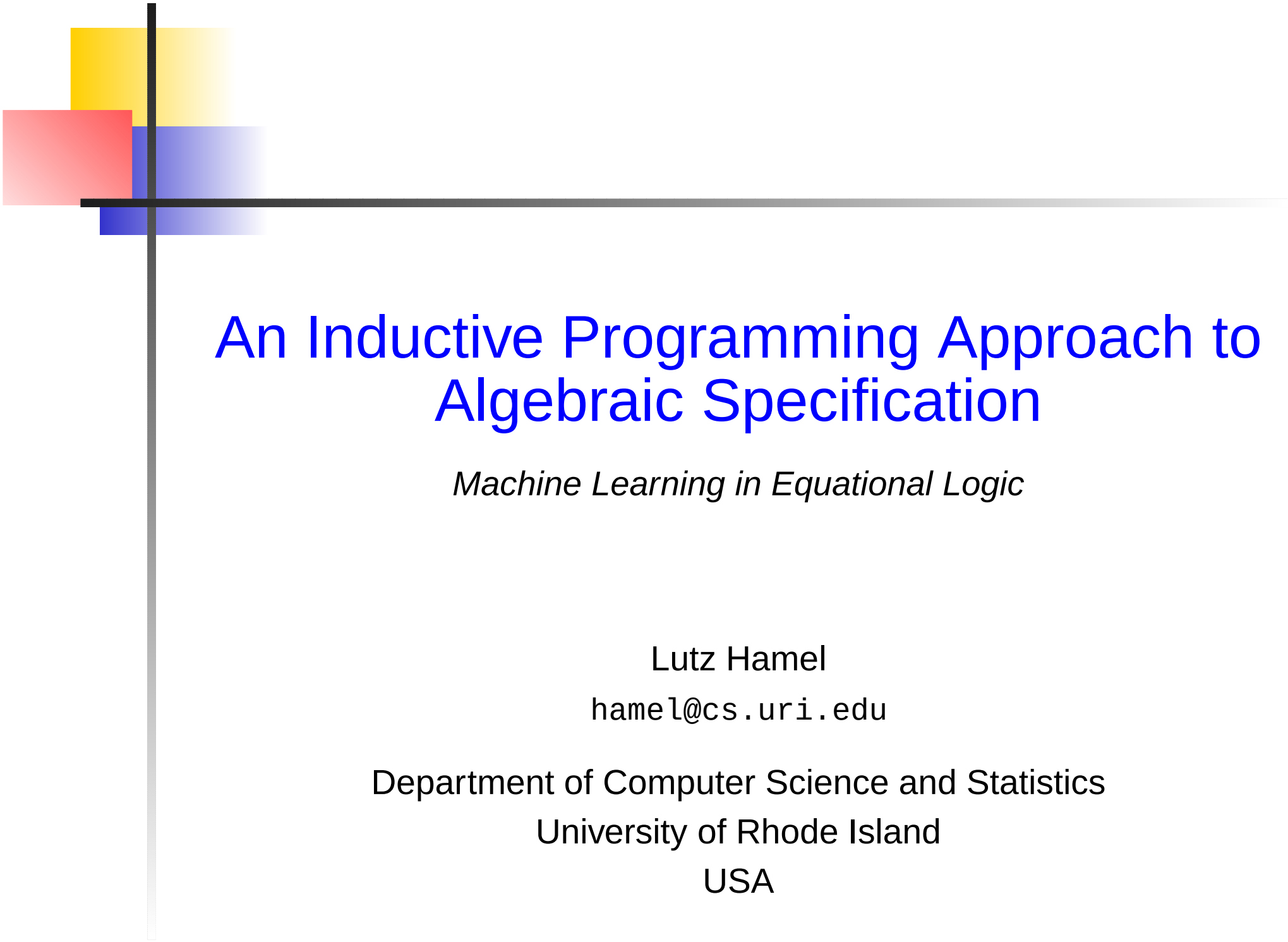


An Inductive Programming Approach to Algebraic Specification

Machine Learning in Equational Logic

Lutz Hamel

hamel@cs.uri.edu

Department of Computer Science and Statistics

University of Rhode Island

USA

Motivation

Build and Test

- Use requirements to construct a theory.
- Use requirements to construct test cases.
- Validate theory with the test cases.

Inductive Programming

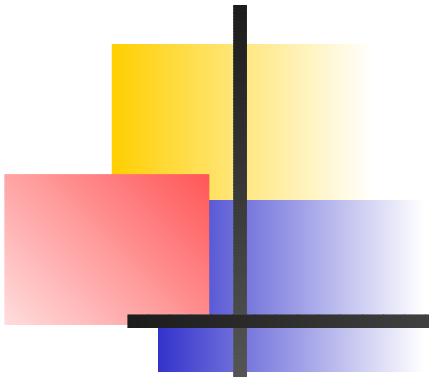
- Use requirements to construct test cases.
- Induce a theory on the test cases.

$\Rightarrow$ Inductive programming seems promising in that it puts less cognitive burden on the engineer/developer.

