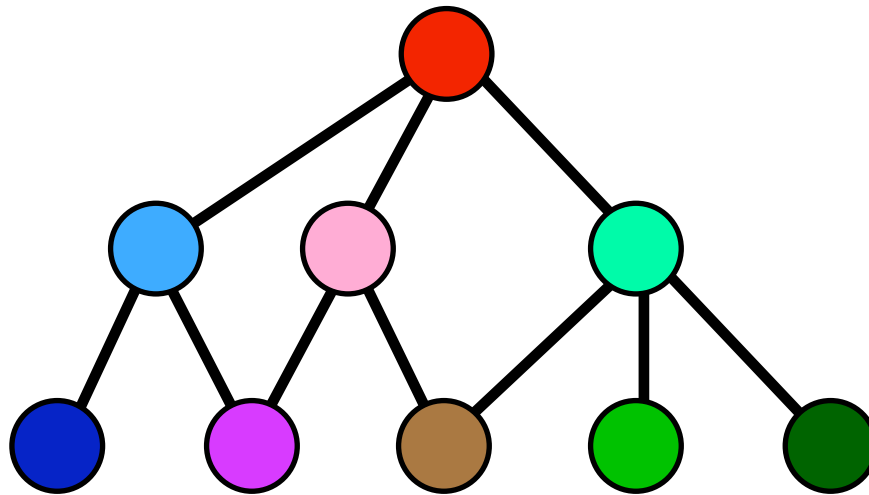


Background

Description Logic Systems



Thomas Russ

Historic Background

Semantic Links

Provide a method of organizing knowledge in a computer system that relied on links between objects to convey meaning.



Structural Classification

Observation that by attaching formal meanings to particular links, one could make useful inferences about the relationship between different objects.

Loom is a Description Logic with a Classifier

■ *Description Logic*

- *Declarative Formalism*
- *Specialized for Writing Descriptions*
- *Has Well-defined Semantics*
- *Supports Automated Inference*

■ *Classifier*

- *Computes Subsumption*
Subsumption = Superset
- *Automatically Manages Type Hierarchy*



In description logics, definitions use a specialized logical language.

Description logics are able to do limited reasoning about concepts expressed in their logic. One important inference is classification (computation of subsumption).

Necessary versus Sufficient

Necessary properties of an object are those properties that are common to all objects of that type.

Being a man is a necessary condition for being a father.

Sufficient properties are those properties that allow one to identify an object as belonging to a type. They do not have to be common to all members of the type.

Speeding is a sufficient reason for being stopped by the police.

Definitions are often necessary and sufficient



Subsumption

Meaning of Subsumption

A more general concept is said to subsume a more specific concept. Members of a subsumed concept are necessarily members of a subsuming concept

Formalization of Meaning

Logic

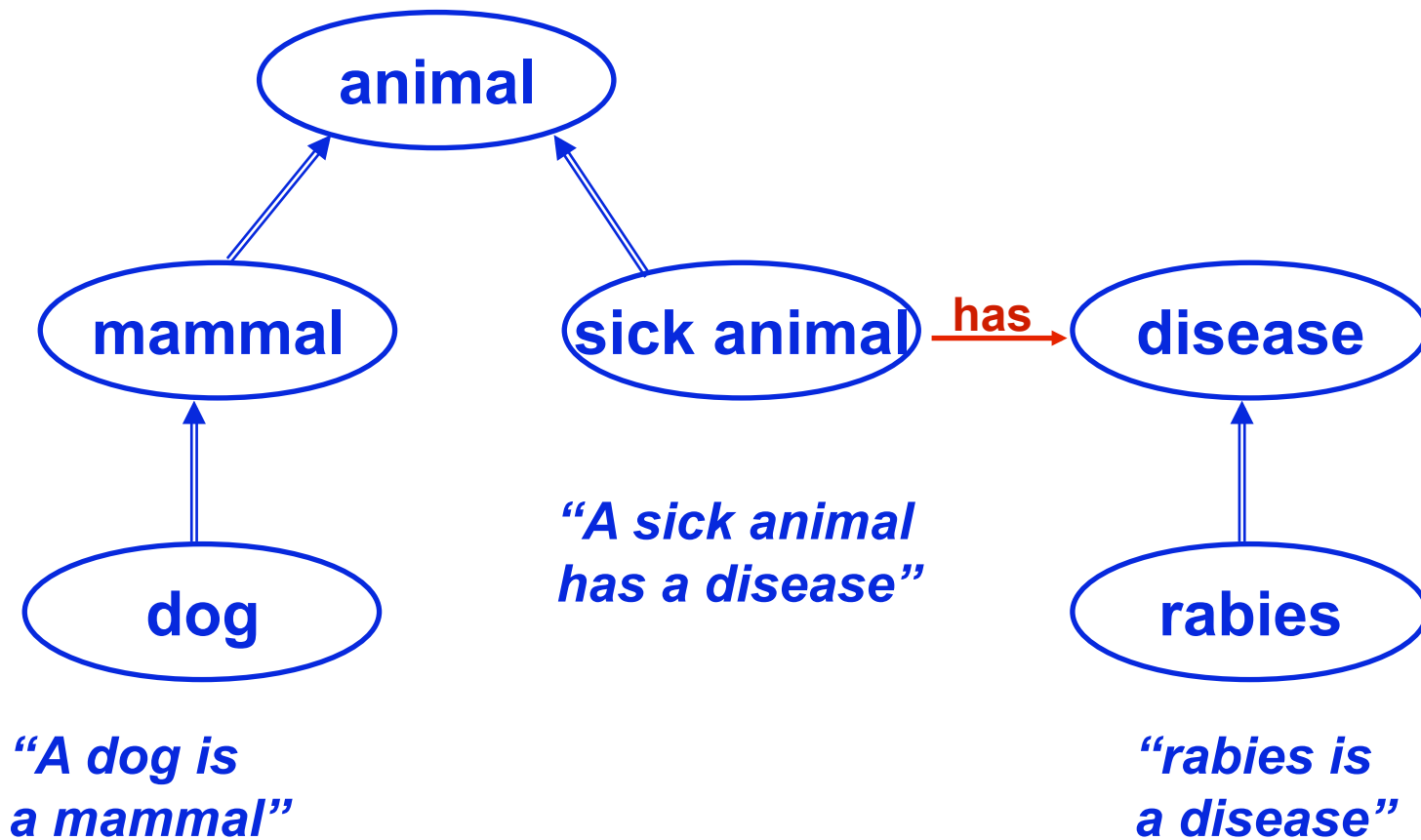
Satisfying a subsumed concept implies that the subsuming concept is satisfied.

Sets

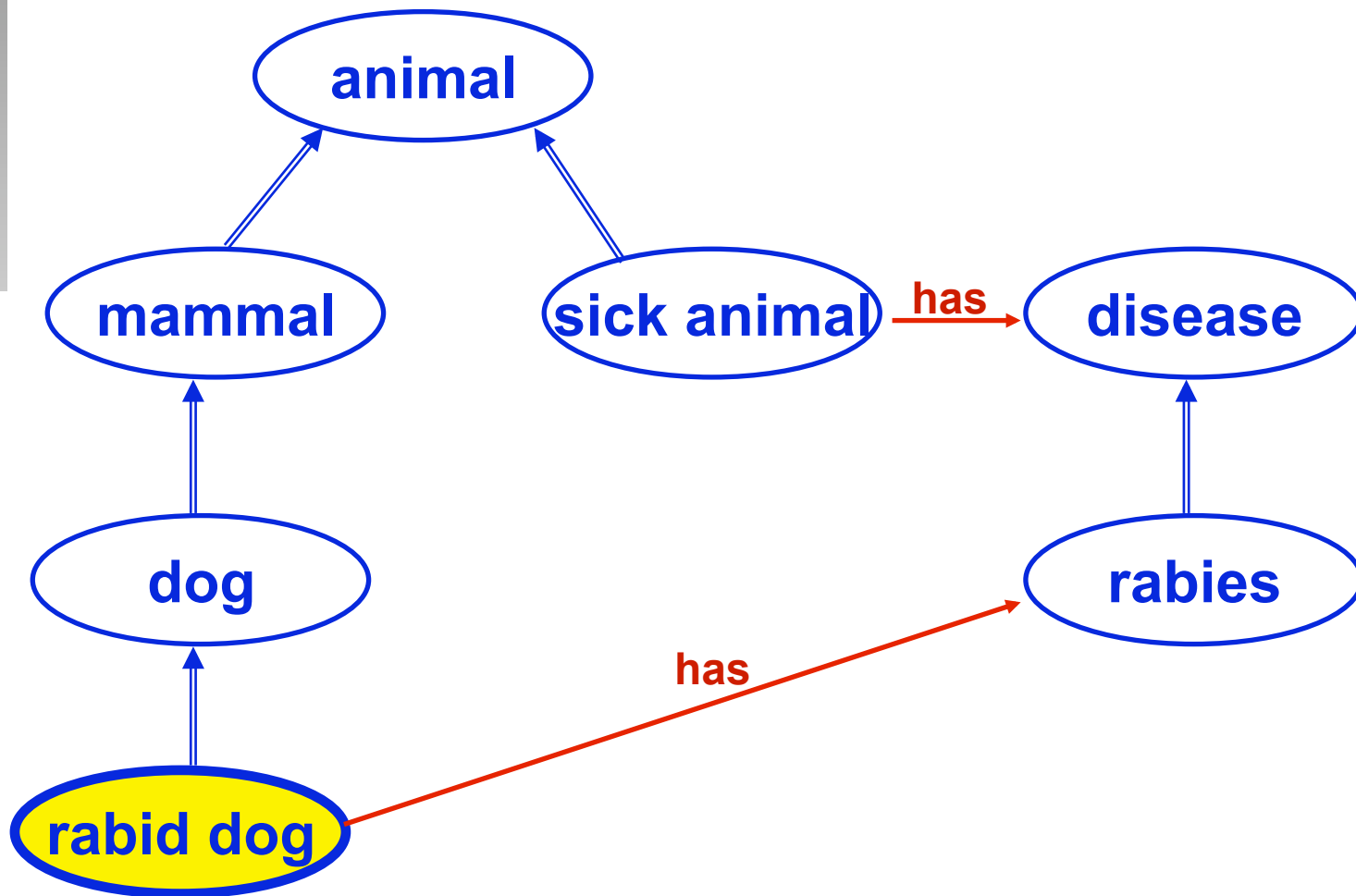
The instances of subsumed concept are necessarily a subset of the subsuming concept's instances.



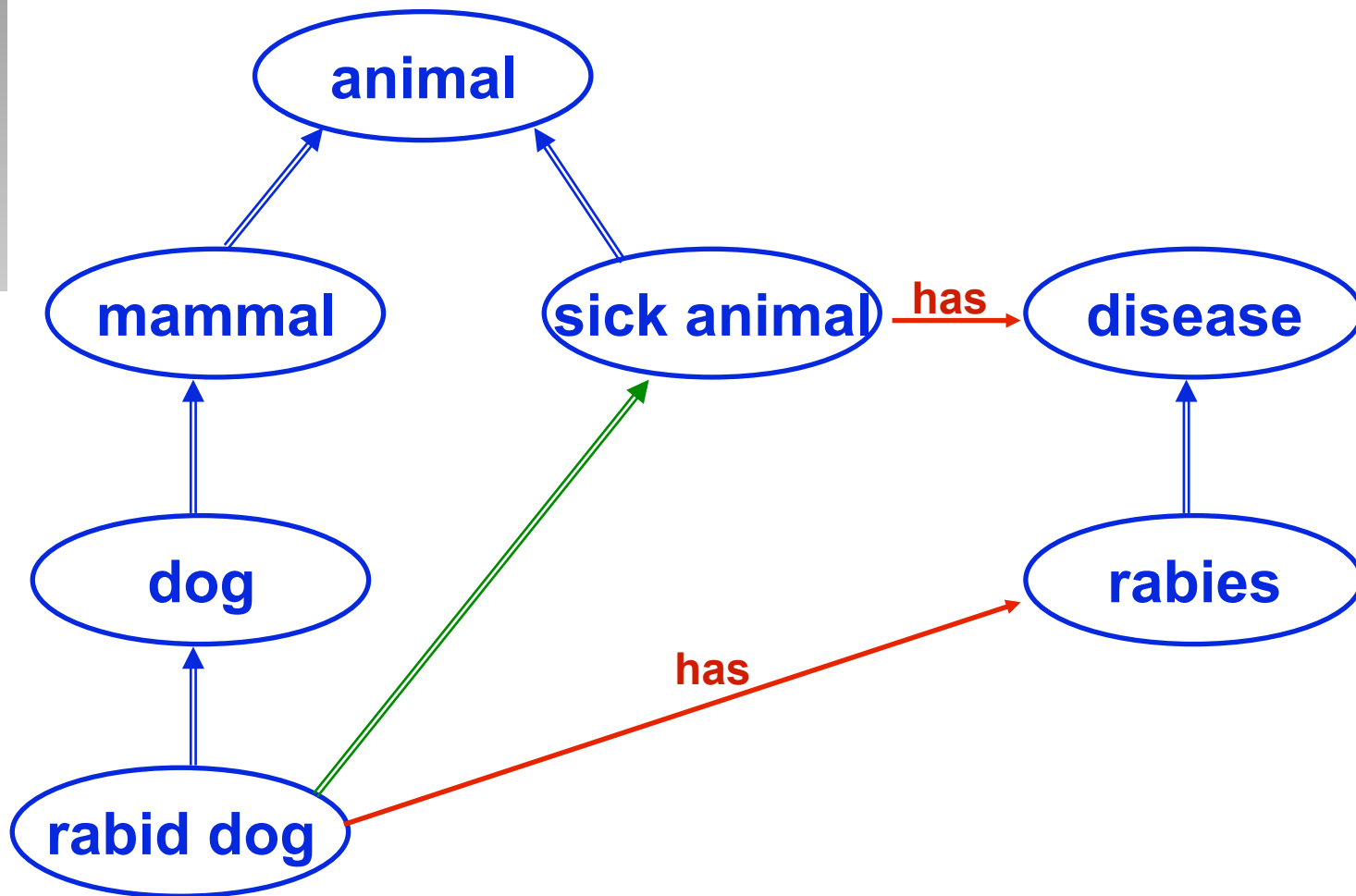
How Does Classification Work?



