Synthesis of Functional Programs Using Answer Set Programming

James Rodden

Functional Program Synthesis

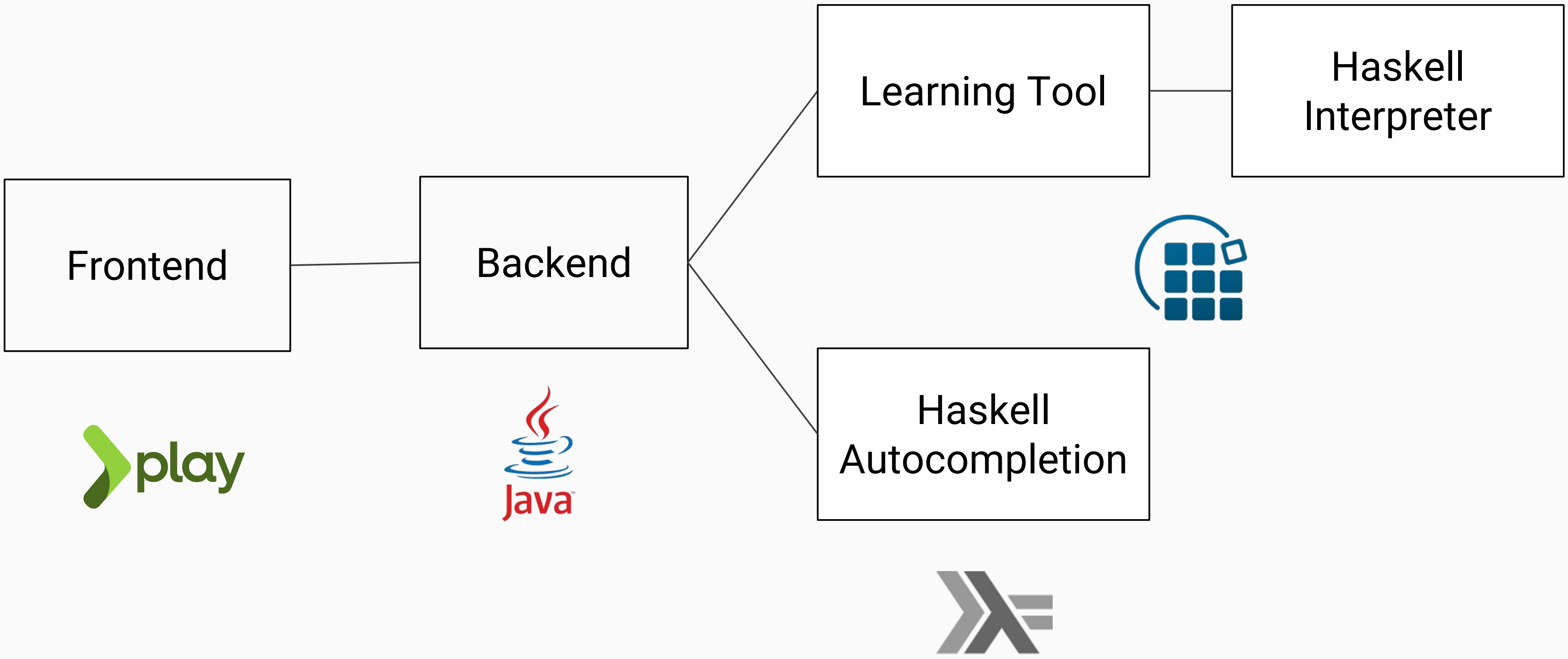
```
fac 0 = 1
fac 1 = 1
fac 2 = 2
fac 3 = 6
```

```
fac n
    | n == 0 = 1
    | otherwise = n * fac (n - 1)
```