Equational Logic

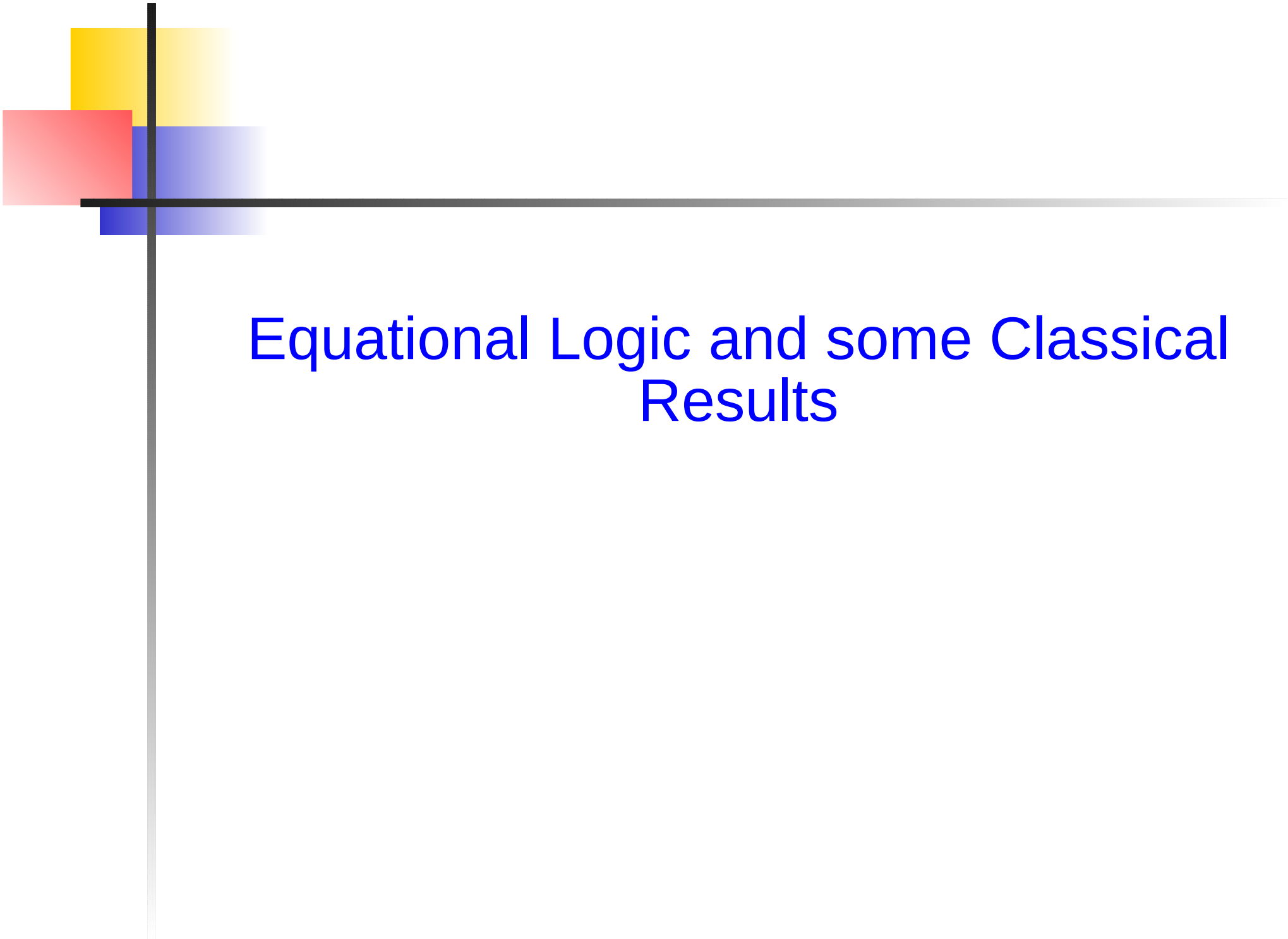
Why equational logic?

- Intuitive - substituting equals for equals.
- Simple proof theory.
- Support for automated deduction.
- Straight forward model theory.



Overview

- Equational logic and some classical results
- Machine learning - an algebraic perspective
- Implementation
- Results
- Related work
- Conclusions and future research



Equational Logic and some Classical Results

Equational Logic

A simple theory:

```
theory STACK is
  sorts Stack Element .
  op nil : -> Stack .
  op top : Stack -> Element .
  op pop : Stack -> Stack .
  op push : Stack Element -> Stack .
  var S : Stack . var E : Element .

  eq top(push(S,E)) = E .
  eq pop(push(S,E)) = S .
endth
```

Note: Theories are typed (many-sorted) to prevent common programming mistakes.

Equational Theories

An equational theory or specification is a pair

$$(\Sigma, E)$$

where Σ is an equational signature (sort and operator names) and E is a set of Σ-equations.

Each equation in E has the form

$$(\forall X)\ l = r$$

where

- X is a set of variables distinct from the equational signature Σ,
- l and r are terms over Σ and X.

Equational Deduction

Given a theory (Σ, E), then the following rules of deduction define the equations that are *deducible*[a]:

Reflexivity *"the equation $t = t$ is deducible"*

Symmetry *"if $t = t'$ is deducible, then $t' = t$ is deducible"*

Transitivity *"if $t = t'$ and $t' = t''$ are deducible, then $t = t''$ is deducible"*

Instantiation *"any substitution instance of an equation is a deducible equation"*

Congruence *"substituting equal terms into the same term yields equal terms"*

[a] *Algebra and Theorem Proving*, Joseph Goguen, unpublished manuscript

Model Theory

Models for equational theories are *algebras*.

Algebras are sets with operations defined on them, one for each symbol in a corresponding signature.

We say that an algebra M *satisfies* an equational theory (Σ, E) if it preserves the equality relations of the equational theory.

We write,

$$M \models e, \text{ for each } e \in E,$$

or

$$M \models E.$$

Note: In algebraic specification programs or software systems are considered models for equational theories; types are models for the sorts, functions and operators are models for the operation symbols in the equational signature

Example

Given an equational theory

```
theory SEMIGROUP is
  sort S .
  op * : S S -> S .
  vars A B C : S .

  eq (A * B) * C = A * (B * C) .
endth
```

The algebra $(\mathbb{I}, +)$ with $\mathbb{I}$ as the interpretation of the sort symbol S and $+ : \mathbb{I} \times \mathbb{I} \longrightarrow \mathbb{I}$ as the interpretation of the $*$-operator is a model, such that

$$(\mathbb{I}, +) \models \texttt{SEMIGROUP}$$

$\Rightarrow$ Integer addition preserves the associativity specified in the equational theory.

Soundness & Completeness

Let (Σ, E) be an equational theory, then for all $M \models E$,

$$M \models (\forall X)\ l = r \text{ iff } E \vdash (\forall X)\ l = r.$$

Automated Deduction

In equational logic automated deduction is accomplished via *term rewriting*:

- Each equation $(\forall X)\ l = r$ is considered a rewrite rule $(\forall X)\ l \rightarrow r$.
- A variant of the *instantiation* rule is used as a rewrite rule application.

$\Rightarrow$ We loose symmetry and congruence; and therefore term rewriting is in general not complete, but it is sound.

Let (Σ, E) be an equational theory, then for all $M \models E$,

$$E \vdash_{RW} (\forall X)\ l = r \text{ implies } M \models (\forall X)\ l = r.$$

Theory Morphisms & Reducts

Given two theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, then a *theory morphism* $\phi : T \to T'$ is a *signature morphism* $\phi : \Sigma \to \Sigma'$, such that

$$E' \vdash \phi(e), \text{ for all } e \in E.$$

"a theory morphism is a translation of a source theory into a target theory in such a way that the translated equations of the source theory can be deduced in the target theory."

Theory Morphisms & Reductions

Let $\phi\colon T \to T'$ be a theory morphism, let M' be an algebra such that $M' \models T'$, then

$$\begin{array}{ccc} \phi M' & \xleftarrow{\Phi} & M' \\ \top\!\!\top & & \top\!\!\top \\ T & \xrightarrow[\phi]{} & T' \end{array}$$

The theory morphism ϕ induces a mapping Φ that takes any T'-algebra M' to the *reduct* $\phi M'$ such that $\phi M' \models T$.