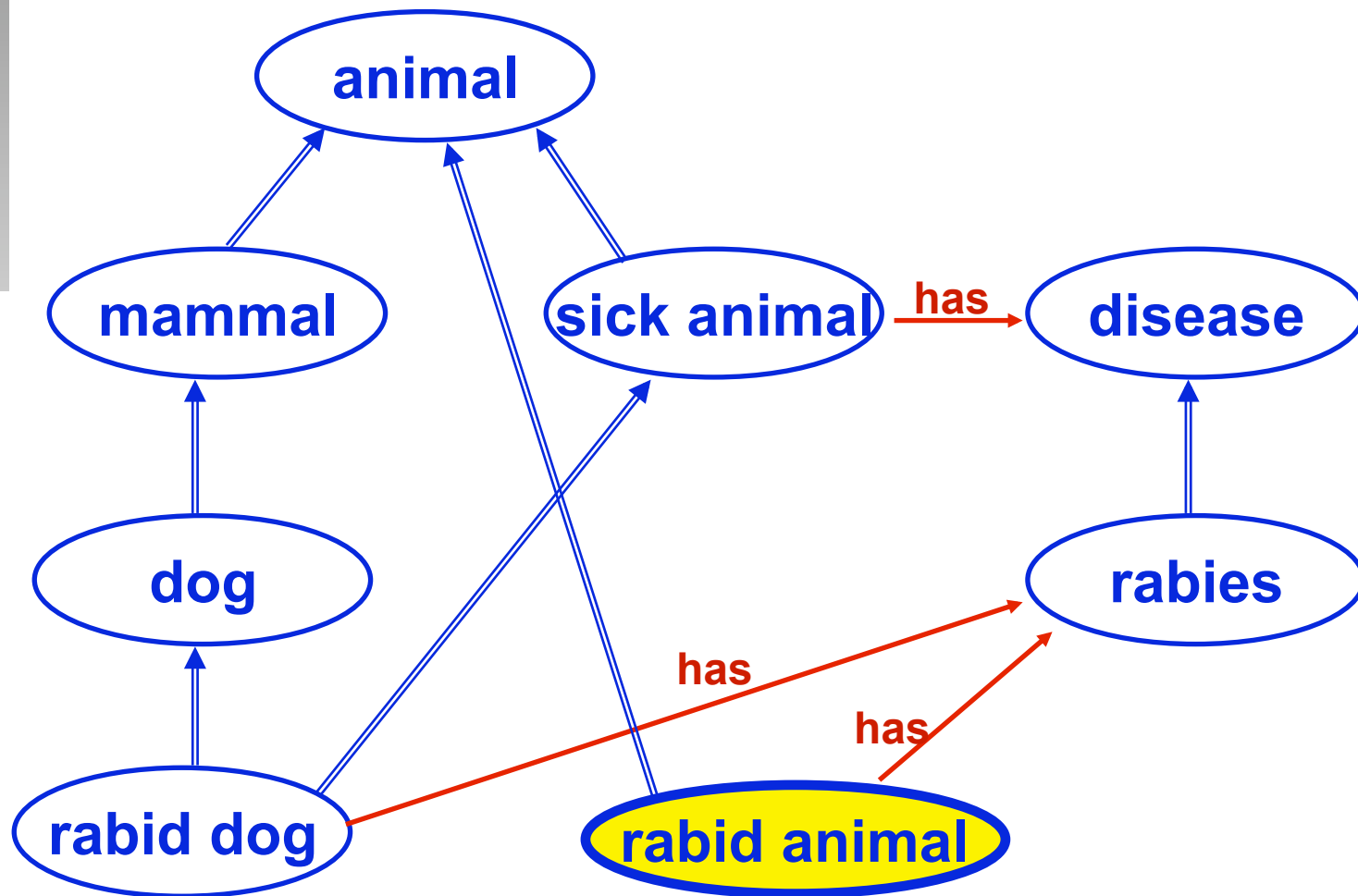
Defining a “rabid dog”



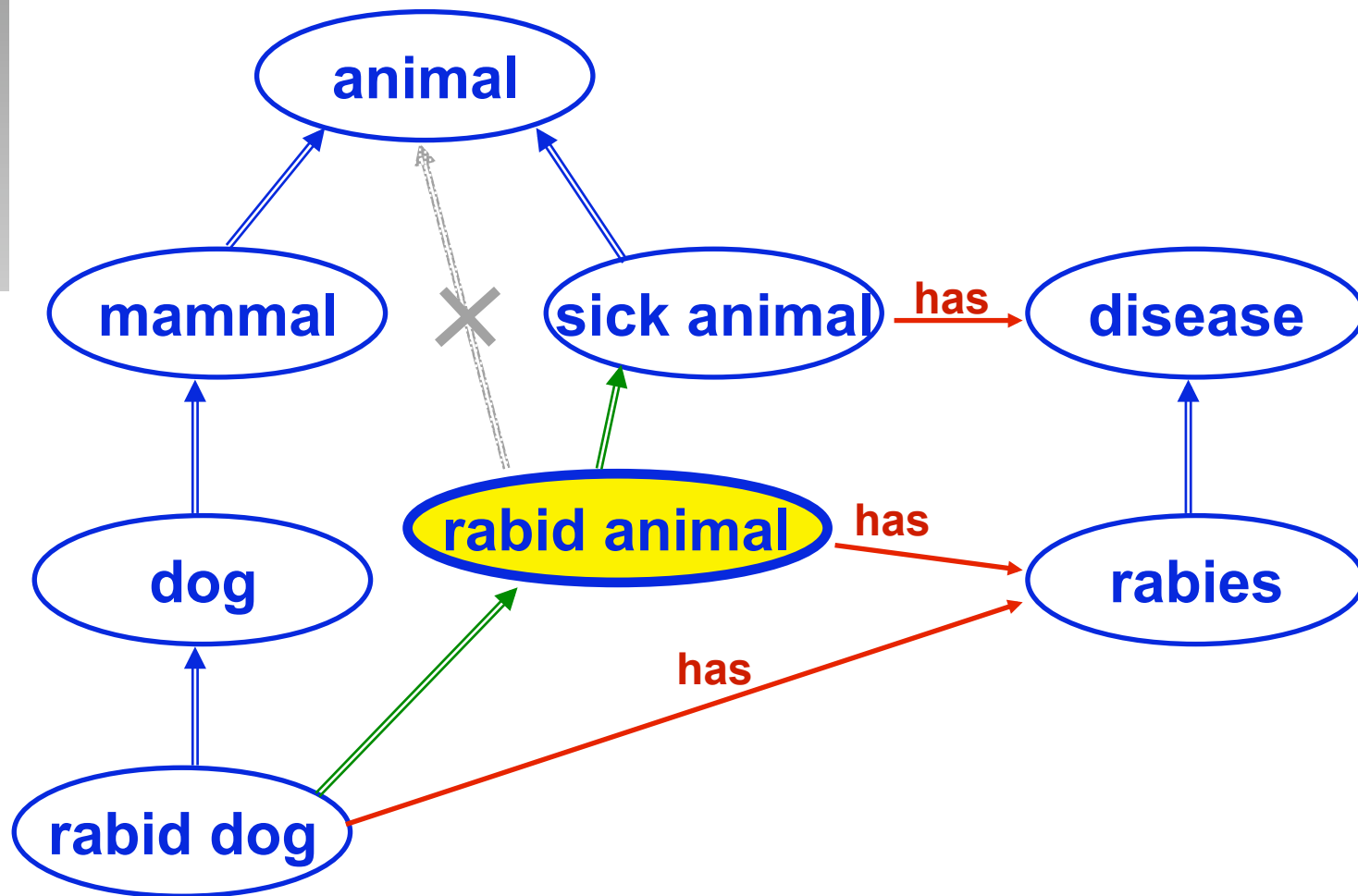
Loom Concludes “sick animal”



Defining “rabid animal”



Loom Places Concept in Hierarchy



Primitive versus Structured (Defined)

Description logics reason with definitions. They prefer to have complete descriptions.

This is often impractical or impossible, especially with natural kinds.

A “primitive” definition is an incomplete definition with the missing element known as the primitiveness. This limits the amount of classification that the system can do automatically.

Example:

Primitive: A Person

**Defined: Parent = Person with at
least 1 child**



Intentional versus Extensional Semantics

Extensional Semantics are a model-theoretic idea. They define the meaning of a description by enumerating the set of objects that satisfy the description.

Intensional Semantics defines the meaning of a description of based on the intent or use of the description.

Example:

Morning-Star

Evening-Star

Extensional: Same object, namely Venus

Intensional: Different objects, one meaning venus seen in the morning and one in the evening.



Definition versus Assertion



A definition is used to describe intrinsic properties of an object. The parts of a description have meaning as a part of a composite description of an object

An assertion is used to describe an incidental property of an object. Asserted facts have meaning on their own.

Example

A black telephone

Could be either a description or an assertion, depending on the meaning and import of “blackness” on the concept telephone.

Open versus Closed World Semantics

Open world recognizes that all information is not available to the system.

Closed world assumes that all (relevant) information about the domain is known to the system.

- "Negation as Failure"
- Common database semantics

Loom offers a choice.



Basic Introduction to Loom

Thomas A. Russ

USC

Information Sciences Institute



What Is Loom?

- *Loom is a Knowledge Representation Language*
- *Loom is a Description Logic*
- *Loom is in the KL-ONE Family of Languages*
- *Loom is a Programming Framework*



Concepts, Relations and Instances

- *Concepts are types. Logically they are unary predicates.*

Dog, Mailman, Theory

- *Relations are tuples. Logically they are n-ary predicates. (Most relations can also be used as logical functions)*

owned-by, employer-of, proof-for

- *Instances are individuals in a domain. They may belong to one or more concepts and participate in relations.*

Fido, Fred, Evolution



Subsumption



- *The main organizing principle behind a description logic is the computation of subsumption.*
- *Concept C1 subsumes another concept C2 when all members of C2 must be members of C1.*
Mammal subsumes Dog
- *C1 is more general and is a super-concept
C2 is more specific and is a sub-concept*
- *Concepts are not related by subsumption are called siblings.*

Subsumption Calculation



- *Loom computes structural subsumption. That means that the subsumption test is based on the structure, or definition, of concepts and relations.*
- *Description logics are derivatives of predicate calculus enhanced by additional combination operators. Subsumption is defined in terms of these additional operators.*
- *Subsumption calculations are done automatically and allow Loom to maintain and organize the knowledge base as it evolves.*

Loom Has Two Main Parts to a Knowledge Base



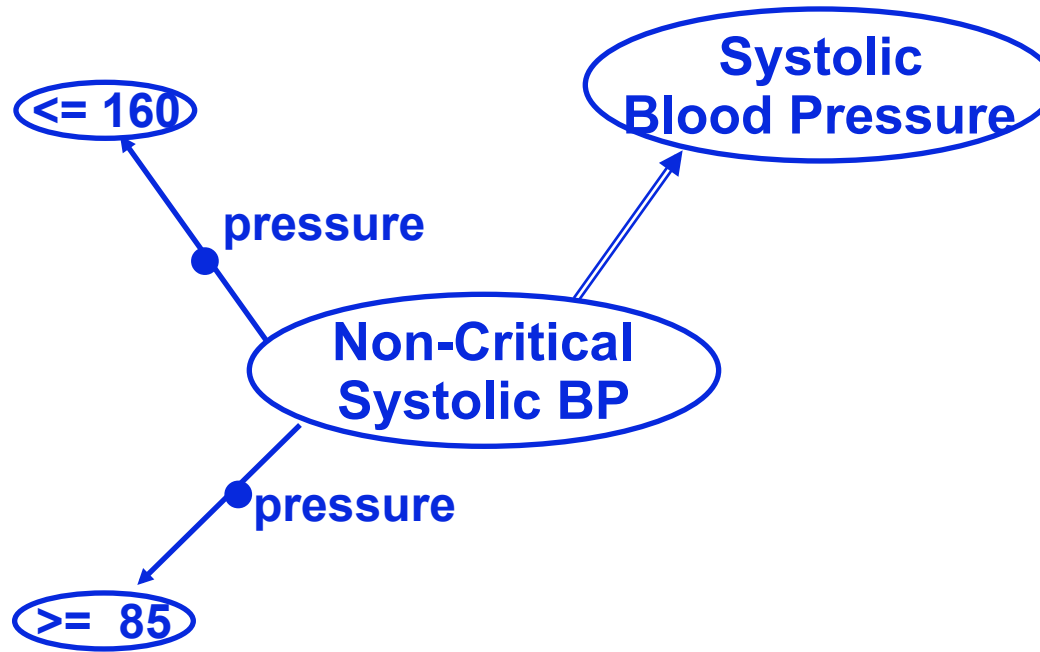
- *The concepts and relations form the terminology. It is the domain-specific language. This is often called the TBox.*
- *Assertions are domain facts. They are made about individuals called instances. This is often called the ABox.*
Assertions use the terminology of the TBox.
- *Instances can belong to concepts and participate in relations.*

Loom Supports Two Languages

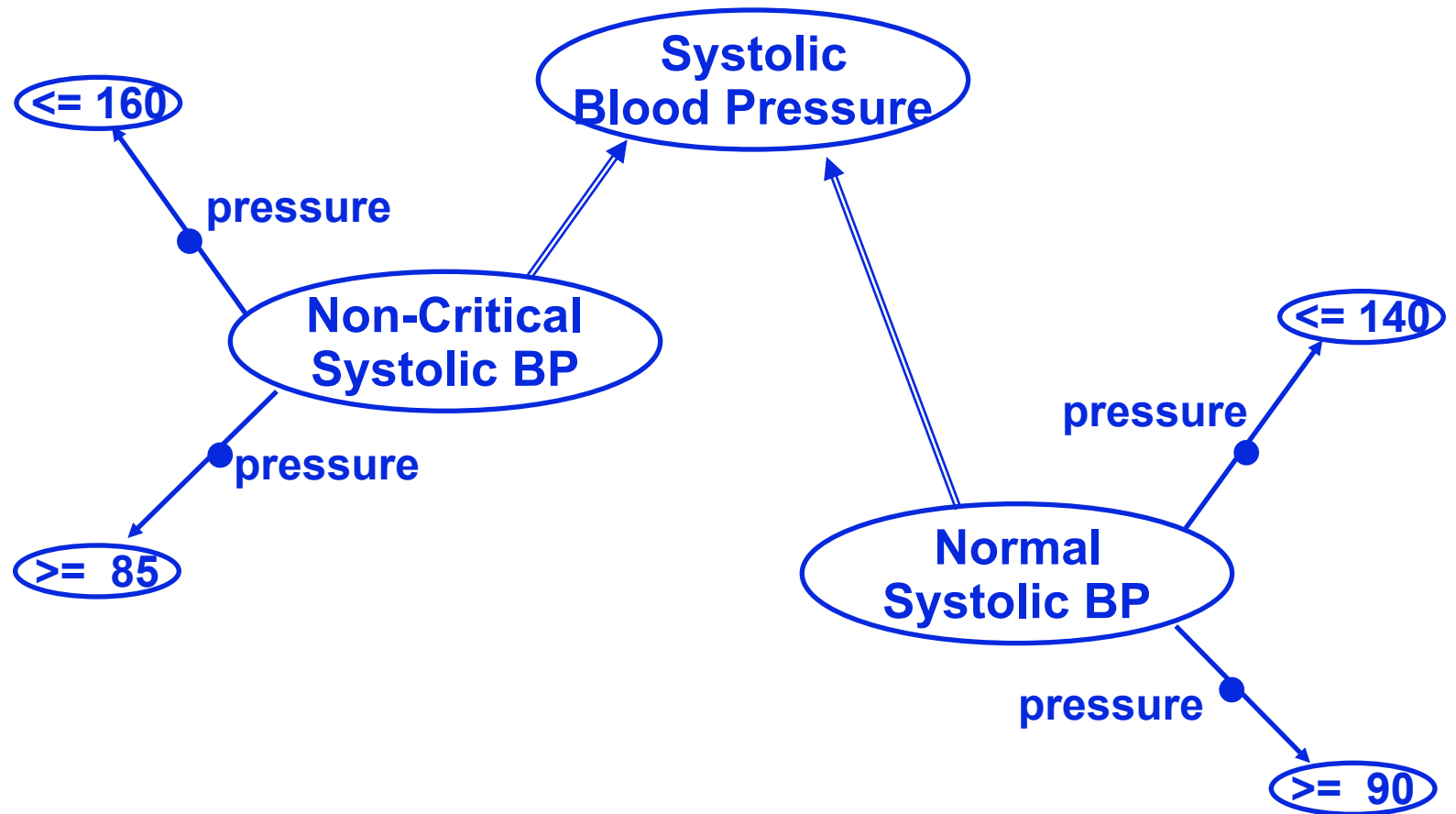


- *The definition language is used to define terminology. The definitions of concepts and relations are written using this language.
The definition language is variable-free.*
- *The query language is used to write questions that are matched against the knowledge base. Queries can be yes/no questions or can request the retrieval of matching instances.
The query language uses variables identified with a leading question mark (?).*
- *Assertions are made using the query language (but all variables must be bound)*