A new approach

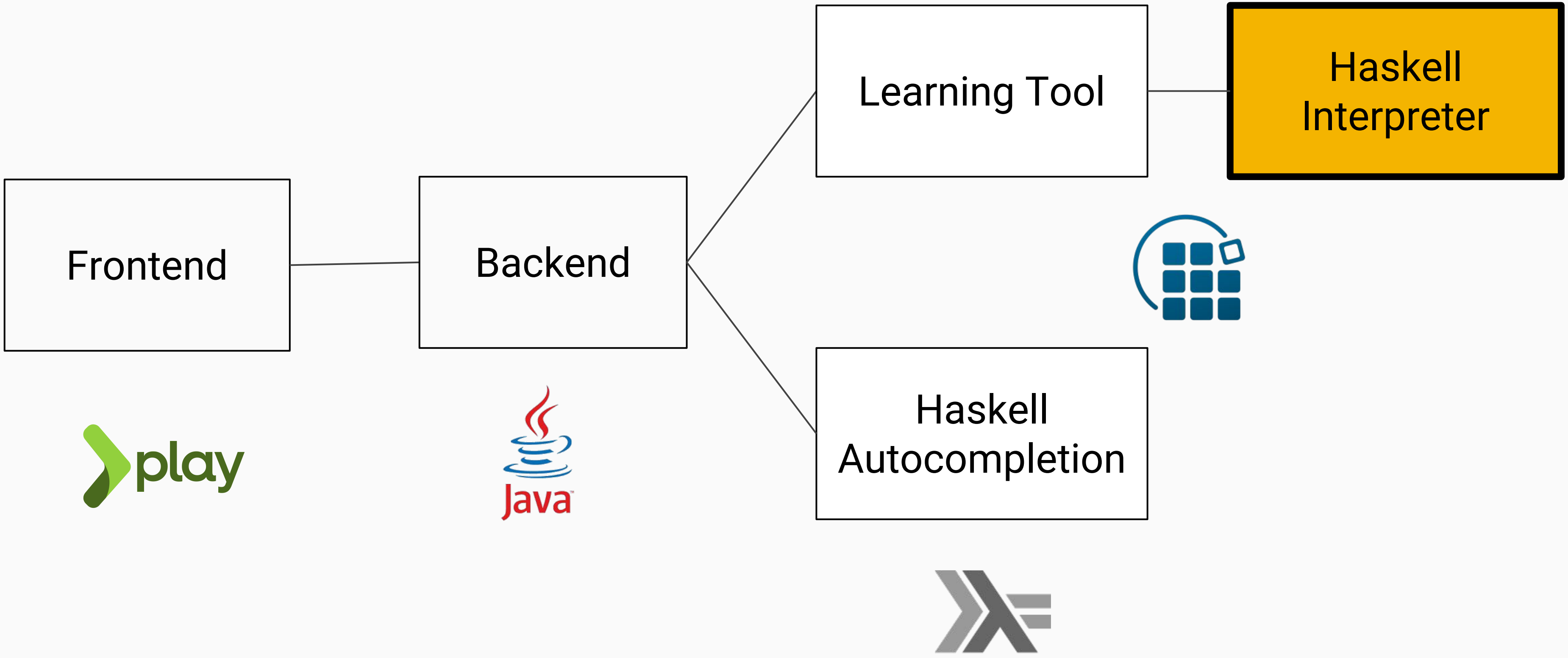
- Usage of ASP
- Human Readability
- Illustrating Teaching

Project Structure



A Learning Overview

Project Structure



The Haskell Interpreter

```
rule(1, fac, N, 1).
rule(2, fac, N, mul(N, call(fac, sub(N, 1)))).

match_guard(fac, 1, N) :- N == 0.
match_guard(fac, 2, N).
```

```
fac n
    | n == 0 = 1
    | otherwise = n * fac (n - 1)
```

- Haskell programs written in ASP
- The output of program execution is in the Answer Set.
- Function input and output represented by predicates:

```
input(Expr)        output(Expr, Out)
```