A Simple Example

Given two theories,

```
theory SEMIGROUP is
  ...
  eq (A * B) * C = A * (B * C) .
endth
```

The algebra $(\mathbb{I}, +)$ with $+ : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ as the interpretation of the $*$-operator is a model, such that

$$(\mathbb{I}, +) \models \texttt{SEMIGROUP}$$

and,

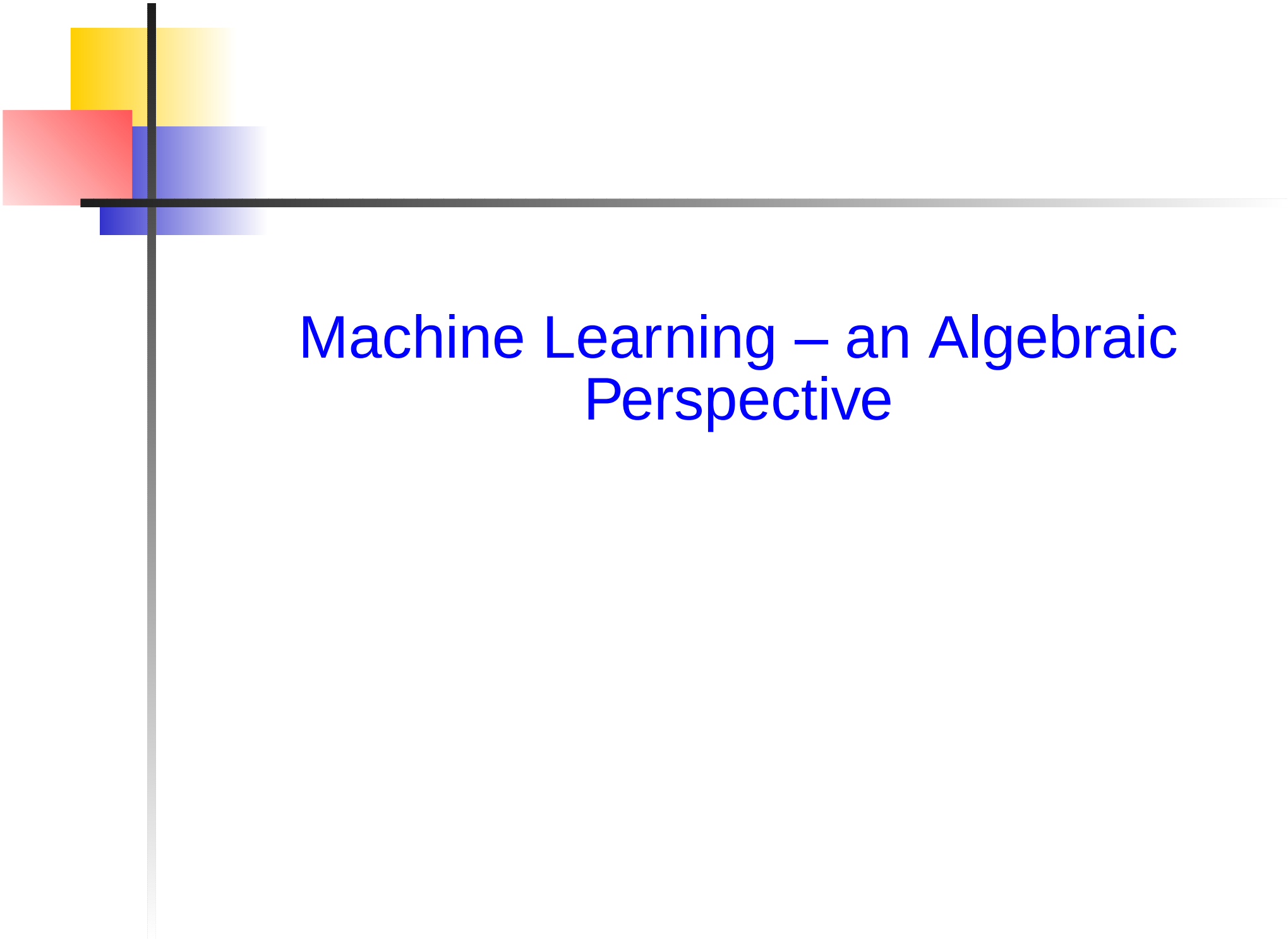
```
theory MONOID is
  ...
  eq (A * B) * C = A * (B * C) .
  eq 1 * A = A .
  eq A * 1 = A .
endth
```

The algebra $(\mathbb{I}, +, 0)$, with $+ : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ as the interpretation of the $*$-operator and $0 \in \mathbb{I}$ as the interpretation for the 1 constant symbol, is a model, such that

$$(\mathbb{I}, +, 0) \models \texttt{MONOID}$$

Then,

$$\texttt{SEMIGROUP} \overset{\phi}{\hookrightarrow} \texttt{MONOID} \text{ implies } (\mathbb{I}, +) \overset{\Phi}{\hookleftarrow} (\mathbb{I}, +, 0).$$



Machine Learning – an Algebraic Perspective

Learning

Goal: Induce algebraic specifications that

(1) entail the positive test cases,

(2) must not entail the negative test cases,

(3) utilize available background information.

$\Rightarrow$ In our case, test cases or facts are ground equations (equations that contain no variables).

Let F be a theory that contains only positive test cases, then we can restate part (1) of our goal with the following diagram,

$$F \xrightarrow{\phi} A$$

"find a theory A such that there exists a theory morphism ϕ from the facts F to A, that is, $A \vdash \phi(e)$ for all $e \in F$"

A Simple Example

```
theory FACTS is
  sorts Stk Elt .
  ops a b : -> Elt .
  op nl : -> Stk .
  op top : Stk -> Elt .
  op pop : Stk -> Stk .
  op push : Stk Elt -> Stk .

  eq top(push(nl,a)) = a .
  eq top(push(push(nl,a),b)) = b .
  eq top(push(push(nl,b),a)) = a .
  eq pop(push(nl,a))= nl .
  eq pop(push(push(nl,a),b)) = push(nl,a) .
  eq pop(push(push(nl,b),a)) = push(nl,b) .
endth
```

$\xrightarrow{\phi}$

```
theory STACK is
  sorts Stk Elt .
  ops a b : -> Elt .
  op nl : -> Stk .
  op top : Stk -> Elt .
  op pop : Stk -> Stk .
  op push : Stk Elt -> Stk .
  var S : Stk .
  var E : Elt .

  eq top(push(S,E)) = E .
  eq pop(push(S,E)) = S .
endth
```

Notes:

Let some algebra M be a model of STACK, then its reduct ϕM is a model of FACTS.