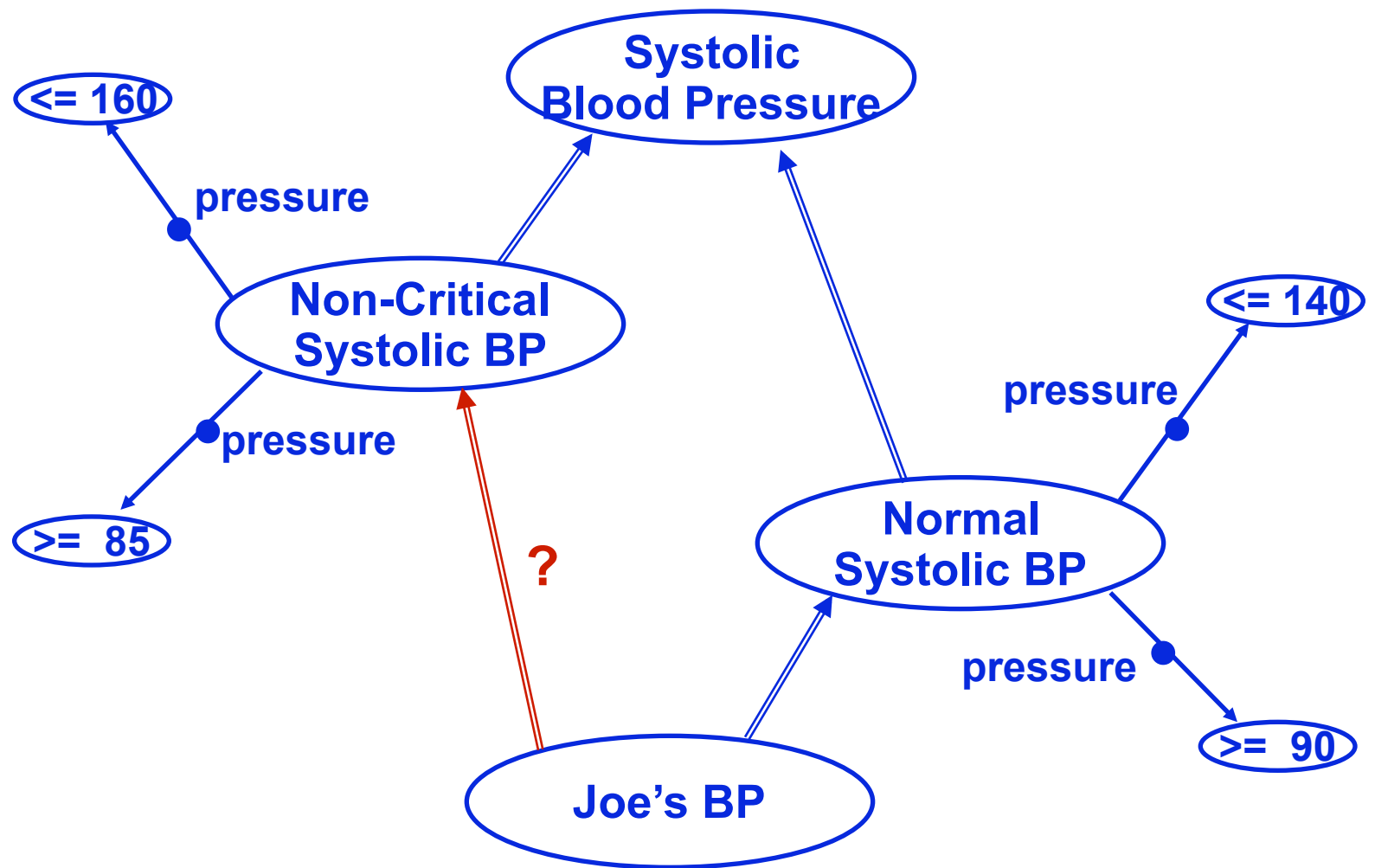
A Non-Critical Blood Pressure is “a Systolic B.P. between 85 and 160.”



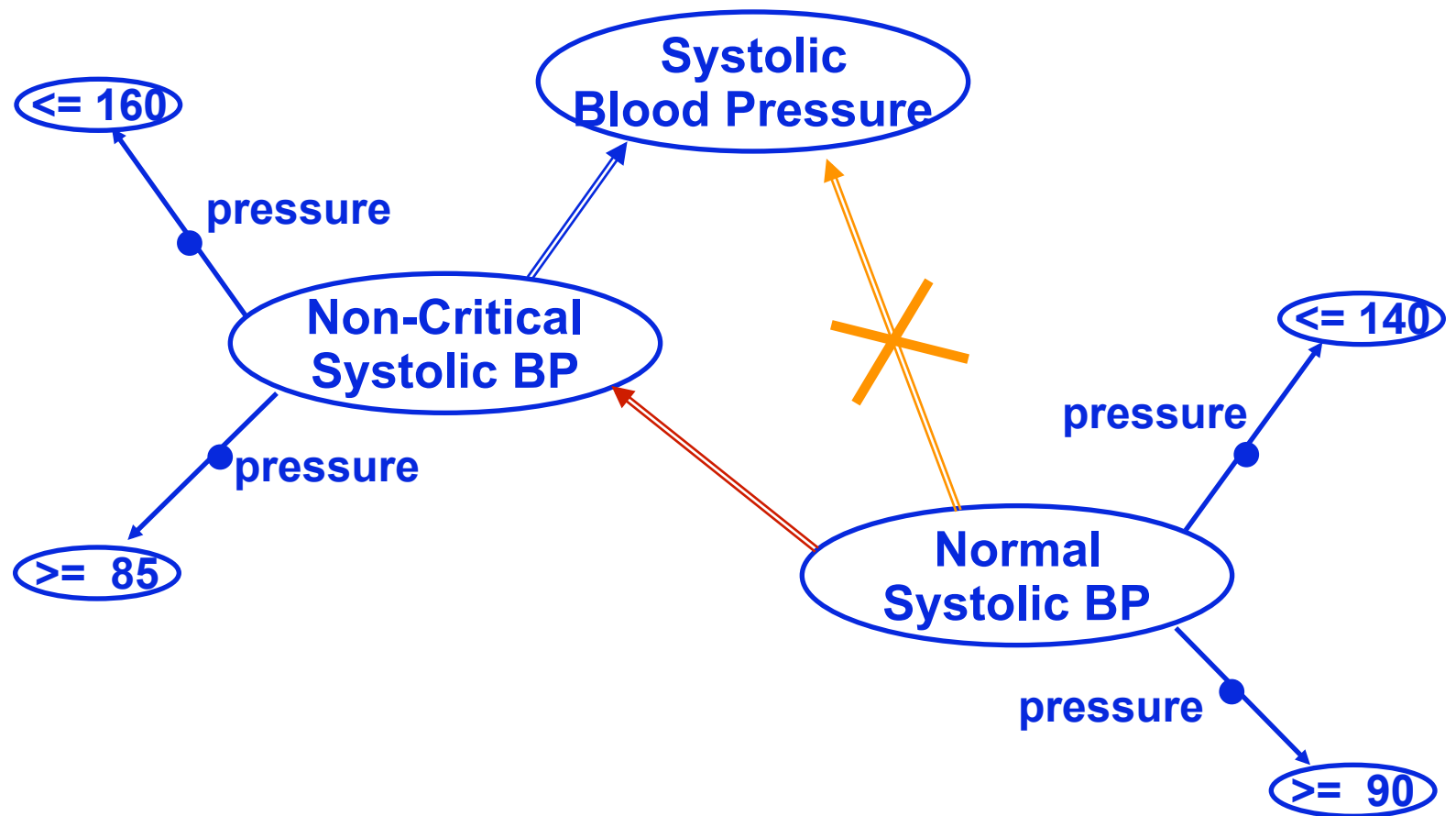
Normal Systolic B.P. is “a Systolic B.P. between 90 and 140.”



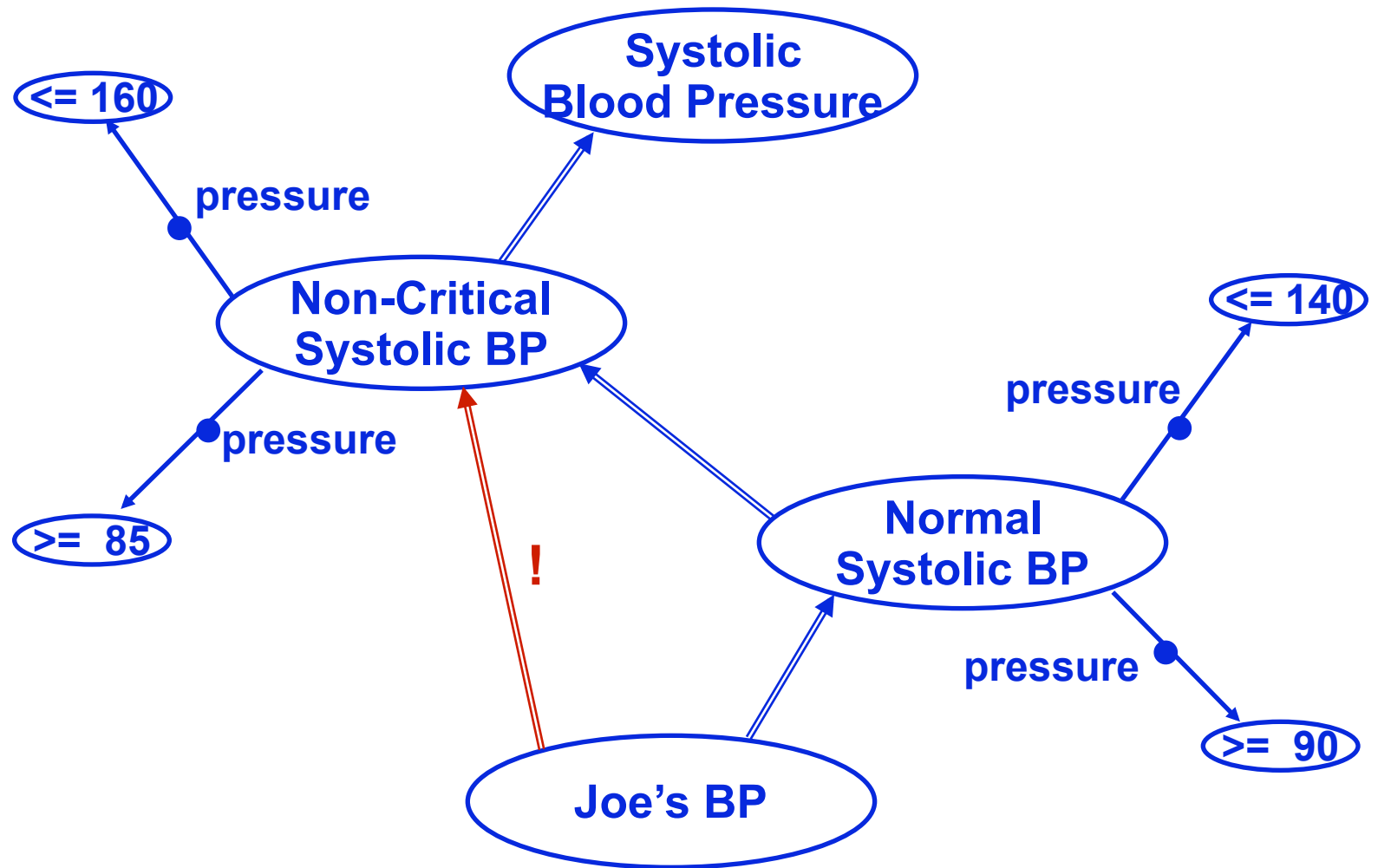
If Joe's BP is Normal is it also Non-Critical?



Concept Classification Infers Normal BP is Subsumed by Non-Critical BP



With Classified Concepts the Answer is Easy to Compute



TBox: *Syntax for Definitions*

Concept Definitions

```
(defconcept <name>  
  :is <definition>)
```

```
(defconcept <name>  
  :is-primitive <definition>)
```

Relation Definitions

```
(defrelation <name>  
  :is <definition>  
  :domain <domain>  
  :range <range>  
  :arity <integer>)
```



Defconcept *:and, :or, :not*

Concepts Defined in Terms of Others

(:and <concept> <concept> ...)

```
(defconcept Slave-Boy
  :is (:and Slave Boy))
```

```
(defconcept Major
  :is (:or Engineering
        Liberal-Arts
        Sciences))
```

```
(defconcept Male :is (:not Female))
```

*Note that “slave” subsumes “slave-boy” but that
“major” subsumes “sciences”*



Defconcept

:at-least, :at-most, :exactly

Number Restrictions on Relations

(:at-least <number> <relation>)

```
(defconcept 4-Door
  :is (:and Car
        (:exactly 4 has-door)))
```

```
(defconcept Parent
  :is (:and Person
        (:at-least 1 has-child)))
```

```
(defconcept Parent2
  :is (:and Person
        (:at-least 2 has-child)))
```

“Parent” subsumes “Parent2”



Defconcept

:all, :some

Range Restrictions on Relations

(:all <relation> <concept>)

```
(defconcept Parent-of-Girls
  :is (:and Person
        (:at-least 1 has-child)
        (:all has-child Female)))
```

```
(defconcept Parent-with-Son
  :is (:and Person
        (:some has-child Male)))
```

“:some” implies “:at-least 1”

“:all” does not imply “:at-least 1”

Booby Trap!