Evaluating Rules

```
eval(Expr) :-
    input(call(F, Input)),
    rule(Index, F, Input, Expr),
    match(F, Index, Input).


eval(Expr) :- eval(call(F, Expr)).
eval(A) :- eval(mul(A, B)).
eval(B) :- eval(mul(A, B)).


input(call(F, Input)) :-
    eval(call(F, Expr)),
    value(Expr, Input).
```

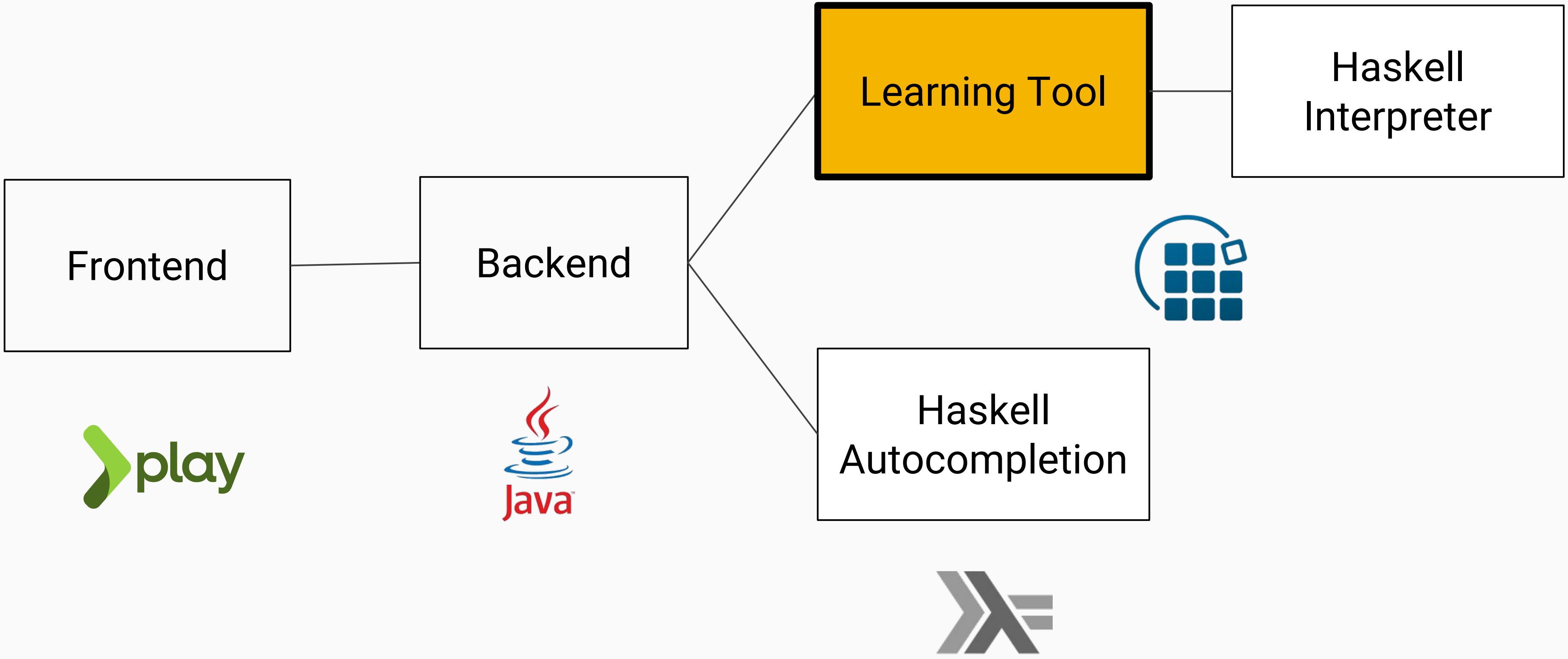
Evaluating Rules

```
output(call(F, N), Out) :-
    rule(Index, F, N, Expr),
    match(F, Index, N),
    value(Expr, Out).

value(mul(A, B), V1 * V2) :-
    eval(mul(A, B)),
    value(A, V1),
    value(B, V2).

value(call(F, Expr), Out) :-
    eval(call(F, Expr)),
    output(call(F, Input), Out),
    value(Expr, Input).
```

Project Structure



Skeleton Rules

```
rule(R, F, N, mul(N, N)) :- input(call(F, Args)), choose(R, 3).

rule(R, F, N, mul(C1, call(F, sub(N, C2)))) :- input(call(F, Args)), choose(R, 38, C1, C2).
```

- An enumeration over all possible function bodies
- Constants replaced with variables representing the range of possible values.