The ground equations in FACTS are the test cases that need to be deducible in the induced theory STACK.

Inequality Constraints

Inequality constraints allow us to specify properties that should *not* hold in an induced theory - part (2) of our goal.

An inequality constraint is an equation of the form

$$(\forall \emptyset)(l \neq r) = \texttt{true}$$

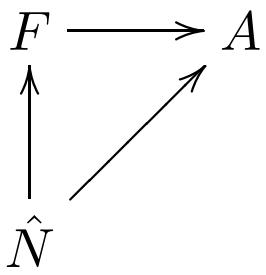
An equational theory $(\Sigma, \hat{N})$, where each $e \in \hat{N}$ is an inequality constraint, is an *inequality constraint theory*.

Observe: Given an equational ground theory (Σ, N) of test cases that should not hold in an induced theory A, i.e. $A \not\vdash \phi(e)$ for all $e \in N$, we can rewrite these as an inequality constraint theory $(\Sigma, \hat{N})$ that should hold in the induced theory, that is, $A \vdash \phi(e)$ for all $e \in \hat{N}$.

Inequality Constraints

We can now extend our notion of theory induction by including inequality constraints.

This can represented as the commuting diagram



"both the facts F and the induced theory A must satisfy the inequality constraints $\hat{N}$"

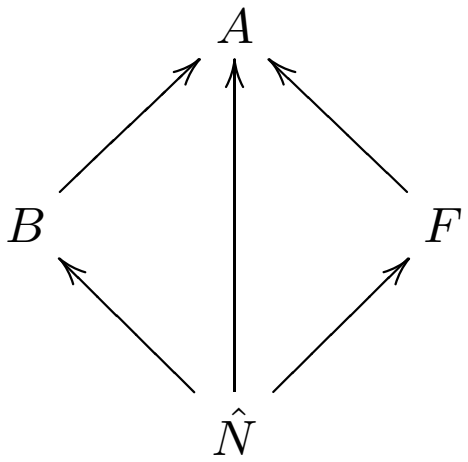
Background Knowledge

A theory (Σ, B) is called a *background theory* if it defines auxiliary concepts that are appropriate for the domain to be learned – part (3) of our goal.

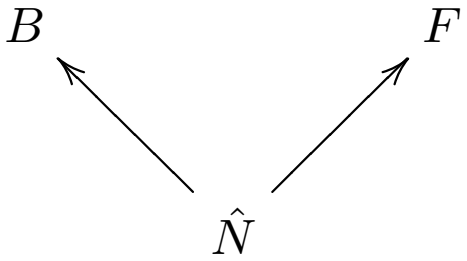
The equations in B do not necessarily have to be ground equations.

Theory Induction

Putting it all together we obtain the following commuting diagram,



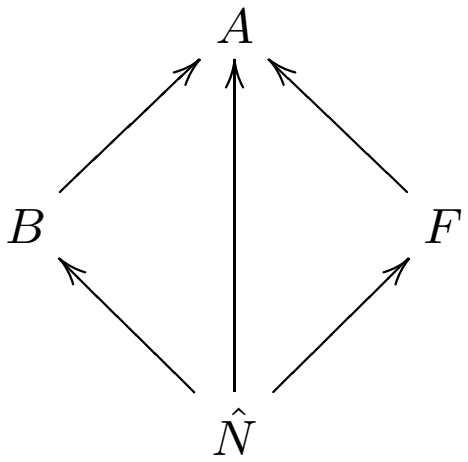
The induced theory A is the apex object of a *cone* over the *specification diagram*,



- A specification diagram represents an *inductive program*.
- There is more structure to an inductive program than a deductive program.

Theory Induction

Given the cone



we can recover classic notions of inductive logic:

completeness - given by theory morphism $F \longrightarrow A$,

consistency - given by the theory morphism $\hat{N} \longrightarrow A$,

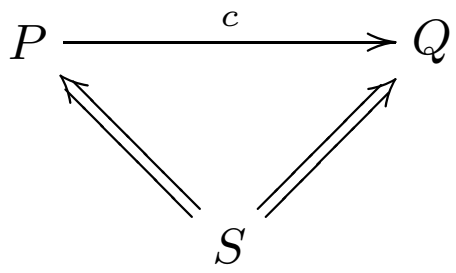
prior necessity - enforced by the fact that $B \not\rightarrow F$,

prior satisfiability - given by the theory morphism $\hat{N} \longrightarrow B$.

Search Space

Let S be some specification diagram, then the cones over S form a category, $\mathbf{H}(S)$, with cone morphisms between them;

> let P and Q be cones in $\mathbf{H}(S)$, then a cone morphism $c\colon\ P \to Q$ is a theory morphism such that the following diagram commutes, [a]



[a] To be technically more accurate, we should say that the diagram commutes for each node in the specification diagram.

Search Space

From an inductive programming point of view we are interested in the most general cone in $\mathbf{H}(S)$, where we define the relation *more general* as follows.

> Let P and Q be cones in $\mathbf{H}(S)$, then we say that Q is *more general* than P iff there exists a cone morphism $P \rightarrow Q$.

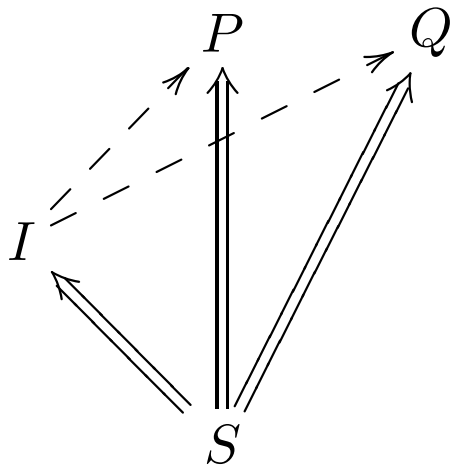
$\Rightarrow$ The most general cone *cannot be constructed*, therefore, we will need to *search* the category $\mathbf{H}(S)$ for an appropriate object.