Defconcept *:the*

Combination of “:exactly 1” and “:all”

(:the <relation> <concept>)

```
(defconcept Exclusive-Ford-Dealer
  :is (:and Business
        (:the sells Ford)))
```



```
(defconcept Exclusive-Ford-Dealer
  :is (:and Business
        (:exactly 1 sells)
        (:all sells Ford)))
```



Defconcept

:filled-by, :not-filled-by

*Restricts relations to have specific instance fillers
(or non-fillers)*

```
(:filled-by <relation> <instance> ...)
```

```
(defconcept USC-Employee  
  :is (:and Person  
        (:filled-by employer USC)))
```

*“USC” is an instance, which will be created by
Loom if necessary.*

```
(defconcept Upperclassman  
  :is (:and Person  
        (:not-filled-by  
          college-year 1 2)))
```



Defconcept

:same-as, :subset

Restrictions on the values of relations

```
(:same-as <relation> <relation>)
```

```
(defconcept In-Town-Worker
  :is (:and Person
        (:same-as work-location
                    residence)))
```

An “in-town-worker” has a work location that is the same value as the residence.

```
(defconcept Contented-Worker
  :is (:and Person
        (:subset work-assignments
                  interests)))
```



Defconcept

:relates, comparisons

Arbitrary relations between role fillers

```
(:relates <relation> <relation> ...)
```

```
(defconcept Socially-Linked-to-Boss
  :is (:and Person
        (:relates Brother
                    Best-Friend
                    Boss)))
```

Special cases for numeric comparisons

```
(defconcept Oversubscribed-Course
  :is (:and Course
        (> course-participants
            course-size)))
```



Defconcept :satisfies

More expressive escape. “:satisfies” introduces variables and allows more expressive statements

(:satisfies <variable> <query>)

```
(defconcept Lender
  :is (:satisfies (?x)
    (:exists (?z)
      (owes-money-to ?z ?x))))
```

Drawback: Loom can't do as much reasoning about subsumption.

Tip: Rewrite to Use Specialized Forms



Defconcept Qualified Restrictions :at-least, :at-most, :exactly

Qualified Number Restrictions on Relations

`(:at-least <number> <relation> <concept>)`

```
(defconcept corporation
  :is (:and Business-Entity
        (:exactly 1 employee
                  President)))
```

```
(defconcept Parent-of-son
  :is (:and Person
        (:at-least 1 child Male)))
```

```
(defconcept Parent-of-son2
  :is (:and Person (:some child Male)))
```

“Parent of son” is the same as “Parent of son 2”



Defconcept Qualified Restrictions :all

Qualified Range Restrictions on Relations

(:all <relation> <concept> <concept>)

```
(defconcept Unenlightened-Company
  :is (:and Company
        (:all employee Male
              Supervisor)))
```

A company, all of whose employees who are supervisors are Male.

“Supervisor” qualifies “employee” and limits the set to which the “Male” restriction applies.

Nothing is said about non-Supervisor employees



Defconcept Qualified Restrictions :all

Contrast

```
[1] (defconcept Unenlightened-Company
      :is (:and Company
              (:all employee Male
                      Supervisor))))
```

```
[2] (defconcept Unenlightened-Company2
      :is (:and Company
              (:all employee
                      (:and Male Supervisor))))
```

[1] can have female employees who are not supervisors. [2] has no female employees

[1] can have employees who are not supervisors. All of [2]'s employees must be supervisors.



Defconcept Keywords

:partitions

The name of a partition, the members of which divide the concept into disjoint sub-concepts.

:exhaustive-partitions

A partition that is collectively exhaustive.

:in-partition

Member of a partition.



Defconcept Characteristics

■ *Values of the keyword :characteristics*

:open-world, :closed-world

Declares the concept to use open or closed world semantics. Closed world semantics implies failure-as-negation. In other words, closed world means all concept members are known.

:monotonic, :perfect

Assertions won't be retracted. For :perfect, no subsequent assertions either.



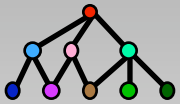
Defrelation :and

Combination of relations. Fillers must satisfy all relations in the conjunction.

```
(:and <relation> <relation> ...)
```

```
(defrelation co-worker-friend  
  :is (:and friend co-worker))
```

Note: Relations cannot use “or”.



Defrelation

:domain, :range

Restrictions on the domain or range of a relation

(:domain <concept>)



```
(defrelation son
  :is (:and child
        (:range male)))
```

```
(defrelation mother-of
  :is (:and child
        (:domain female)))
```

Defrelation :inverse

Defines the inverse relation

(:inverse <relation>)



(defrelation parent) ; primitive

*(defrelation child
:is (:inverse parent))*

A relation cannot be defined as its own inverse using this syntax, since that would be a circular definition. (See :symmetric later)

Defrelation

:compose

Composition is used to chain relations

```
(:compose <relation> <relation> ...)
```

```
(defrelation grandfather  
  :is (:compose parent father))
```

```
(defrelation great-grandfather  
  :is (:compose parent  
                parent  
                father))
```



Defrelation

:satisfies

Escape to allow more complicated descriptions of relations by introducing variables

```
(:satisfies <variables> <query>)
```

```
(defrelation owns-same-stock
  :is (:satisfies (?x ?y)
    (:exists (?z)
      (:and (Stock ?z)
        (owns ?x ?z)
        (owns ?y ?z))))))
```



Defrelation

:domain and :range constraints

The domain and range of a relation can be specified as constraints instead of definitions.

```
(defrelation owns-stock  
  :domain Person  
  :is (:and owns (:range stock)))
```

Non-definitional
constraint

Part of Definition

*Using constraints can be a way of avoiding
circular definitions.*

N-Ary Relations

Loom relations do not need to be binary, but must have a fixed arity

`(defrelation love-triangle :arity 3)`

`(defrelation square :arity 4)`

A “love-triangle” is a relationship among three persons. A “square” is a relation between four geometric points.



Relation Characteristics

:single-valued, :multiple-valued

Determines how many fillers allowed.

:symmetric

The relation is its own inverse.

:commutative

The order of the first N-1 arguments doesn't matter.

:open-world, :closed-world

:monotonic, :perfect

Assertions won't be retracted. For :perfect, no subsequent assertions either.



Using :function and :predicate

The :function and :predicate arguments to defconcept and defrelation allow the specification of arbitrary decision parameters

```
(defconcept odd-Number  
  :is-primitive Number  
  :predicate oddp)
```

*Definition used
for subsumption*

*Predicate determines
membership*

```
(defrelation plus  
  :arity 3  
  :function ((x y) (+ x y)))
```

*Function
arguments*

*Function
body*



Using :function and :predicate

Either function names or lambda expressions (without the “lambda”) can be used.

The function or predicate must be a complete test for the concept or relation. The definition is used only for subsumption computation.

Lisp functions used as :predicates have the same arity as the concept (1) or relation.

Lisp functions used as :functions have take one fewer arguments than the arity of the concept or function.

For example, concept :functions take no arguments.



Assertions

Assertions can be for concept membership:

```
(tell (Dog Fido) (Man Jim)
      (Politician Bush))
```



Assertions can be relation (role) fillers:

```
(tell (owner Fido Jim)
      (supports Jim Bush))
```

Assertions with :about

Many assertions about a single individual gets repetitious:

```
(tell (Man Jim) (Professor Jim)
      (Republican Jim) (age Jim 45)
      (department Jim Biology))
```

:about syntax shortens this by

```
(tell (:about Jim
          Man Professor Republican
          (age 45)
          (department Biology))
```

The subject of the :about clause is not present in any of the assertion forms.



Additional Assertions with :about

Certain assertions can only be made using :about syntax. These are descriptive assertions rather than ground facts:

```
(tell (:about Jim
      (:at-least 2 brother)
      (:at-most 1 job)))
```



Queries

Queries can be Yes/No questions:

```
(ask (Professor Jim))
```

```
(ask (:and (Professor Jim)
            (brother Jim Fred)
            (:not (brother Jim Bob))))
```

Queries can retrieve matching instances:

```
(retrieve ?prof (Professor ?prof))
==> (professor-1 prof-2 ...)
```

```
(retrieve (?prof) (Professor ?prof))
==> ((professor-1) (prof-2) ...)
```



Queries (cont.)

Multiple variables can also be used:

```
(retrieve (?x ?y)
  (:and (married ?x ?y) (Happy ?y)))
```

```
(retrieve (?x ?y ?z)
  (:and (friend ?x ?z)
    (friend ?y ?z)))
```



Query Language

:and, :or

:not

The negation can be proven.

:fail

The positive cannot be proven.

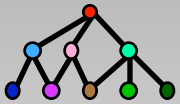
:exists, :for-all

Introduces an existential or universal variable.

:same-as

:collect, :set-of

Introduce sub-query that returns a set.



Query Language Examples

Retrieve friends of friends:

```
(retrieve (?x ?y)
  (:exists (?z)
    (:and (friend ?x ?z)
          (friend ?z ?y))))
```

Retrieve people with a mutual friend

```
(retrieve (?x ?y)
  (:exists (?z)
    (:and (friend ?x ?z)
          (friend ?y ?z))))
```



Query Language Examples (cont.)

Retrieve people all of whose brothers are older

```
(retrieve (?x)
  (:for-all (?z)
    (:implies (brother ?x ?z)
      (> (age ?z) (age ?x))))))
```

Retrieve people with more than 4 siblings:

```
(retrieve ?x
  (:about ?x (:at-least 5 sibling)))
```



Query Language Examples (cont.)

Retrieve the number of people with happy spouses:

```
(retrieve ?n
  (count (:collect (?sp)
    (:exists (?y)
      (:and (person ?y)
        (spouse ?y ?sp)
        (happy ?sp))))
    ?n) )
```



Getting Started

Start the Loom system.

- *Create a new context (theory) and establish a Lisp package:*

```
(loom:use-loom "PROJECT")
```

- *Creates a package named “PROJECT”*
- *Creates a theory context named “PROJECT-THEORY”*



Examining the Knowledge Base

Printing concepts, instances, relations

```
(pc <conceptName>), (pi ...), (pr ...)
(pc parent) ==>
  (defconcept Parent
    :is (:and Person
        (:at-least 1 child)))
```

Finding concepts, instances, relations

```
(fc <conceptName>), (fi ...), (fr ...)
(fc parent) ==> |C|Parent
(fr child) ==> |R|child
(fi Jim) ==> |I|JIM
```



Starting Over

Commands that clear the state of the knowledge base and allow a new start:

■ *Clearing the current workspace:*

`(clear-context)`

■ *Clearing all workspaces. Restoring Loom to its initial state:*

`(initialize-network)`

■ *Clearing instances in all contexts*

`(initialize-instances)`



Loom: Basic Concepts

Thomas A. Russ

**USC
Information Sciences Institute**



Outline of Tutorial

LOOM Terminology
Definition Language
Classifier Examples
Assertion Language
Query Language
Additional Inferences



LOOM Terminology

Two Compartments

TBox for Definitions

ABox for Assertions (Facts)



TBox

Term Forming Language

Concepts

Relations

Subsumption Is Reasoning Method

Defines “Vocabulary” of Domain



Defconcept

```
(defconcept name  
  [:is | :is-primitive] description)
```

Definition Options:

Primitive/Non-primitive

`:is` `:is-primitive`

Combination of Other Concepts

`(:and A B)` `(:or C D)`

Role Number Restrictions

`(:at-least 2 arms)`

Role Type Restrictions

`(:some child male)`



Defconcept Examples

```
(defconcept Soldier)
```

```
(defconcept Medic  
  :is (:and Soldier Medical-Personnel))
```

```
(defconcept Casualty  
  :is (:and Person (:at-least 1 injuries)))
```



Defconcept

```
(defconcept name
  [:is | :is-primitive] descr options)
```

Additional Options:

Characteristics

:closed-

world :monotonic

Roles of the concept

(:roles R1 R2 R3)

roles are relations that are
closely associated with a
particular concept

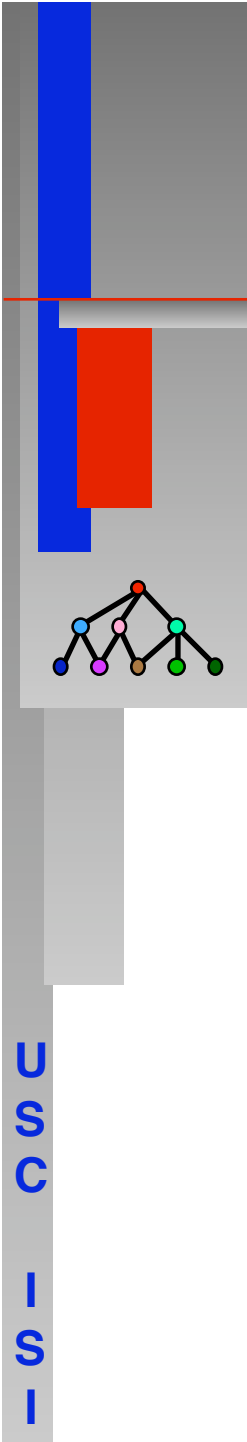


USC ISI

The diagram shows a binary tree structure with 8 leaf nodes. The root node is red. The root has two children: a light blue node on the left and a light green node on the right. The light blue node has two children: a dark blue node on the left and a pink node on the right. The pink node has two children: a purple node on the left and a brown node on the right. The light green node has two children: a green node on the left and a dark green node on the right. The green node has two children: a light green node on the left and a dark green node on the right. The dark green node has two children: a light green node on the left and a dark green node on the right.

USC ISI

The diagram shows a binary tree structure with 8 leaf nodes. The root node is red. The root has two children: a light blue node on the left and a light green node on the right. The light blue node has two children: a dark blue node on the left and a pink node on the right. The pink node has two children: a purple node on the left and a brown node on the right. The light green node has two children: a green node on the left and a dark green node on the right. The green node has two children: a light green node on the left and a dark green node on the right. The dark green node has two children: a light green node on the left and a dark green node on the right.



Defrelation

```
(defrelation name
  [:is | :is-primitive] description)
```

Definition Options:

Primitive/Non-primitive

:is :is-primitive

Relation to Other Concepts

(:compose R S)

Domain and Range Restrictions

(:domain person)

Characteristics

:symmetric :closed-world



USC ISI



```
(defconcept A
  :is (:and B C))
```

(implies A (:and B C))

(implies (:and B C) A)

Observations About Definitions

The Loom language is “variable-free”

Requires special constructs and implicit bindings

(:at-least 2 Child Male)



Sometimes this isn't sufficiently expressive

Adding Expressivity (:satisfies)

Loom definitions can be made more expressive with the “:satisfies” construct

:satisfies is used to introduce variables.