C1 * f (n - C2)

0 + f (n - 0)

0 + f (n - 1)

1 + f (n - 0)

1 + f (n - 1)

Learning Programs

```
1 {
choose(R, 1...4;15...20;25...32),
choose(R, 5...14;21...24;33..40, C0) : expr_const(C0)
} 1 :- num_rules(R).
```

```
example(call(f, 0), 1).
example(call(f, 1), 1).
example(call(f, 2), 2).
example(call(f, 3), 6).
```

```
choose(1, 1, 1).
choose(2, 37, 1).
```

```
rule(1, fac, N, 1).
rule(2, fac, N, mul(N, call(fac, sub(N, 1)))).
```

Where rules

```
rule(1, fac, N, 1).
rule(2, fac, N, mul(N, x1)).

where(x1, N, call(fac, x2)).
where(x2, N, sub(N, 1)).

match_guard(fac, 1, N) :- N == 0.
match_guard(fac, 2, N).
```

```
fac n
    | n == 0 = 1
    | otherwise = n * x1
      where x1 = fac x2
            where x2 = n - 1
```

Evaluating Where Rules

```
value(X, V) :- eval(X), where(X, Args, Expr), value(Expr, V).
```

```
value(x1, 6).        value(x2, 3).
value(x1, 2).        value(x2, 2).
value(x1, 1).        value(x2, 1).
```

```
value_with(X, V, Args) :- eval_with(X, Args), where(X, Args, Expr), value_with(Expr, V, Args).
```

```
value_with(x1, 6, 4).        value_with(x2, 3, 4).
value_with(x1, 2, 3).        value_with(x2, 2, 3).
value_with(x1, 1, 2).        value_with(x2, 1, 2).
```

Examples of Learning

Tail Recursive Factorial

```
example(call(f, (0, 1)), 1).
example(call(f, (1, 1)), 1).
example(call(f, (2, 1)), 2).
example(call(f, (3, 1)), 6).
example(call(f, (2, 3)), 6).
```

```
fac x y
        | x == 0 = y
        | otherwise = fac x0 x1
        where x0 = x - 1
              where x1 = x * y
```

Greatest Common Divisor

```
example(call(gcd, (1, 1)), 1).
example(call(gcd, (2, 1)), 1).
example(call(gcd, (4, 3)), 1).
example(call(gcd, (3, 6)), 3).
example(call(gcd, (9, 6)), 3).
example(call(gcd, (4, 7)), 1).
example(call(gcd, (9, 3)), 3).
```

```
gcd x y
        | y == 0 = x
        | x > y = gcd y x0
        | otherwise = gcd x1 x
        where x0 = x - y
              where x1 = y - x
```

Example Performance

Function	Performance
Factorial	**Time : 1.640** Prepare : 0.060 Prepro. : 0.020 Solving : 1.560
Tail Recursive Factorial	**Time : 419.860** Prepare : 2.060 Prepro. : 0.840 Solving : 416.960
Greatest Common Divisor	**Time : 1546.820** Prepare : 1.940 Prepro. : 0.980 Solving : 1543.900