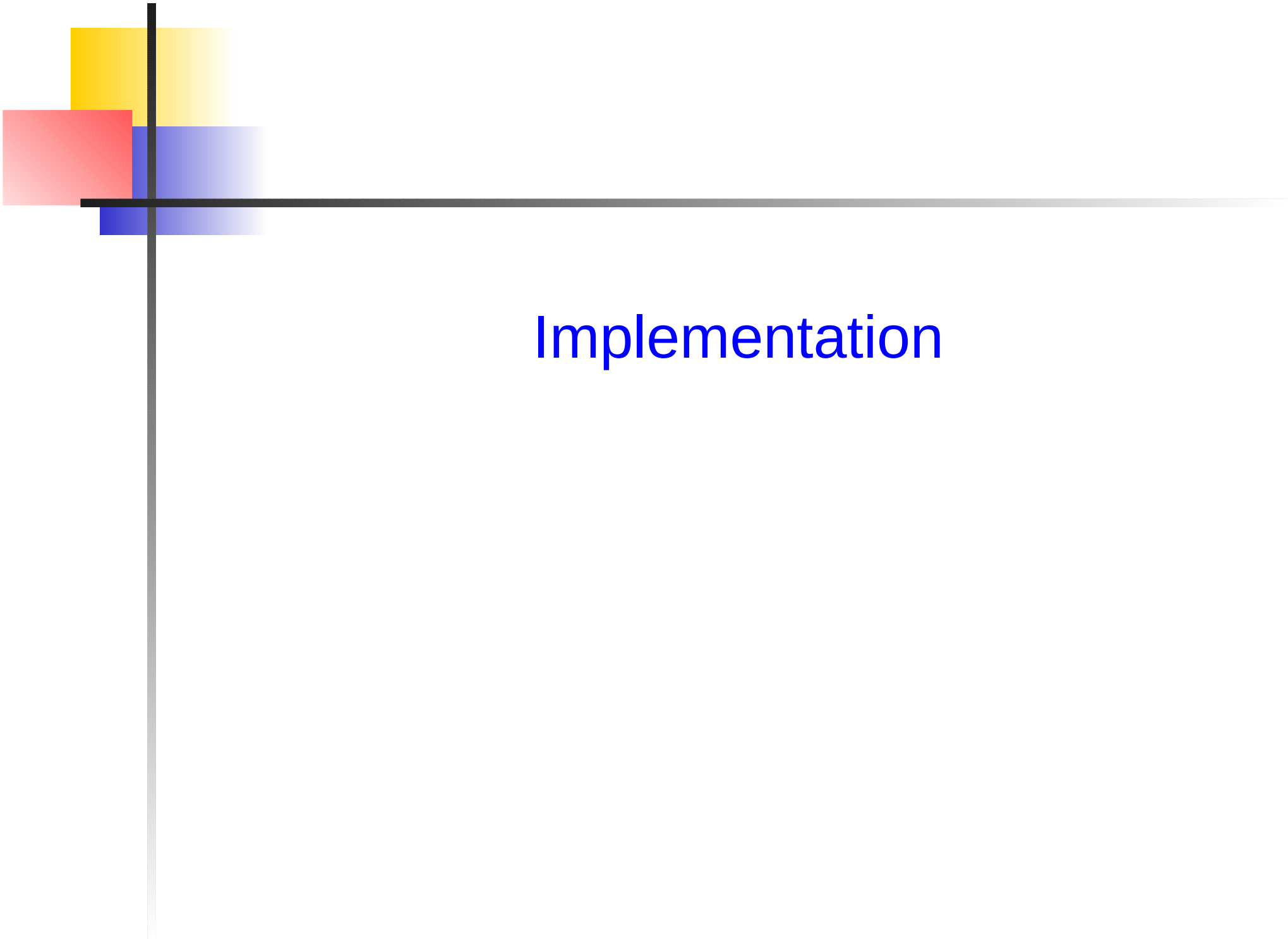
$\Rightarrow$ "Generalization as search" [a]

[a] Mitchell, T.M.: Generalization as search. Artificial Intelligence 18(2) (1982) 203 – 226

Search Space

Observation: The cone we obtain by simply memorizing the facts, inequality constraints, and background theory, call it I, is the *least general cone* in $\mathbf{H}(S)$ – it is the *initial* object in this category.





Implementation

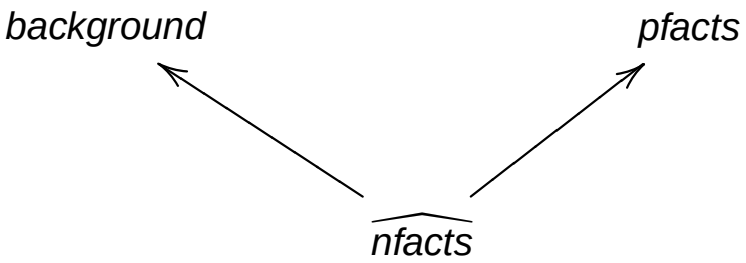
New Command in Maude

We have implemented an equational theory induction system in the specification language Maude.

The induction system is accessible via the new command,

> `induce` *theory-name pfacts nfacts background parameters*

which gives rise to the following specification diagram



The induce command is implemented as an evolutionary search in the category of cones over a specification diagram for the most general cone. [a]

[a] More specifically, an approximation to the most general cone, since evolutionary systems are not guaranteed to find the global optimum.

Genetic Programming

1. Compute an initial (random) population of cones;
2. Evaluate the fitness of each cone H,

$$\text{fitness}(H) = \text{facts}(H) + \text{constraints}(H) + \frac{1}{\text{length}(H)} + \frac{1}{\text{terms}(H)} ,$$

3. Perform reproduction using genetic crossover and mutation operators;
4. Compute new population of cones;
5. Goto step 2 or stop if target criteria have been met.