Example—Transitive closure

```
(defrelation R*  
  :is (:satisfies (?x ?y)  
    (:or (R ?x ?y)  
      (:exists ?z  
        (:and (R ?x ?z)  
              (R* ?z ?y))))))
```

Expressivity is higher, but Loom cannot do as much inference with :satisfies clauses

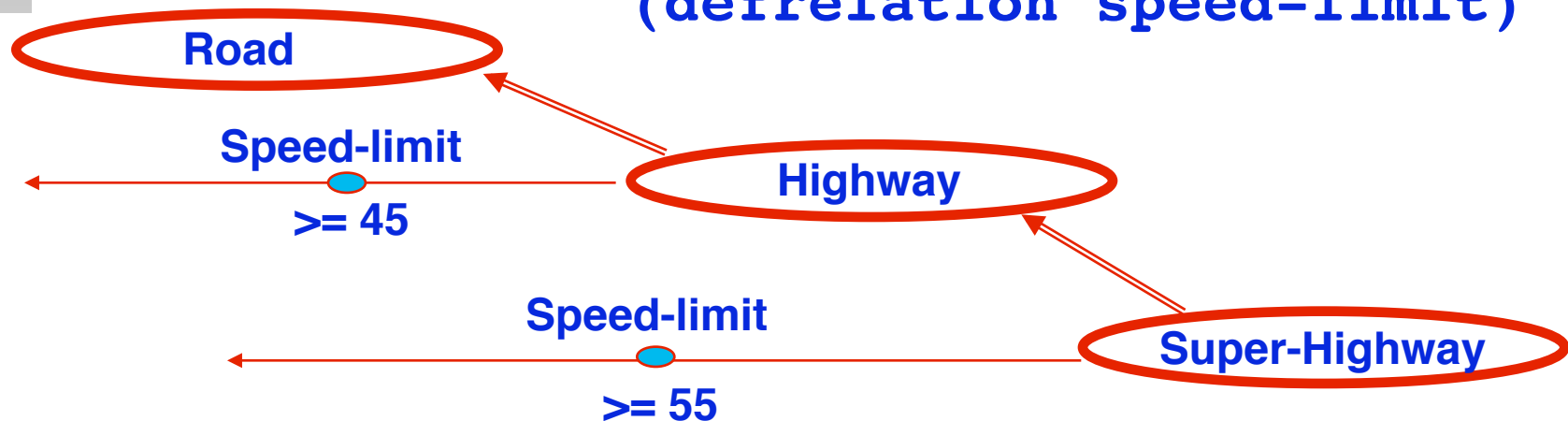


Subsumption

```
(defconcept road)
(defconcept highway
  :is (:and road
        (>= speed-limit 45)))
```

```
(defconcept super-highway
  :is (:and road
        (>= speed-limit 55)))
```

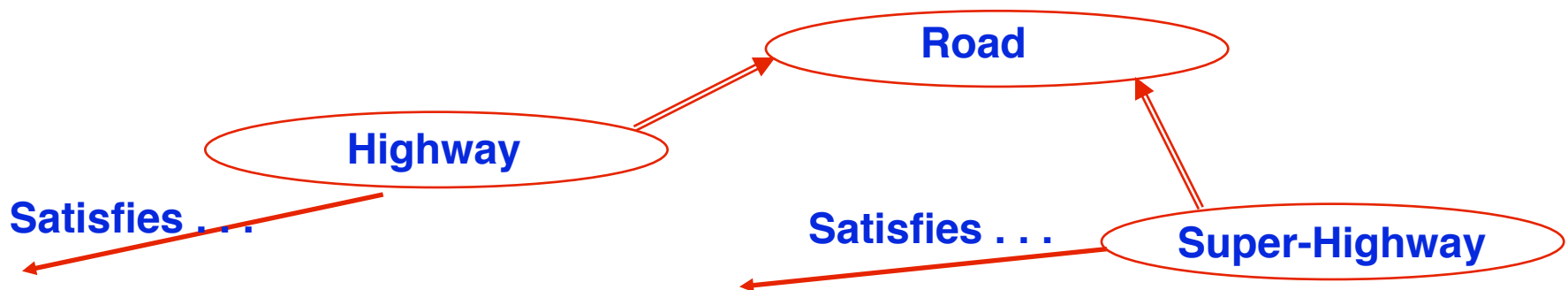
```
(defrelation speed-limit)
```



No Subsumption

```
(defconcept road)
(defrelation speed-limit)
(defconcept highway
  :is (:and road
        (:satisfies (?x)
                     (>= (speed-limit ?x) 45))))
```

```
(defconcept super-highway
  :is (:and road
        (:satisfies (?x)
                     (>= (speed-limit ?x) 55))))
```



Relation Hierarchies

In Loom, relations can also be defined in hierarchies

```
(defrelation child)
  (defrelation son
    :is (:and child (:range Male)))
```

Assertions and queries don't have to match syntactically, only semantically

If one asserts Joe is Tom's son, then asking for Tom's children will return Joe

Similarly, asserting that Joe is a male and Tom's child will let Joe be retrieved by asking for Tom's son



ABox

Uses TBox Vocabulary

Assertions About “Individuals”

Is-a

Role Values

Restrictions



Assertions

Basic Forms:

tell—Adds assertions to the knowledge base

forget—Removes assertions from the knowledge base



Assertions

Basic Syntax

Assert is-a concept
(tell (A Joe) (B Joe))

Concept Name

Instance Identifier



Assertions

Basic Syntax

Assert is-a concept

`(tell (A Joe) (B Joe))`

Assert role values

`(tell (R Joe 3) (R Joe 4) (S Joe 2))`

Role Name

Role Value

Instance Identifier



Assertions

Basic Syntax

Assert is-a concept

```
(tell (A Joe) (B Joe))
```

Assert role values

```
(tell (R Joe 3) (R Joe 4) (S Joe 2))
```

:about Syntax

Used for multiple assertions about a single individual:

```
(tell (:about Joe A B (R 3) (R 4) (S 2)))
```

Instance Identifier

Concept Name

Role Name

Role Value



Assertions

Basic Syntax

Assert is-a concept

```
(tell (A Joe) (B Joe))
```

Assert role values

```
(tell (R Joe 3) (R Joe 4) (S Joe 2))
```

:about Syntax

Used for multiple assertions about a single individual:

```
(tell (:about Joe A B (R 3) (R 4) (S 2)))
```

Allows assertion of restrictions

```
(tell (:about Jim (:at-least 3 R) (R 2)))
```



Queries

Ask About Grounded Facts

Retrieve Individuals Matching Query Schema



Query Language

(ask statement)

Is fido a dog?:

(ask (dog fido))



Query Language

(ask statement)

Is fido a dog?:

(ask (dog fido))

(retrieve var-list query)

Return all dogs in the KB:

(retrieve ?d (dog ?d))



Query Language

(ask statement)

Is fido a dog?:

(ask (dog fido))

(retrieve var-list query)

Return all dogs in the KB:

(retrieve ?d (dog ?d))

Return list of dogs and their owners:

*(retrieve (?d ?o)
(:and (dog ?d)
(owner ?d ?o)))*

Note: Ownerless dogs are not returned.



Different Decompositions

Two Axes:

Cover

Partition

Enable different reasoning strategies.



Cover

```
(defconcept a)
(defconcept b)
(defconcept c)
(defconcept or-abc :is (:or a b c))
```



Cover

```
(defrelation r)
(defrelation s)

(defconcept x)
(defconcept a
  :is-primitive (:and x (:at-most 1 r)))
(defconcept b
  :is-primitive (:and x (:at-most 0 s)))
(defconcept c :is-primitive x)
(defconcept or-abc :is (:or a b c))

(tell (or-abc Joe))
  ;Joe is one-of A, B, or C
(tell (R Joe 1) (R Joe 2) (S Joe 1))
(ask (C Joe))    ==> T
  ;because we can rule out A and B
```

; A common primitive parent
; (ie, "x") is required for
; this inference to be made



Partition

```
(defconcept p :partitions $p$)
```

```
(defconcept x :is-primitive p  
  :in-partition $p$)
```

```
(defconcept y :is-primitive p  
  :in-partition $p$)
```

```
(defconcept z :is-primitive p  
  :in-partition $p$)
```

```
(tell (x i2)) ==> |C|x
```

```
(tell (z i2)) ==> INCOHERENT
```

```
(forget (x i2)) ==> |C|z
```



Mapping from Logic to an Object Framework

Loom's language provides a logical description of instances in terms of properties and restrictions

CLOS classes provide a physical description in terms of slots

Loom concept descriptions can be mapped into CLOS class definitions



Mapping from Logic to an Object Framework

Superclasses can come from

The superconcepts (subsumption) of the concept definition

Explicit specification via :mixin-classes

Slots can be determined multiple ways


All :roles become slots

All restricted relations (:at-least, etc.) in the concept definition become slots

(Optional) All :domain restricted relations become slots.



Mapping from Logic to an Object Framework—Example



```
(defconcept C
  :is (:and A B X
            (:at-least 2 R)
            (:at-most 1 S))
  :roles (P Q)
  :mixin-classes (browser-item))
```



```
(defclass C (A B X browser-item)
  ((R :accessor R :initarg :R
       :initform nil)
   (S :accessor S ...)
   (P :accessor P ...)
   (Q :accessor Q ...)))
```

Summary

TBox Determines Domain Vocabulary

Definitions

Subsumption

Disjointness

ABox Describes Specific Domain

Instances

Facts

Queries Retrieve Information from the ABox

Yes/No Questions

Find Matching Instances



Modeling in a Medical Domain

Basic Concepts

Using Recognition

More Sophisticated Relations

Methods, Actions and Production Rules



Basic Concepts—Personnel

Primitive Concepts

```
(defconcept person)
(defconcept official-responder
  :is-primitive person)
```

Closed World Concepts

```
(defconcept medical-person
  :is-primitive person
  :characteristics :closed-world)
```

Defined Concepts

```
(defconcept medic
  :is (:and official-responder
            medical-person))
```



Basic Concepts—Personnel (alternate)

Primitive Concepts and Relations

```
(defconcept person)
(defrelation training)
```

Defined Concepts

```
(defconcept medical-person
  :is (:and person
        (:some training medical)))

(defconcept emergency-responder
  :is (:and person
        (:some training emergency)))

(defconcept medic
  :is (:and emergency-responder
        medical-person))
```



Basic Concepts—Injury

Full set

```
(defconcept injury
  :is (:one-of `airway `breathing
    `circulation `neurologic-disability
    `exposure `head `neck `chest `other))
```

Subsets (subsumption calculated automatically)

```
(defconcept primary-injury
  :is (:one-of `airway `breathing
    `circulation `neurologic-disability
    `exposure))

(defconcept secondary-injury
  :is (:one-of `head `neck `chest `other))
```



Basic Concepts—Injury (alternate)

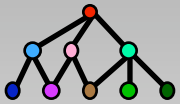
Subsets

```
(defconcept primary-injury
  :is (:one-of 'airway 'breathing
    'circulation 'neurologic-disability
    'exposure))
```

```
(defconcept secondary-injury
  :is (:one-of 'head 'neck 'chest 'other))
```

Union specified by “or”

```
(defconcept injury
  :is (:or primary-injury
    secondary-injury))
```



Basic Relations—Injuries

Primitive

```
(defrelation injuries
  :characteristics :closed-world)
```



Defined (range restricted)

```
(defrelation primary-injuries
  :is (:and injuries
        (:range primary-injury)))

(defrelation secondary-injuries
  :is (:and injuries
        (:range secondary-injury)))
```

Closed world by inheritance

Basic Concepts—Casualties

Defined by number restrictions

```
(defconcept casualty
  :is (:and person
        (:at-least 1 injuries)))
```

```
(defconcept critical-casualty
  :is (:and person
        (:at-least 1
          primary-injuries)))
```

Negated concepts can also be formed

```
(defconcept non-critical-casualty
  :is (:and casualty
        (:at-most 0
          primary-injuries)))
```

Needs closed world!

Implies ≥ 1 injuries



Basic Concepts—Negations

Negated concepts can also be expressed

```
(defconcept helper
  :is (:and person
        (:at-most 0 injuries)
        (:not medical-person)))
```

Recall that “medical-person” was declared to be closed world

This is crucial to reasoning with “:not”

Without the closed world assumption, any individual not explicitly asserted to not be a medical-person could conceivably be one.

This uncertainty would inhibit recognition.



Using Recognition

Loom can recognize when assertions about individuals causes them to fulfill definitions

This allows information to be added as it becomes available

The logical consequences of the existing information is always maintained

Example:

```
(tell (:about p2 Person
      (injuries `airway)
      (injuries `other)))
```

p2 is no longer a "Helper"

p2 is now a "Casualty" and a
"Critical Casualty"



Sophisticated Relations

Some relations can involve sophisticated calculations

Loom provides a method for defining a relation that is the result of a calculation rather than an assertion

:predicate indicates a test for the relation

:function indicates a generator for the relation

Such relations are assumed to be single-valued.



Sophisticated Relations—Geography

We need to associate a location with individuals

```
(defrelation location
  :characteristics :single-valued)
```

We want to calculate distance between locations

```
(defrelation distance-from-locations
  :arity 3
  :function grid-distance)
```

The auxiliary function does the calculation

```
(defun grid-distance (loc1 loc2)
  (sqrt (* (- (loc-x loc2)
              (loc-x loc1))
          ...)))
```



Sophisticated Relations—Geography

We also want to find the distance between individuals

```
(defrelation distance :arity 3
  :is (:satisfies (?x ?y ?d)
    (distance-from-location
      (location ?x)
      (location ?y)
      ?d) ) )
```

Direction can be handled analogously

Loom uses computed relations in backward chaining mode only—Information is not propagated forward.



Sophisticated Relations— Inference Direction

*Concepts and relations can be defined in terms of
computed relations:*

```
(defrelation in-range
  :is (:satisfies (?x ?y)
        (< (distance ?x ?y)
            (range ?x))))
```

*This relation can be queried, but it will not
propagate information forward.*

```
(ask (in-range helo-1 Hospital))
(retrieve ?c
  (:and (casualty ?c)
        (in-range helo-1 ?c)))
```



Sophisticated Relations—Alternate Inheritance

Problem: How can we automatically update the locations of individuals being transported by a vehicle?

- ***Each time the vehicle moves, update all passenger locations***
- ***Determine the passenger location based on the vehicle location***



Sophisticated Relations—Transitive Closures

Base relation “contained-in” is single-valued

```
(defrelation contained-in
  :characteristics :single-valued)
```



Transitive Closures

```
(defrelation contained-in* :is
  (:satisfies (?x ?y)
    (:exists (?z)
      (:and (contained-in ?x ?z)
        (contained-in* ?z ?y))))))
```

Note the recursive definition



Transitive Relation Idiom

Standard Definition of a Transitive Relation R^ Based on the Relation R :*

```
(defrelation  $R^*$  :is
  (:satisfies (?x ?y)
    (:exists (?z)
      (:and ( $R$  ?x ?z)
        ( $R^*$  ?z ?y ))))))
```



Sophisticated Relations—Following a Transitive Link

Base relation “position” is single-valued

```
(defrelation position
  :characteristics :single-valued)
```

Transitive Closures

```
(defrelation position* :is
  (:satisfies (?x ?y)
    (:exists (?z)
      (:and (contained-in* ?x ?z)
              (position ?z ?y))))))
```

The transitive link is followed in this relation to find a ?z with a position. Note that this will find ALL such ?z's!



