathsf{Expr}$, in turn, is either a string literal, a variable, a future channel selection, a conditional, or a task application:

$$\mathsf{Expr} = \mathsf{Str} \cup \mathsf{Var} \cup \mathsf{Select} \cup \mathsf{Cnd} \cup \mathsf{App} \qquad (1)$$

We introduce each of these syntactical items in turn. Figure 3 summarizes the abstract syntax of Cuneiform expressions.

4.1.1 String Literal

The string literal is Cuneiform's only basic data type. In Cuneiform's abstract syntax, a string literal is represented as an abstract term of type str, i.e., a 2-tuple whose first element is the symbol **str**. The second element of the tuple is a character sequence s holding the content of the string literal. Thus, let the set String denote the set of strings (quoted character sequences), then the set Str of string literals is given by the rule:

$$\frac{s \in \mathsf{String}}{(\mathbf{str}, s) \in \mathsf{Str}} \qquad (2)$$

4.1.2 Variable

Variables are placeholders for expression lists. In the abstract syntax, a variable is represented by an abstract term of type var. It comprises the symbol **var** and a character sequence holding the variable's identifier. Thus, let String denote the set of strings, then the set Var is given by the rule:

$$\frac{n \in \mathsf{String}}{(\mathbf{var}, n) \in \mathsf{Var}} \qquad (3)$$

SOAMED

Two Modeling Challenges (ii)

Interpreter

- Reduction of query expression

Execution Environment

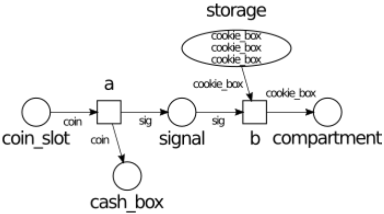
- Distributed System

How to model distributed systems?

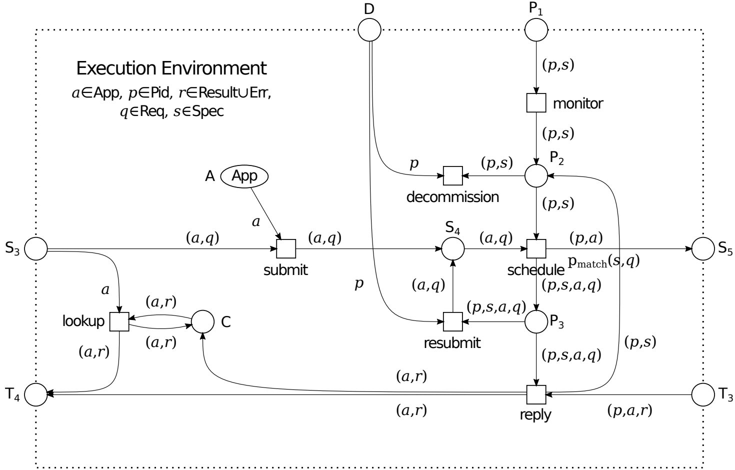
How to Model Distributed Systems

Petri Nets

- Mature theory
  - Liveness
  - Invariants
  - Traps/Cotraps
  - ...
- Local decision/synchronization
- Parallel execution of independent transitions



Modeling the Cuneiform Execution Environment



How to do Petri Nets in Erlang

The pnet application - Mozilla Firefox

cuneiform-lang.org/man/gen_pnet/index.html

Modules

- cvm1
- cvm2
- gen_pnet

Overview

A generic Petri net OTP library

Copyright © 2016 Jörgen Brandt

Version: 0.1.0

Authors: Jörgen Brandt (`brandjoe@hu-berlin.de`).

References

- Source code hosted at GitHub.

See also: cvm2, gen_pnet.

Some applications exhibit behavioral patterns that lend themselves to Petri Nets. The major advantage of modeling applications with Petri Nets is that they provide a natural view on the concurrent behavior of an application. This is achieved by making explicit the preconditions for an operation to be carried out while leaving implicit how and when an operation is triggered and what other operations might run in parallel.

This OTP library is a framework for programming with Petri Nets. It implements a very general form of Petri Nets: Colored Petri Nets (CPN). I.e., tokens may not only be markers but can be any conceivable data structure. Furthermore, a place can hold any number of tokens not just one.

While many simulation libraries only mimic the concurrent behavior of Petri Nets, the `gen_pnet` library allows the definition of Nets with an arbitrary number of transitions competing for a place's tokens neither imposing order in the form of an overarching loop nor otherwise constraining parallelism.

Quick Start

This Quick Start section provides an overview about how Petri nets are started, queried, and manipulated with the `gen_pnet` module. We demonstrate the module's API in terms of a cookie vending machine implemented in the `cvm2` module which is also part of this code repository. Then, we have a look at how the callback functions of the cookie vending machine are implemented.

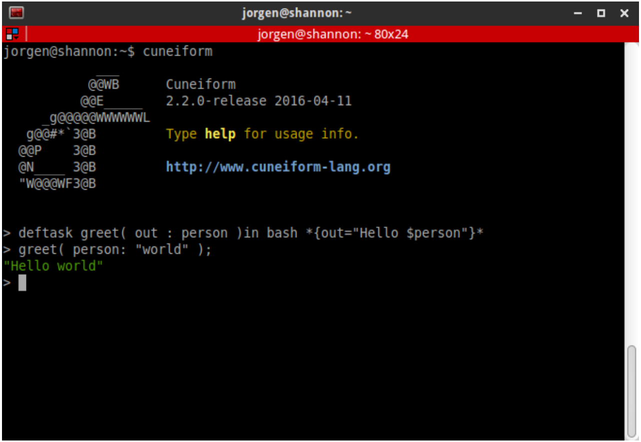
Flow-based programming revisited

- System languages for heavy lifting
- Large-scale data analysis is hard
  - Because programming languages are hard
  - Because distributed systems are hard
- Erlang is good fit
  - Because FP is already close to operational semantics style notation
  - Because Erlang process model already close to Petri Nets

SOAMED

REPL

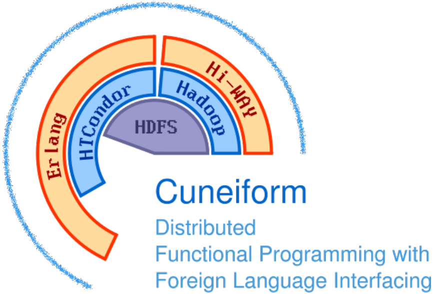
cuneiform-lang.org



SOAMED

Conclusion

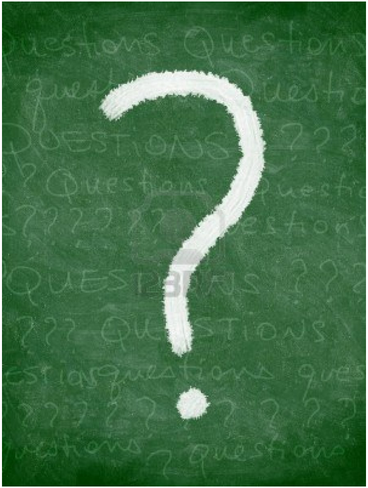
- Functional Programming + Foreign Languages
- Distributed Execution
  - Local Multicore
  - HTCondor
  - Hadoop
- Automatic Parallelization
- Expressive data analysis workflows
- Foreign Language Interface
  - Bash
  - Perl
  - ...



cuneiform-lang.org

SOAMED

Questions?



Questions?