A Constraint Based Approach

- Maintain a set of Equality Constraints

$$(2 * 2) + 1 = 5$$
$$(2 * 2) = 4$$
$$2 = 2$$

- Broken if any simple contradiction found

$$(2 * 2) + 1 = 6$$
$$(2 * 2) = 5$$

A Constraint Based Approach

Maintaining Constraints

```
eq(In, Out) :- example(In, Out).

:- eq(N1, N2), const_range(N1), const_range(N2), N1 != N2.

eq(A, Val - B) :- eq(add(A, B), Val), const_number(B),  B <= Val.

eq(Expr, Val) :- rule(Index, F, Arg_Expr, Expr), match(F, Index, Arg_Expr), eq(call(F, Arg_Expr), Val).
```

Handling Termination

```
:- example(In, Out), not terminates(In).

terminates(Expr) :- next_step(Expr, Val), const(Val).
terminates(Expr) :- next_step(Expr, Term_Expr), terminates(Term_Expr).

next_step(add(A, B), B) :- is_next_step(add(A, B)), const_number(A).
```

Skeleton Rule Optimisations

```
rule(R, F, N, add(N, N)) :- input(call(F, Args)), choose(R, 2).

rule(R, F, N, mul(N, 2)) :- input(call(F, Args)), choose(R, 4, 2).
```

```
rule(R, F, N, (N + N)) :- input(call(F, Args)), choose(R, 2).

rule(R, F, N, (2 * N)) :- input(call(F, Args)), choose(R, 4, 2).
```

Further Examples

Tail Recursive Factorial

```
example(call(f, (0, 1)), 1).
example(call(f, (1, 1)), 1).
example(call(f, (2, 1)), 2).
example(call(f, (3, 1)), 6).
example(call(f, (2, 3)), 6).
```

```
fac x y
      | x == 0 = y
      | otherwise = fac x0 x1
      where x0 = x - 1
            where x1 = x * y
```

```
fac x y
      | x == 0 = y
      | otherwise = fac (x - 1) (x * y)
```

Greatest Common Divisor

```
example(call(gcd, (1, 1)), 1).
example(call(gcd, (2, 1)), 1).
example(call(gcd, (4, 3)), 1).
example(call(gcd, (3, 6)), 3).
example(call(gcd, (9, 6)), 3).
example(call(gcd, (4, 7)), 1).
example(call(gcd, (9, 3)), 3).
```

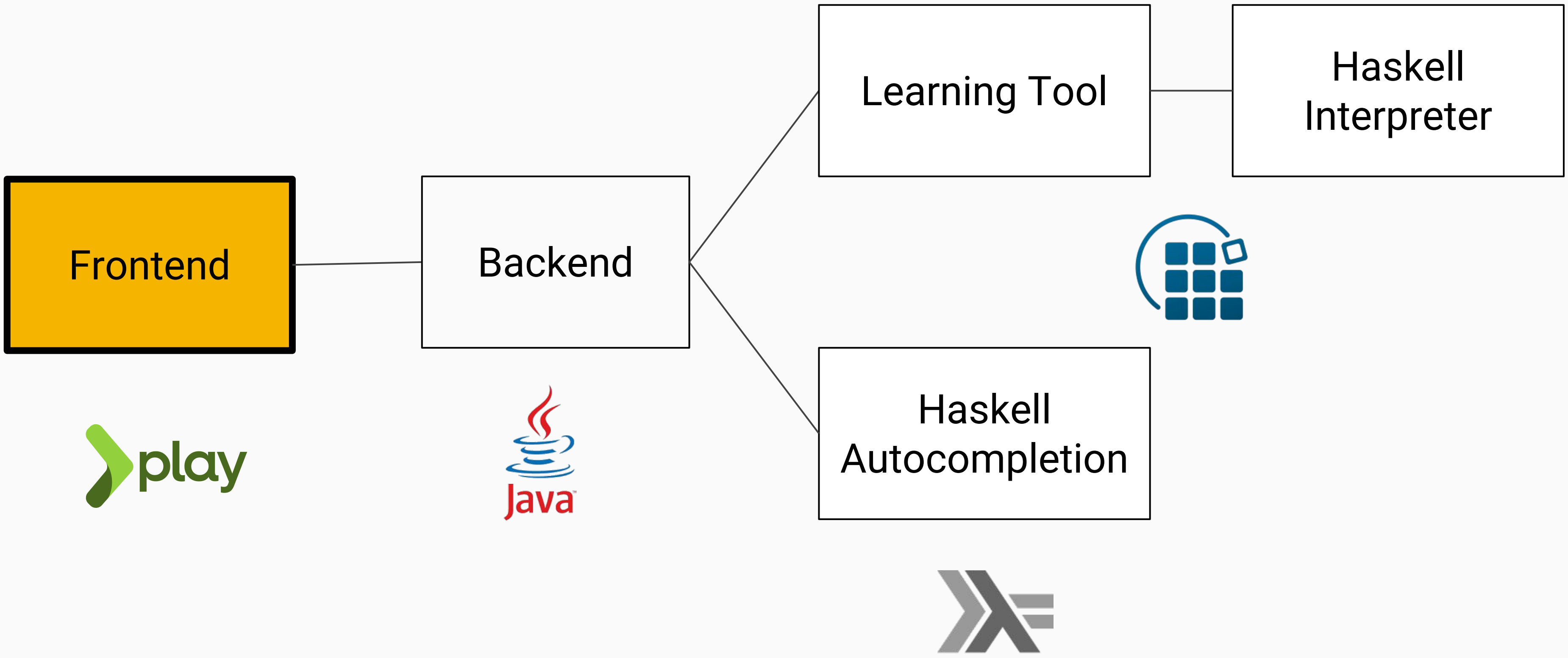
```
gcd x y
      | y == 0 = x
      | x > y = gcd y x0
      | otherwise = gcd x1 x
      where x0 = x - y
            where x1 = y - x
```

```
gcd x y
      | x == y = x
      | x > y = gcd (x - y) y
      | x < y = gcd y x
```