Sophisticated Relations—Alternate Inheritance Path

Base relation requires inverse

```
(defrelation contained-in)
(defrelation contains
  :is (:inverse contained-in))
```

“position” inherits via “contained-in”

```
(defrelation position
  :inheritance-link contained-in)
```

This allows the creation of meaningful “part-of” hierarchies, with inheritance of appropriate properties.



Methods, Actions and Production Rules

Methods specify procedures that are specialized by Loom queries

Loom methods have a richer vocabulary than CLOS methods

Actions specify properties of methods such as selection rules

Production rules trigger on changes in the state of the knowledge base

Production rules allow a reactive or event-driven style of programming



Example Method

```
(defmethod assess-casualty (?medic ?casualty)
  :situation
    (:and (Medic ?medic) (casualty ?casualty))
  :response
    ((format t "~%Medic ~8A examines ~8A"
              ?medic ?casualty)
     (tell (examined ?casualty 'yes)))
)
```



Example Method

```
(defmethod assess-casualty (?medic ?casualty)
  :situation
    (:and (Medic ?medic) (casualty ?casualty))
  :response
    ((format t "~%Medic ~8A examines ~8A"
              ?medic ?casualty)
     (tell (examined ?casualty 'yes)))
)
```

Query determines applicability



Example Method

```
(defmethod assess-casualty (?medic ?casualty)
  :situation
    (:and (Medic ?medic) (casualty ?casualty))
  :response
    ((format t "~%Medic ~8A examines ~8A"
              ?medic ?casualty)
     (tell (examined ?casualty 'yes)))
)
```

Query determines applicability

Lisp code in the response



Example Method

```
(defmethod assess-casualty (?medic ?casualty)
  :situation
    (:and (Medic ?medic) (casualty ?casualty))
  :response
    ((format t "~%Medic ~8A examines ~8A"
              ?medic ?casualty)
     (tell (examined ?casualty 'yes)))
)
```

Query determines applicability

Lisp code in the response

Loom assertions in the response

Methods Can Be Performed Immediately or Scheduled

To call a method immediately use the “perform” function

To schedule a method for execution use the “schedule” function

Scheduled methods can be given a priority (the built-in priorities are :high and :low)

Methods are performed the next time there is a knowledge base update (ie, “tellm”)

Methods are executed in accordance with the priority

Within a priority methods are executed in the ordered they were scheduled



The :situation Determines Method Applicability



```
(defmethod treat-patient (?medic ?patient)
  :situation (:and (medic ?medic)
                   (critical-casualty ?patient)
                   (examined ?patient 'no))

  :response
  ((schedule (goto ?medic ?patient)
              :priority :high)
   (schedule (assess-casualty ?medic ?patient)
              :priority :high)))

(defmethod treat-patient (?medic ?patient)
  :situation (:and (medic ?medic)
                   (non-critical-casualty ?patient)
                   (examined ?patient 'no))

  :response
  ((schedule (goto ?medic ?patient)
              :priority :low)
   (schedule (assess-casualty ?medic ?patient)
              :priority :low)))
```

More on Choosing a Method

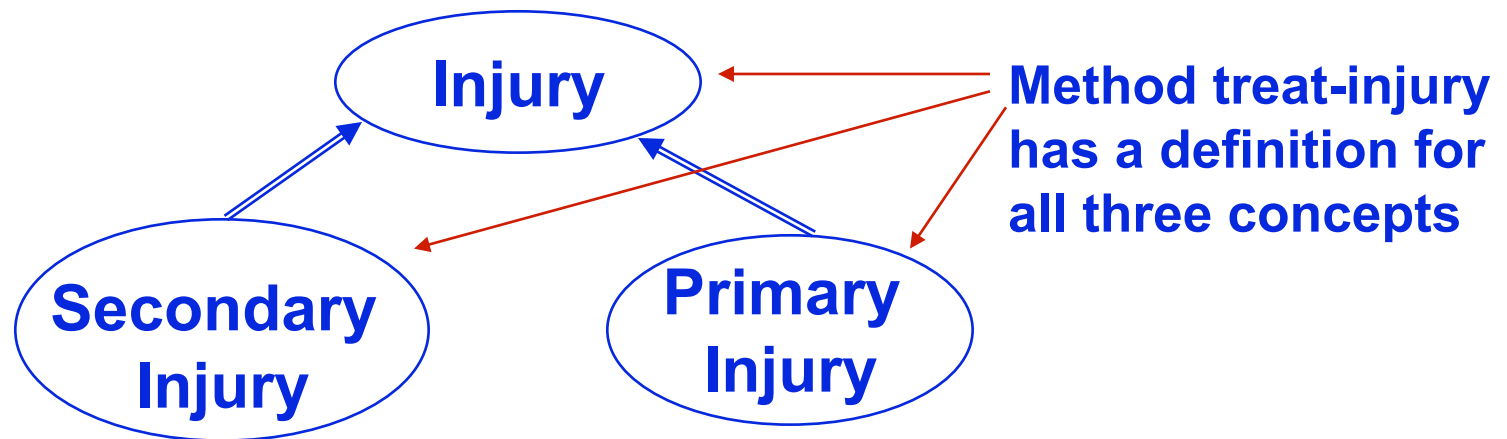
Often several methods are applicable to a particular situation. “defaction” forms can specify how to resolve ambiguities

- *Choose all applicable methods*
- *Choose the most specific method*
- *Choose the last method defined*
- *Choose a method at random*
- *Issue a warning*
- *Cause an error*

These resolution methods can be combined and are used in order



Example of Combined Resolution



If both secondary and primary injuries exist, :most-specific does not give a single result

Multiple selection criteria resolves the problem

```
(defaction treat-injury (?medic ?patient)
  :filters (:most-specific :select-all))
```

The criteria are prioritized

Avoids the need to define methods for all combinations of concepts

Methods Can Also Have Query-Based Iteration

Finding all casualties reported on Medic's clipboard

```
(defmethod locate-casualties (?medic)
  :situation (medic ?medic)
  :with (casualties
         (clipboard ?medic) ?c)
  :response (...))
```

*The response is executed once for each ?c
that the query in the :with clause finds.*

*In the response ?medic is bound to the
method argument and ?c to a particular
casualty reported on the medic's clipboard.*



Production Rules Trigger on Changes in the Knowledge Base

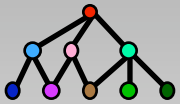
*The changes can be additions to the KB
(:detects)*

This applies to relation additions and concept additions

*The changes can be deletions from the KB (
(:undetecteds)*

This applies to relation deletions and concept deletions

*The change can be in a relation value
(:changes)*



Noticing a New Injury

```
(defproduction notice-injury
  :when ((:and (:detects (injury ?self ?i))
                (phone ?i ?phone)))
  :do ((perform (report-injury ?phone ?i)))
```

The :detects clause triggers the production

The additional query (phone ?i ?phone) is a guard clause and also provides an additional variable binding

The variables from the :when clause are bound for the execution of the production body. In this example, the injury is reported using a phone by calling the method “report-injury”.

A different method could be used if a radio were available.



Extended Example

Hospital Knowledge Base

Definitions and Queries



Extended Query Example: Hospital Knowledge Base

```
(defconcept facility)
(defconcept hospital :is
  (:and facility
    (:at-least 1 ward-capacity)))
```

```
(defrelation ward-capacity
  :domain hospital)
```

```
(tell (:about h-1
  (ward-capacity 120)
  (ward-capacity 120)
  (ward-capacity 100)))
```

```
(tell (:about h-2
  (ward-capacity 110)
  (ward-capacity 90)))
```



Is H-1 a Hospital?

```
(tell (:about h-1
      (ward-capacity 120)
      (ward-capacity 120)
      (ward-capacity 100)))
```



Is H-1 a Hospital?

```
(tell (:about h-1
      (ward-capacity 120)
      (ward-capacity 120)
      (ward-capacity 100)))
```

Yes, for classified instances,
because of the :domain entry in

```
(defrelation ward-capacity
  :domain hospital
  :range ...)
```



How Many Wards for H-1?

```
(tell (:about h-1
      (ward-capacity 120)
      (ward-capacity 120)
      (ward-capacity 100)))
```



How Many Wards for H-1?

```
(tell (:about h-1
      (ward-capacity 120)
      (ward-capacity 120)
      (ward-capacity 100)))
```

```
(pi h-1) ==>
```

```
(:ABOUT H-1...
  (WARD-CAPACITY 120)
  (WARD-CAPACITY 100))
```

ONLY 2!



What Does This Query Ask?

```
(retrieve (?x ?y)
  (> (ward-capacity ?x)
    (ward-capacity ?y)))
```



What Does This Query Ask?

```
(retrieve (?x ?y)
  (> (ward-capacity ?x)
      (ward-capacity ?y)))
```



Implicit :for-some wrapped around
query, therefore returns:

```
((|I|H-1 |I|H-1) (|I|H-1 |I|H-2)
  (|I|H-2 |I|H-1) (|I|H-2 |I|H-2))
```

What Is Wrong with This?

```
(defrelation ward-capacity) ; no :domain
```

```
(retrieve (?x ?y)  
  (> (ward-capacity ?x)  
      (ward-capacity ?y)))
```



What Is Wrong with This?

```
(retrieve (?x ?y)
  (> (ward-capacity ?x)
      (ward-capacity ?y)))
```



Performance Warning: Query scans the entire knowledge base to generate bindings for the variables ?X and ?Y.

Query time solution:

```
(retrieve (?x ?y)
  (:and (hospital ?x)
        (hospital ?y)
        (> (ward-capacity ?x)
            (ward-capacity ?y))))
```


Find Hospitals Ordered by Their Largest Wards

```
(defrelation ward-capacity  
  :domain hospital)
```

```
(retrieve (?x ?y)  
  (:and (> (max (ward-capacity ?x))  
           (max (ward-capacity ?y)))))
```

```
==> ((|I|H-1 |I|H-2))
```



What About All Wards Larger?

```
(retrieve (?x ?y)
  (:and (> (min (ward-capacity ?x))
            (max (ward-capacity ?y)))))
```

==> NIL



Hospital with a Ward Larger Than 100 beds?

```
(retrieve (?x)
  (:for-some (?len)
    (:and (ward-capacity ?x ?len)
      (>= ?len 100))))
```



Note the explicit :for-some designation!

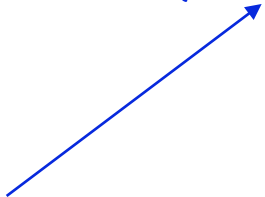

Hospital with All Wards Larger Than 100?

```
(retrieve (?x)
  (:for-all (?len)
    (:implies
      (ward-capacity ?x ?len)
      (>= ?len 100))))
```



Special Syntax in :for-all

```
(retrieve (?x)
  (:for-all (?len)
    (:implies
      (ward-capacity ?x ?len)
      (>= ?len 100))))
```



Implication used in :for-all to restrict the domain of the quantified variable (?len)

Alternate possibility:

```
(...(:for-all (?len)
  (:or
    (not (ward-capacity ?x ?len))
    (>= ?len 100))))
```

Implication Equivalence

(:implies A B)



(:or (:not A) B)



Hospital with All Wards Larger Than 100?

```
(retrieve (?x)
  (:for-all (?len)
    (:implies
      (ward-capacity ?x ?len)
      (>= ?len 100))))
```

**Problem: Couldn't find a closed set of
fillers for the role ward-capacity.**



Three Possible Solutions

At the individual level:

```
(tell (:about h-1
          (:exactly 2 ward-capacity)))
```



At the relation level:

```
(defrelation ward-capacity ...
  :characteristics :closed-world)
```

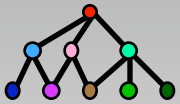
At the context level:

```
(setf (open-closed-mode
      (current-context))
      :closed)
```


Hospital with All Wards Larger Than 100?

```
(retrieve (?x)
  (:and (hospital ?x)
    (:for-all (?len)
      (:implies
        (ward-capacity ?x ?len)
        (>= ?len 100)))))
```

$\Rightarrow (|I|H-1)$



Nested Queries Are OK

```
(retrieve (?x ?y)
  (:and (hospital ?x) (hospital ?y)
    (:for-all (?a)
      (:implies
        (ward-capacity ?x ?a)
        (:for-some (?b)
          (:and (ward-capacity ?y ?b)
            (> ?a ?b)))))))
```

$\Rightarrow ((|I|_{H-1} \ |I|_{H-2}))$

How To Get Multiple Wards of the Same Size for H-1?

```
(tell (:about h-1
      (ward-capacity 120)
      (ward-capacity 120)
      (ward-capacity 100)))
```



Need to make wards individuals,
so they can be differentiated.

New Domain Model

```
(defconcept facility)
(defconcept hospital :is
  (:and facility
    (:at-least 1 hospital-ward)))
```

```
(defconcept ward :is
  (:and facility
    (:exactly 1 ward-capacity)))
```

```
(defrelation hospital-ward
  :domain hospital :range ward
  :characteristics :closed-world)
```

```
(defrelation ward-capacity
  :domain ward
  :characteristics :closed-world)
```



Auxiliary Relation

```
(defrelation hospital-ward-capacity
  :is (:compose hospital-ward
                 ward-capacity))
```



Domain Facts

```
(tell (:about h-1
        (hospital-ward w1)
        (hospital-ward w2)
        (hospital-ward w3)))
```

```
(tell (ward-capacity w1 120)
      (ward-capacity w2 120)
      (ward-capacity w3 100))
```

```
(tell (:about h-2
        (hospital-ward w4)
        (hospital-ward w5)))
```

```
(tell (ward-capacity w4 110)
      (ward-capacity w5 90))
```



Retrieve Multiple Wards for H-1

```
(retrieve (?w ?l)
  (:and (hospital-ward h-1 ?w)
    (ward-capacity ?w ?l)))
```



```
==> ((|I|w1 120) (|I|w2 120) (|I|w3 100))
```

Retrieve Multiple Wards for H-1

```
(retrieve (?w ?l)
  (:and (hospital-ward h-1 ?w)
    (ward-capacity ?w ?l)))
```



```
=> ((|I|W1 120) (|I|W2 120) (|I|W3 100))
```

What about a short-hand notation?

```
(retrieve ?l
  (hospital-ward-capacity h-1 ?l))
```

```
=> (120 100)
```


Lessons from the Example

Modeling Advice:

Determine Detail Level

Use Specialized Operators

Be Explicit in Queries



Procedural Programming

Outline of talk:

- Deductive Kb with Multiple Paradigms
- Production rules
- Methods
- Lisp-to-Loom Interface
- Interpretations of Updates



Multiple Paradigm Programming

Idea: Suite of programming paradigms that each exploit a dynamically changing deductive knowledge base.



Loom paradigms:

Data driven	(production rules, monitors)
Methods	(pattern-directed dispatch)
Procedural	(Lisp)

Upgrading Traditional Paradigms

```
(defproduction P1
  :when (:detects (Foo ?x))
  :do ((print "New Foo")))

(defmethod M1 (?self)
  :situation (Foo ?self)
  :response ((print "It's a Foo all right")))
```

Innovations:

- "Foo" can expand to an arbitrarily complex description;
- "Edge-triggered" productions;
- Pattern-based method dispatching.