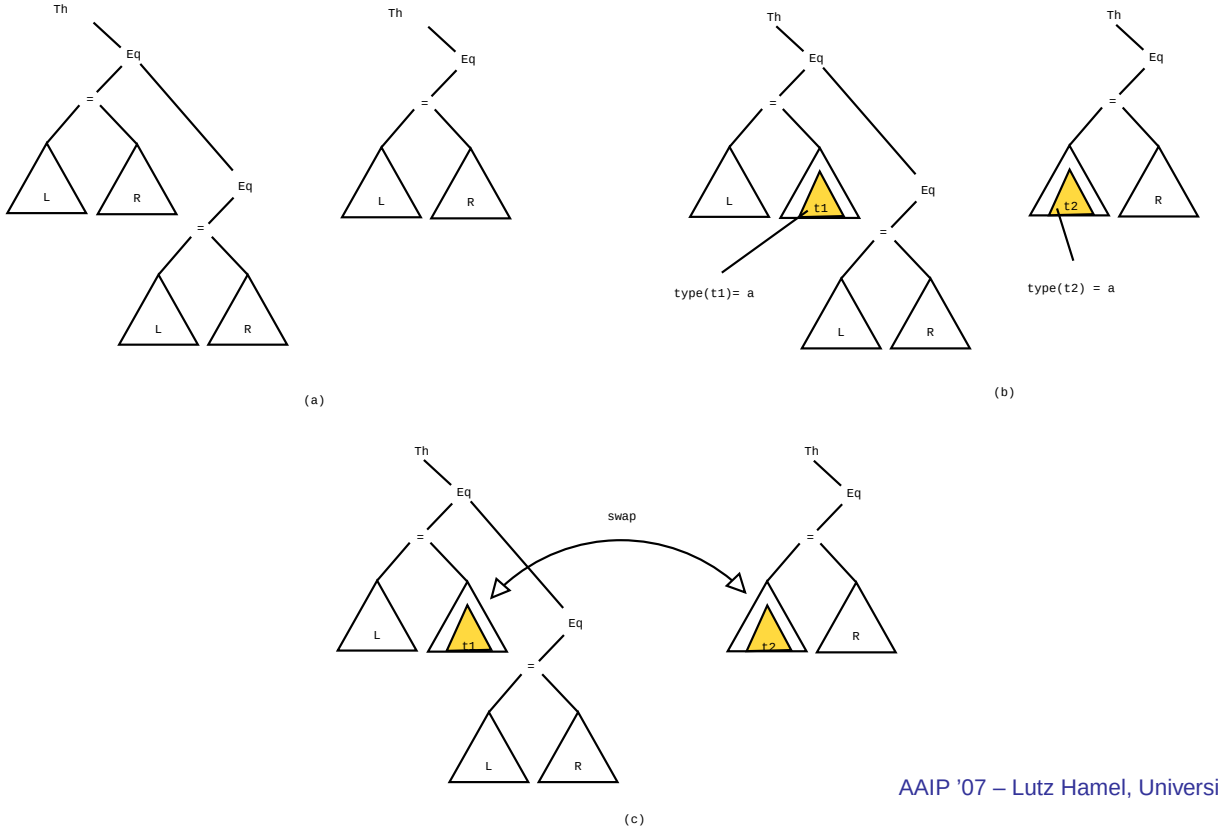
Note: If a cone does not satisfy all facts or constraints we often talk about a pre-cone.

Note: In order to prevent premature convergence we use a *multi-deme* genetic programming model

Genetic Operators: Crossover

Expression-level crossover - exchange expression subtrees at the level of the left and right sides of equations between theories.

Equation-level crossover - exchange whole equations or sets of equations between theories.

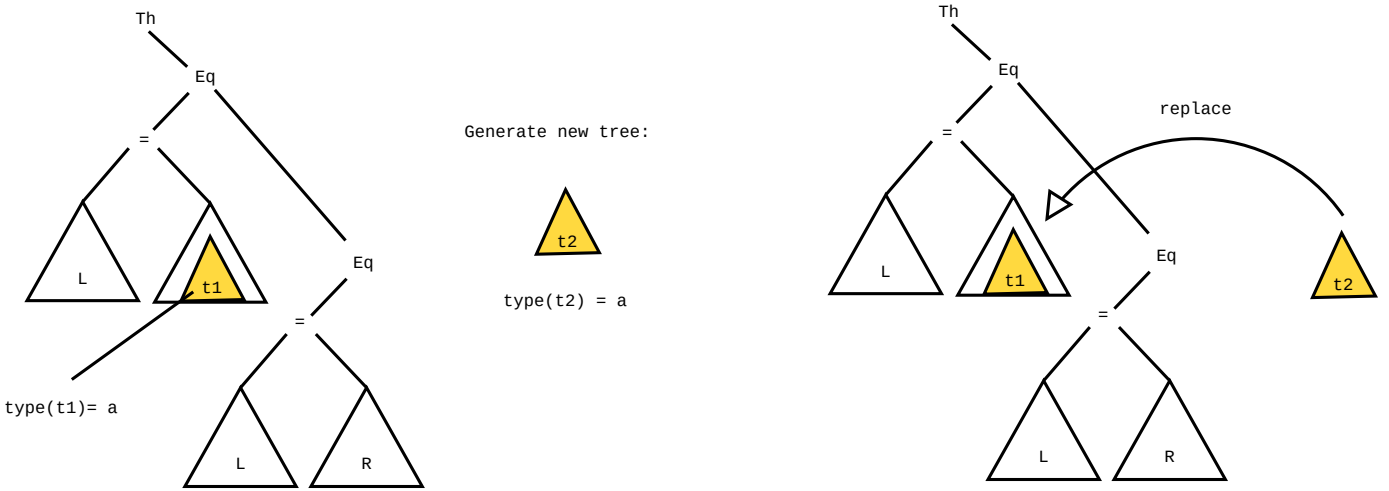


Genetic Operators: Mutation

Expression-level mutation - replace an expression with a newly generated expression of the same sort.

Equation addition/deletion - delete an equation from some theory or add a newly generated equation to some theory.

Literal generalization - replace a literal expression with a variable of the appropriate sort.