A Performance Comparison

Function	Interpreter Performance	Constraint Performance
Factorial	**Time : 1.640** Prepare : 0.060 Prepro. : 0.020 Solving : 1.560	**Time : 1.240** Prepare : 0.200 Prepro. : 0.140 Solving : 0.900
Tail Recursive Factorial	**Time : 419.860** Prepare : 2.060 Prepro. : 0.840 Solving : 416.960	**Time : 173.900** Prepare : 5.540 Prepro. : 1.440 Solving : 166.920
Greatest Common Divisor	**Time : 1546.820** Prepare : 1.940 Prepro. : 0.980 Solving : 1543.900	**Time : 1599.420** Prepare : 11.720 Prepro. : 2.100 Solving : 1585.600

Project Structure



The Iterative Learning Workflow

Future Work

- Type Usage
- Lists
- Strings
- Haskell Inbuilt Functions
- Parallel Learning

- Advanced Skeleton Rule Generation
- Improved Optimisation Statements
- Supporting Multiple Function Calls

Summary of Contributions

- I have presented a system which, given a set of input / output examples, learns functional programs.

- I extend the system with two alternate approaches to learning, one that is shown to have better performance and one that is shown to be more expressive.

- This system is easily usable through use of a user interface specialised towards helping new programmers learn Haskell

MagicHaskeller Comparison

Function	My Solution	MagicHaskeller Solution
Factorial	f n | n == 0 = 1 | otherwise = n * f (n - 1)	f = (\a -> product [1..a])
Greatest Common Divisor	gcd x y | x == y = x | x > y = gcd (x - y) y | x < y = gcd y x	f = (flip gcd)
Fibonacci	fib x | x == 1 = x | x == 2 = x - 1 | otherwise = x0 + x2 where x0 = fib x1 where x1 = x - 1 where x2 = fib x3 where x3 = x - 2	No Result!

MagicHaskeller Comparison

Function	My Performance	MagicHaskeller Performance
Factorial	1.24s	2.168s
Power of Two	5.58s	2.147s
Tail Recursive Factorial	173.9s	133.65s
Greatest Common Divisor	1599.4s	2.315s