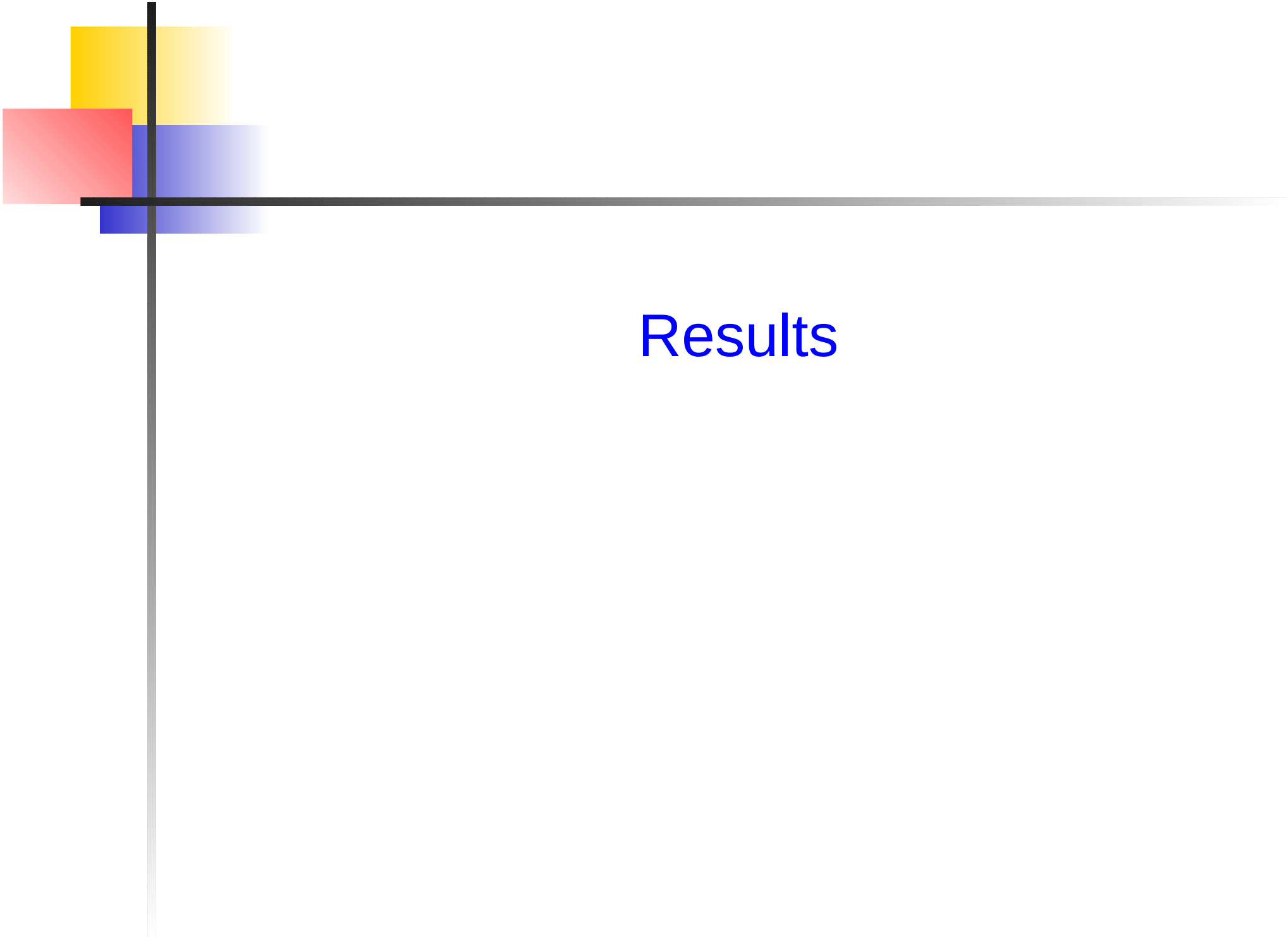


(a) (b)

Implementation Notes

- Theories are represented as strongly typed syntax trees.
- We use fitness convergence rate as a termination criterion.

  Should the fitness of the best individuals increase by less than 1% over 25 generations we terminate the evolutionary search.
- Our genetic programming engine is implemented as a strongly typed genetic programming system using Matthew Wall's GALib C++ library within Maude.
- Since the equations in the hypotheses are generated at random, there is no guarantee that the theories do not contain circularities throwing the rewriting engine into an infinite rewriting loop while computing the fitness of a particular hypothesis. To guard against this situation we allow the user to set a *parameter that limits the number of rewrites* the engine is allowed to perform.



Results

Overview

Experiments:

- Primitive stack operators

  multi-concept learning

- Sum of natural numbers

  recursive definition of sum operator

- Evenness

  recursive definition of even operator

- Sum of list elements

  recursive iteration over a list, background theory

- Play tennis

  classification problem with a fixed number of attributes

- Train directions

  Michalski's train classification problem, structural classification

Overview

Genetic Algorithm Parameters:

300	Total population
100	Maximum number of generations
15	Number of demes (sub-populations)
2	Migration number
0.7	Probability of crossover
0.3	Probability of mutation
0.99	Convergence ratio
25	Convergence window (generations)

List Sum

```
fmod SUM-LIST-PFACTS is
  ...
  eq suml(cons(nil,0)) = 0 .
  eq suml(cons(nil,s(0))) = s(0) .
  eq suml(cons(nil,s(s(0)))) = s(s(0)) .
  eq suml(cons(cons(nil,0),s(0))) = s(0) .
  eq suml(cons(cons(nil,s(0)),s(0))) = s(s(0)) .
  eq suml(cons(cons(nil,s(s(0))),s(0))) = s(s(s(0))) .
  eq suml(cons(cons(nil,s(s(0))),s(s(0)))) = s(s(s(s(0)))) .
  eq suml(cons(cons(nil,0),s(s(0)))) = s(s(0)) .
  eq suml(cons(cons(nil,0),s(s(s(0))))) = s(s(s(0))) .
  eq suml(cons(cons(nil,s(s(0))),0)) = s(s(0)) .
endfm
```

```
fmod SUM-LIST-NFACTS is
  ...
  eq suml(cons(nil,0)) = s(0) .
  eq suml(cons(nil,s(0))) = 0) .
  eq suml(cons(nil,s(s(0)))) = s(0) .
  eq suml(cons(cons(nil,0),s(0))) = s(s(0)) .
  eq suml(cons(cons(nil,s(0)),s(0))) = s(s(s(0))) .
  eq suml(cons(cons(nil,s(0)),s(0))) = s(0) .
  eq suml(cons(cons(nil,s(0)),s(s(0)))) = s(s(0)) .
  eq suml(cons(cons(nil,0),s(s(0)))) = s(s(s(0))) .
  eq suml(cons(cons(cons(nil,s(0)),s(0)),s(0))) = s(s(0)) .
  eq suml(cons(cons(cons(nil,s(0)),0),s(0))) = s(0) .
endfm
```

```
fmod SUM-LIST-BACKGROUND is
  ...
  eq sum(0,X0) = X0 .
  eq sum(s(X0),X1) = s(sum(X0,X1)) .
endfm
```

List Sum

A typical hypothesis:

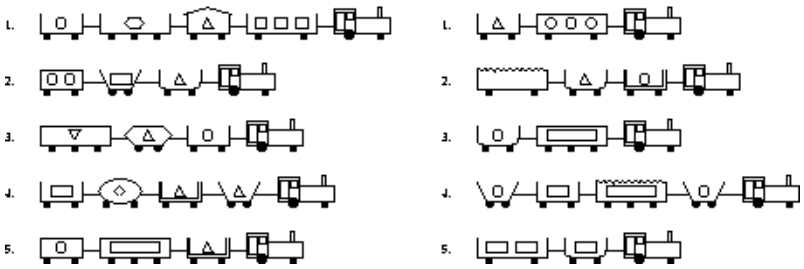
```
fmod SUM-LIST is
  sorts Nat NatList .
  op 0 : -> Nat  .
  op s : Nat -> Nat  .
  op sum : Nat Nat -> Nat .
  op nil : -> NatList .
  op cons : NatList Nat -> NatList .
  op suml : NatList -> Nat .
  vars X0 X1 X2 : Nat .
  vars NatA NatB NatC : Nat .
  vars NatListA NatListB NatListC : Nat .

  eq sum(0,X0) = X0 .
  eq sum(s(X0),X1) = s(sum(X0,X1)) .
  eq suml ( nil ) = 0
  eq suml ( cons ( NatListA , NatB ) ) = sum ( suml ( NatListA ) , NatB )
endfm
```

Notes:

A correct solution was found in 38 out of 50 runs.

The average number of generations to convergence as 35.54 with a standard deviation of 13.88.

Trains



Eastbound Westbound

```
...
eq direction( addCar(addCar(addCar(emptyTrain,
   makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(triangle, s(0)) ) ),
   makeCar( bucketshaped, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) ),
   makeCar( rectangular, short, makeWheels(s(s(0))), flat, makeLoad(circle, s(s(0))) ) )
       ) = east .
 ...
 eq direction( addCar(addCar(emptyTrain,
   makeCar( ushaped, short, makeWheels(s(s(0))), open, makeLoad(rectangle, s(0)) ) ),
   makeCar( rectangular, long, makeWheels(s(s(0))), open, makeLoad(rectangle, s(s(0))) ) )
       ) = west .
 ...
```

Hypothesis:

```
eq direction (addCar(TrainA,makeCar(ShapeA,LengthB,WheelsB,flat,LoadB))) = east
eq direction (addCar(addCar(addCar(TrainA,CarC),CarA),makeCar(rectangular,short,WheelsC,RoofA,LoadC))) = eas
eq direction ( TrainB ) = west
```

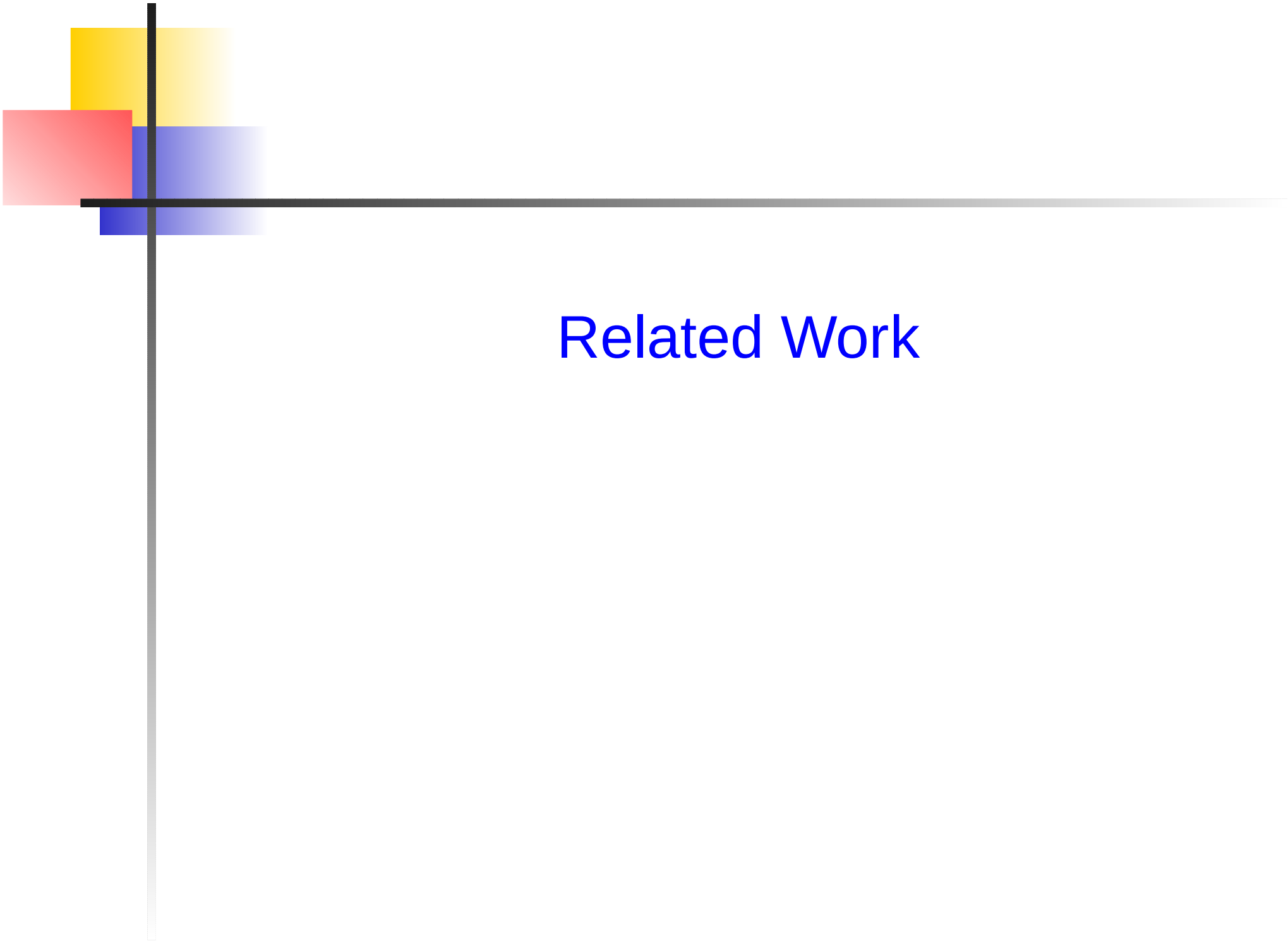
Trains

Notes:

Hypothesis: *“trains ending with a flat-roofed car, or trains at least three cars long and ending with a short, rectangular car are east-bound; all other trains are heading west.”*

The system found a correct solution in 40 out of 50 runs.

The average number of generations to convergence was 62.04 with a standard deviation of 15.33.



Related Work

Related Work

The synthesis of equational and functional programs has a long history in computing extending back into the mid 1970's, see:

Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the Association for Computing Machinery 24(1) (1977) 44–67

Summers, P.: A Methodology for LISP Program Construction from Examples. JACM 24(1) (1977) 161–175

Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. TOPLAS 2(1) (1980) 90–121

Dershowitz, N., Reddy, U.: Deductive and Inductive Synthesis of Equational Programs. JSC 15(5/6) (1993) 467–494

Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7 (2006) 429–454

Related Work

Related systems that share evolutionary search as the generalization strategy are:

Kennedy, C.J., Giraud-Carrier, C.: An evolutionary approach to concept learning with structured data. In: Proceedings of ICANNGA, Springer Verlag (1999) 1–6

Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1) (1995) 55–58

Conclusions and Future Research

Conclusions and Future Research

We have developed a system that supports an inductive programming approach to algebraic specification.

As part of that effort we have developed an algebraic semantics for inductive equational logic programming.

Further work:

- Extend the system to include order-sorted, conditional equational logic.
- We have a general algebraic framework, can we instantiate this framework with other logics that form an institution? e.g. hidden equational logic
- Better genetic operators and primitives in the evolutionary search strategy – hybrid algorithms, viz. Kennedy *et. al* and Olsson.

Thank You!

hamel@cs.uri.edu