Production Rule Semantics

```
(defproduction <name>
  :when <condition> :perform <action>)
```

Semantics: Whenever a set of variable bindings in *<condition>* becomes true (provable), call *<action>* with that set of bindings.

Example:

```
(defproduction P2
  :when (and (Switch ?s)
              (:detects (status ?s 'on)))
  :perform (turn-on (appliance-of ?s)))
```

The *:when* condition of a production must include at least one of the transition operators *:detects*, *:undetecteds*, or *:changes*.



Semantics of :detects

(:detects (A ?x))

is defined as

(and (A ?x)

(:previously (:fail (A ?x))))

(:previously (B ?x))

is defined as

*(:at-agent-time (- *now* 1)*

(B ?x)))



Semantics of Detects (cont.)

`(:detects (:and (A ?x) (B ?x)))`

*will trigger if A and B become true simultaneously
or if A becomes true and B is already true
or if B becomes true and A is already true*

`(:and (:detects (A ?x))
(:detects (B ?x)))`

*will trigger only if A and B become true
simultaneously*



Production Rule Semantics (cont).

All production rule instantiations at the end of an update cycle are fired in parallel.

- No conflict resolution (this is a feature!)
- Effects of one production cannot inhibit firing of another (parallel) production.

Rationale:

- We want productions to be “well-behaved” (no race conditions);
- Preference semantics is the province of the method paradigm.

Division of responsibility:

- Production determines when to perform task;
- Method determines how to perform task.



Task Scheduling

Productions can post tasks on a queue rather than executing them immediately.



```
(defproduction P5
  :when (and (:changes (home-team-score ?game))
             (basketball-game ?game))
  :schedule (celebrate)
  :priority :low)
(defproduction P6
  :when (and (:changes (home-team-score ?game))
             (football-game ?game))
  :schedule (celebrate)
  :priority :high)
```

Monitors

Monitors are productions that fire only when specifically designated instances undergo property transitions.



```
(defmonitor Watch-for-Redraw
  :when (or (:changes (color ?object))
            (:changes (size ?object)))
  :do ((redraw (slot-value ?object 'window)))
  (tellm (color Thing5 'Red)))
```

nothing happens

```
(attach-monitor 'Thing' Watch-for-Redraw)
(tellm (color Thing5 'Green'))
```

calls redraw

Monitors generalize the active value paradigm

Methods

defaction: Defines Loom equivalent of "generic function".

defmethod: Defines procedurally-invoked situation-response rule.

```
(defmethod <name> (<parameters>)  
  :situation <situation>  
  :response <response>)
```



Method Filters

Most frequent modes of method use. Given a call to invoke an action M:

- (1) execute all methods named M whose situations are satisfied, or
- (2) execute the most specific among those methods named M whose situations are satisfied.

A ``filter sequence'' determines the criteria for choosing which methods to fire (among those that are eligible).



Method Filters Example

```
(defaction M2 (?x ?y) :filters (:perform-all))
(defmethod M2 (?x ?y)
  :situation (= ?x ?y)
  :response ((print "EQ")))
(defmethod M2 (?x ?y)
  :situation (<= ?x ?y)
  :response ((print "LE")))
```

```
(perform (M2 3 4))
--> "LE"
```

```
(perform (M2 4 4))
--> "LE"
    "EQ"
```

both methods fire

```
(defaction M2 (?x ?y) :filters (:most-specific))
(perform (M2 4 4))
--> "EQ"
```

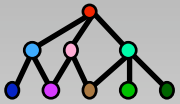
only the most specific method fires



Tuning for Performance

■ *Outline of talk:*

- *Classifier Performance*
- *Recognizer Performance*
- *Performance Tips*
- *CLOS Instances and the Backchainer*



Performance

Where does the time go?

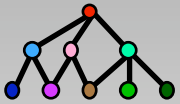


- *In some systems, slow performance is due to poorly-tuned code.*
- *In Loom, slow performance can result from the enormous amount of inferencing that occurs under the hood.*

Classifier Performance

■ *Classifier Phases*

- (1) normalization (compute closure of ~100 inference rules)*
- (2) classification (compute subsumption links — very fast)*
- (3) completion (normalize constraints)*
- (4) sealing (compile access functions)*



Classifier Performance

■ *Classifier Phases*

(1) normalization

(2) classification

(3) completion

(4) sealing

■ *Bulk of time is spent in phases (1) and (3), normalizing features:*

- (i) start with local features (:at-most, :at-least, :all, ...);*
- (ii) inherit features from parent concepts;*
- (iii) compute larger set of features (deductive closure);*
- (iv) keep only the most specific features;*
- (v) classify the remaining features.*



Speeding Up Normalization



- *Each constraint in Loom represents a rule of inference (not just a type check).*
- *The overhead of normalization depends on the number of features per concept (it's estimated to be quadratic in the number of features).*
- *So, a simple way to speed up an application is to specify fewer constraints :-).*

Speeding Up Normalization (cont.)



- *Loom permits you to lobotomize the classifier*
 - *“(power-level :medium)” causes Loom to ignore a few of the most expensive normalization rules.*
 - *“(power-level :low)” causes Loom to make a single pass over the normalization rules (rather than computing their closure).*

Load-Time vs. Run-Time Classification

- *Most applications perform the bulk of classification at load time; for them, speed of classification may not be critical.*
 - *Normally, run-time production of new system-generated descriptions will quiesce (no more “.”s and “+”s);*



Recognizer Performance

- *An explicit call by an application (e.g., (tellm)) triggers reclassification of updated instances .*
- *Recognition strategy:*
 - *For each instance on the queue*
 - (1) normalize asserted and inherited features;*
 - (2) classify the instance;*
 - (3) install dependency bombs (TMS monitors);*
 - (4) test for incoherence;*
 - (5) propagate forward constraints.*
- *Steps 1-5 are applied to each instance at least two times (once each in strict and default mode).*



Classifying Instances

During the recognition process, each feature in a concept definition represents a miniature query.

Examples:

(:at-least k R)

Retrieve fillers of the role R;

Succeed if the number of fillers is at least k.

(:at-most k R)

If role R is closed, retrieve fillers of the role R;

Succeed if the number of fillers is at most k.

(:all R A)

If role R is closed, retrieve fillers of the role R;

Succeed if each of the fillers satisfies the concept A.

The bulk of recognition time consists of computing feature satisfaction *and* truth maintaining the results.



Testing for Closed Roles

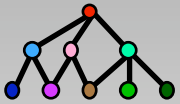


- *Probing features such as $(:all\ R\ A)$ or $(:at-most\ k\ R)$ usually entails proving that the role R is closed.*
- *This test is fast if*
 - *R has the $:closed-world$ property, or*
 - *R is $:single-valued$ and a role filler exists.*
- *Tip : Always specify the $:single-valued$ and $:closed-world$ properties on relations whenever they are valid for your application domain.*

Subtlety in the semantics of role closure:

```
(defconcept A
  :implies (:at-least 1 R))
(defrelation R
  :characteristics (:closed-world))
(tell (Thing Joe)
      (A Fred))
```

- *The role “(R of Joe)” is closed, but the role “(R of Fred)” is not closed.*



Domain and Range Constraints

- Tip : Always specify domain and range constraints for a relation (unless they are inherited from a parent relation).



`(defrelation R :domain A :range B)`

`(tellm (R Fred Joe))`

➔ *Loom infers that Fred satisfies A and that Joe satisfies B.*

`(defconcept A :implies (:exactly 1 R))`

`(defrelation R :domain A)`

➔ *Loom infers that R is :single-valued.*

Performance Warnings



- A *`no generator found'* performance warning indicates that a query will exhibit abysmal performance.

- *Slower (sometimes) :*

`(retrieve (?x ?y) (R ?x ?y))`

- *Faster (sometimes) :*

`(retrieve (?x ?y)`

`(and (A ?x) (R ?x ?y)))`

- *If no domain is specified for R, the slower query will scan the entire kb to generate bindings for ?x.*

Performance Tips:

- Tip : Always rephrase definitions or queries to eliminate performance warnings.
- Tip : Never wrap an eval around an ask or retrieve unless you are single, childless, and have no desire to graduate, e.g.,
 - (eval `(retrieve (?y) (and (R ,foo ?y) (A ?y))))
- Tip: To programmatically compose a query on the fly, use “query” or bind variables:
 - (query ‘(?y) `(and (R ,foo ?y) (A ?y)))
 - (let ((?x foo))
(retrieve (?y) (and (R ?x ?y) (A ?y))))

Better!



:perfect relations

- *Marking a concept or relation :perfect tells Loom that facts about it cannot change.*

- Tip: *Use of the :perfect properties reduces match overhead.*
- Tip: *Computed relations are prime candidates for the :perfect attribute .*

(defrelation <>

 :domain Number :range Number

 :characteristics (:symmetric :perfect)

 :predicate /=)



How to Get No Recognition



- *The overhead of instance classification (recognition) is eliminated if you specify as a creation policy :clos-instance or :lite-instance.*
- *Deduction over CLOS instances and LITE instances is backward chained, with no caching.*
- *However (there is always a catch) inference without instance classification is strictly weaker than inference with it.*

Deduction with CLOS and LITE Instances

- *With creation policy set to :clos-instance or :lite-instance inference is performed using backward chaining.*



- *The backchainer recognizes rules of the form*
(implies A B) and
(implies <description> B)
but ignores rules of the form
(implies A <descriptions>)

Backward chaining and type restrictions

- *The design decision not to chain backwards across value restrictions was a judgment call.*

```
(defconcept A
  :implies (:all R B))
(tell (A Fred) (R Fred Joe))
(ask (B Joe))  --> ???
```

- *The recognizer will prove that Joe satisfies B; the backchainer won't.*



Using Time in Loom

Thomas A. Russ

USC

Information Sciences Institute

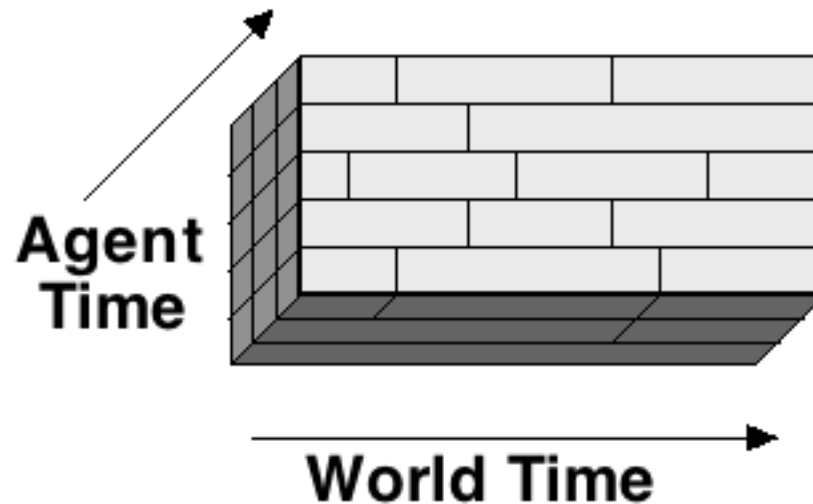


Outline

- *Time Representation*
- *Basic Assertions*
- *Basic Queries*
- *Persistence*
- *Time and the Classifier*
- *Advanced Examples*



Agent and World Time



- *World Time Records Domain Facts*
- *Agent Time Records Knowledge Base Changes*

Time Representation

■ *Definite Times*

- *Integers*
- *Time Strings “10/28/94 11:33”*

■ *Anchored to Calendar*

- *Common Lisp universal time*

■ *Points Are Basic Units*

■ *Intervals Are Derived*

■ *“Property” Interpretation of Intervals*



Properties and Events

■ *Properties*

- *True over all subintervals*
- *“The house is red”*

■ *Events*

- *True only over the entire interval*
- *“John ran completely around the track.”*



Basic Assertions

■ *Transitions Only*

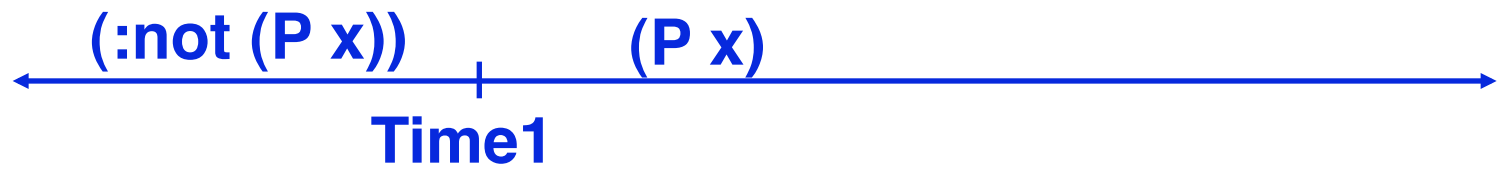
- *(:begins-at time-point assertion)*
- *(:ends-at time-point assertion)*

■ *Strong Temporal Assertion*

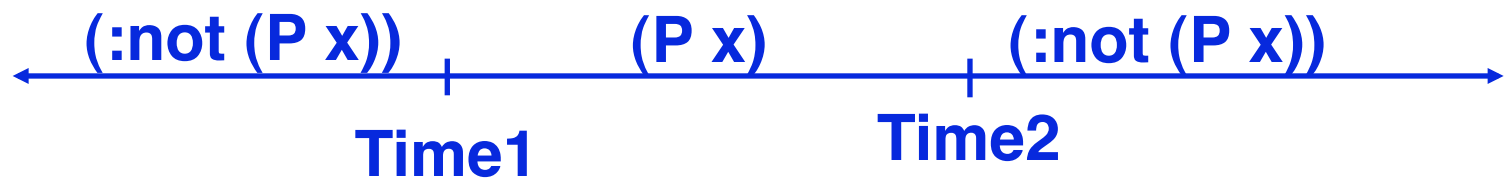
- *Before :begins-at, assertion is false.*
- *After :begins-at, assertion is true.*



Basic Assertions



`(tell (:begins-at Time1 (P x)))`



`(tell (:ends-at Time2 (P x)))`

Basic Queries—Transitions

■ *Transitions:*

- *(ask (:ends-at t1 (P x)))*



USCIS

- **States:**

- *(ask (:holds-at t1 (P x)))*

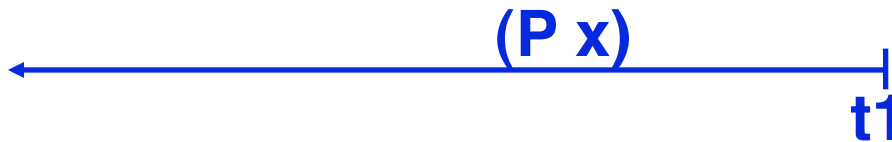
Basic Queries—States Problem

■ Transitions:

- *(ask (:ends-at t1 (P x)))*

■ States:

- *(ask (:holds-at t1 (P x)))*
- *But this can be ill-defined*



Basic Queries—States Solution

■ Introduce Directional Operators

- `(ask (:holds-before t1 (P x)))`
- `(ask (:holds-after t1 (P x)))`

■ Yields well-defined results:

$(P\ x)$

←—————|
t1

<code>:holds-before</code>	<code>==></code>	<code>t</code>
<code>:holds-after</code>	<code>==></code>	<code>nil</code>



Non-Transitional Assertions

■ *Persistence Only*

- *(:holds-after time-point assertion)*
- *(:holds-before time-point assertion)*

■ *Weak Temporal Assertion*

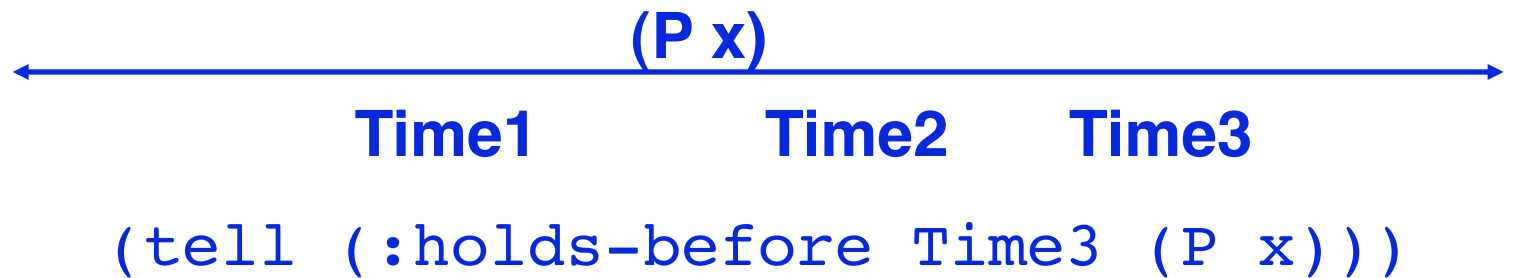
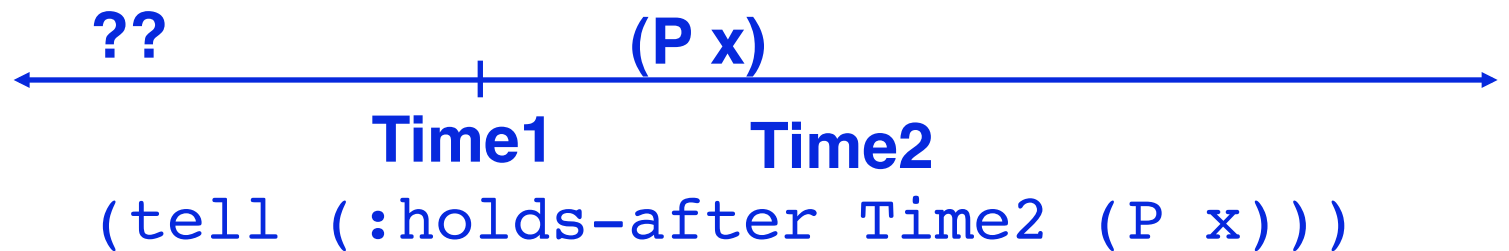
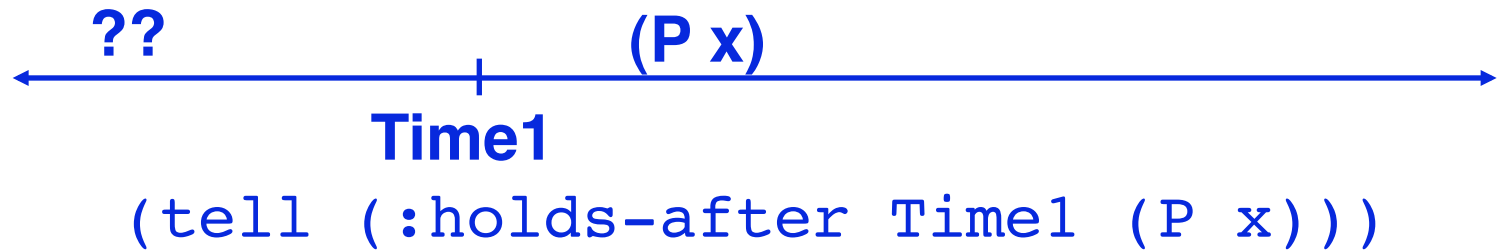
- *Before :holds-after, assertion can be true or false.*
- *After :holds-before, assertion can be true or false.*

■ *:holds-at is the combination of :holds-before and :holds-after*

- *The assertion is true both before and after a :holds-at*



Persistence Assertions



Temporal Operator Truth Table



	$(P\ x)$		
	t1	t2	t3
■ <i>:begins-at</i>	t	nil	nil
■ <i>:holds-after</i>	t	t	nil
■ <i>:holds-at</i>	nil	t	nil
■ <i>:holds-before</i>	nil	t	t
■ <i>:ends-at</i>	nil	nil	t

Changes to Classifier

■ *Classifier Is Time Sensitive*

- *Temporal information in the ABox affects classification*



■ *Definitions Are Time Invariant*

- *TBox definitions hold over the entire time line*

Bachelor Example

```
(defconcept Married  
  :characteristics :temporal)
```

```
(defconcept Bachelor :is  
  (:and Male (:not Married)))
```

```
(tell (Male p1)  
  (:begins-at t1(Married p1)))
```

(Male p1)

(:not (Married p1)) | (Married p1)

t1

(Bachelor p1)

t1

Widow Example

```
(defconcept Dead
  :characteristics :temporal)
```

```
(defrelation husband
  :is (:and spouse (:range Male))
  :characteristics :temporal)
```

```
(defconcept widow
  :is (:and Female
        (:some husband Dead)))
```



Widow Assertions

```
(tellm (Female Mary) (Male John))
```

```
(tellm (:begins-at "1/1/90"  
                  (spouse Mary John))  
       (:begins-at "1/1/94"  
          (Dead John)))
```

(Male John)

(Female Mary)

(spouse Mary John)

(Dead John)

1/1/90

1/1/94



Widow Derivation

```
(tellm (Female Mary) (Male John))
```

```
(tellm (:begins-at "1/1/90"  
                  (spouse Mary John))  
      (:begins-at "1/1/94"  
        (Dead John)))
```

(Male John)

(Female Mary)

(spouse Mary John)

(Dead John)

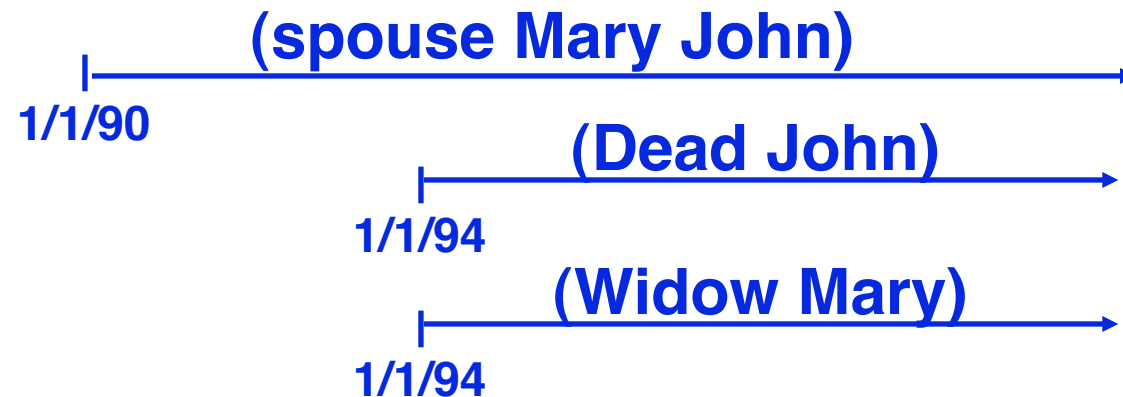
(Widow Mary)

1/1/90

1/1/94

1/1/94

Widow Queries



```
(retrieve ?x (:holds-at "10/28/94"  
                        (widow ?x)))
```

```
=> (|i|Mary)
```

```
(retrieve ?x (:begins-at ?x  
                        (Widow Mary)))
```

```
=> (2966400000)  
; = "1/1/94 00:00:00"
```


Former Hockey Player

```
(defconcept former-hockey-player :is
  (:and person
    (:satisfies (?p)
      (:for-some (?t)
        (:and (past ?t)
          (:ends-at ?t
            (hockey-player ?p
              ))))))))
```



Former Hockey Player

```
(defconcept former-hockey-player :is
  (:and person
    (:satisfies (?p)
      (:for-some (?t)
        (:and (past ?t)
          (:ends-at ?t
            (hockey-player ?p
              ))))))))
```



- Temporal concept “**past**” constrains matches for ?t to occur before the time this definition is satisfied.
- A former hockey player is “someone who ceased to be a hockey player sometime in the past.”



Former Hockey Player

Temporal Clause

```
(defconcept former-hockey-player :is
  (:and person
    (:satisfies (?p)
      (:for-some (?t)
        (:and (past ?t)
          (:ends-at ?t
            (hockey-player ?p
              ))))))))
```

- *Temporal relation to the concept “hockey-player” established.*



Former Hockey Player Assertion and Queries

```
(tellm (Person Fred))
```

```
(tellm (:ends-at "1/1/90"  
                (hockey-player Fred)))
```



```
(ask (:holds-at "1/1/88"  
               (hockey-player Fred))) => T
```

```
(ask (:holds-at "1/1/88"  
       (former-hockey-player Fred))) => NIL
```

```
(ask (:holds-at "1/1/94"  
       (hockey-player Fred))) => NIL
```

```
(ask (:holds-at "1/1/94"  
       (former-hockey-player Fred))) => T
```

Summary

- *World and Agent Time Supported*
- *Definite, Calendar-Anchored Time*
- *ABox Supports Temporal Assertions*
- *Inference Is Time Sensitive*



Miscellaneous Issues

- *Frame Functions*
- *Saving and Restoring*
- *CLOS Classes*
- *Mix and Match Inferencing*



Frame Functions: Concepts



add-type <instance> <concept>

"assert that <instance> ISA <concept> "

get-types <instance>

"retrieve all concepts satisfied by <instance>"

Frame Functions: Setting Role Fillers



add-value <instance> <role> <filler>

"add <filler> to the set of fillers of role <role> on instance <instance>"

set-value <instance> <role> <filler>

"set <filler> to be the only filler of role <role> on instance <instance>"

only recommended for single-valued roles

**set-values <instance> <role>
<list-of-fillers>**

"make each filler in <list-of-fillers> be a filler of role <role> on instance <instance>"

Always requires a list, even for single valued roles.

Frame functions

get-value <instance> <role>

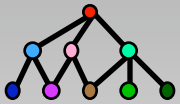
"retrieve the single filler of role <role> on instance <instance>"

Error if there is more than one value.

get-values <instance> <role>

"return the set of fillers of role <role> on instance <instance>"

Always returns a list of fillers.



Saving and Restoring

- *Knowledge Bases and contexts can be saved to files*

```
(save-kb [<kbName>]  
         :pathname <filename>)
```

```
(save-context contextName  
              :pathname <filename>)
```

- *The save files can be compiled and loaded into Loom images to restore the state of the knowledge base.*



CLOS Classes

Execute the following:

```
(creation-policy :clos-instance)
(defconcept Ship
  :roles ((name :type String)
          length))
(create nil 'Ship)
```

Side-effects:

```
(defrelation name)
(defrelation length)
(eval '(defclass Ship (THING)
        ((name :initform nil)
         (length :initform nil))))
(make-instance 'Ship)
```



Mix and Match Inferencing

■ **INSTANCE-IN-CONTEXT**

at creation time, adds an instance to the concept-instance index

■ **INSTANCE-WITH-CONCEPTS**

permits more than one type to be asserted on an instance

■ **INSTANCE-WITH-INVERSES**

automatically adds and removes inverse links in response to slot updates

■ **INSTANCE-WITH-DYNAMIC-SLOTS**

non-preallocated slots use alist storage on an instance



Mix and Match Inferencing (cont.)

■ **INSTANCE-WITH-NEGATION**

supports negated type and negated role filler assertions

■ **INSTANCE-WITH-HISTORIES**

record (in a differential history) the prior states of an instance

■ **INSTANCE-IN-MATCH-NETWORK**

instance participates in matches that trigger production rules

■ **INSTANCE-WITH-TIME**

supports temporal assertions



Frequently Asked Questions



- *Why Don't Instances Get Recognized?*
- *Use of “:all”*
- *Use of “:for-all”*
- *Compiling Loom Code*
- *Combining Number Restrictions*
- *Why Doesn't (:exactly 1 R) Clip?*
- *Why Aren't Concepts Disjoint?*
- *My Concept Name Changed!*
- *Multiple Value Roles & Defaults*
- *Inverse Relations*
- *Loom vs. CLOS*

FAQ:

Why Aren't Instances Recognized?



- *Why don't instances get recognized as belonging to a concept when I assert them?*
 - *Time needs to be advanced:*
Use (tellm) or (new-time-point)
 - *Lite instances are being used instead of classified instances:*
Use (creation-policy :classified-instance)
- *How do I tell if I have classified or lite instances?*
 - *Use the function (creation-policy).*
 - *Subtle: Look at the printed representation:*

Note case
of letter "i"

| i | Fred
| I | Barney

is a lite instance;
is classified

FAQ:

Use of “:all”

■ *Value restrictions using :all*

- (defrelation R)
- (defconcept C)
- (defconcept C-all
:is (:and C (:all R C)))

■ *Assertions*

- (tell (C c1) (C c2) (R c2 c1))

■ *Query*

- (retrieve ?x (C-all ?x))



FAQ:

Use of “:all”



■ Value restrictions using :all

- (defrelation R)
- (defconcept C)
- (defconcept C-all
:is (:and C (:all R C))

■ Assertions

- (tell (C c1) (C c2) (R c2 c1))

■ Query

- (retrieve ?x (C-all ?x)) ==> NIL

■ Why NIL? Because R is not closed, therefore other unknown R fillers could exist which are not Cs.

FAQ:

Use of “:all”

■ Value restrictions using :all

- (defrelation R
:characteristics :closed-world)
- (defconcept C)
- (defconcept C-all
:is (:and C (:all R C)))

■ Assertions

- (tell (C c1) (C c2) (R c2 c1))

■ Query

- (retrieve ?x (C-all ?x)) ==> (c1 c2)

■ Why both of them? How can all of c1's R fillers be Cs if c1 doesn't have any Rs? Since there are no such fillers, it is trivially fulfilled.



FAQ:

Use of “:all”



■ Value restrictions using :all

- (defrelation R
:characteristics :closed-world)
- (defconcept C)
- (defconcept C-all
:is (:and C (:all R C)
(:at-least 1 R))

■ Assertions

- (tell (C c1) (C c2) (R c2 c1))

■ Query

- (retrieve ?x (C-all ?x) ==> (c2))

■ The :at-least 1 restriction expresses what we really mean!

FAQ:

Proper use of “:for-all”



- *Loom’s universal quantification is does not have a type restriction built in. The consequence is that special syntax is needed inside :for-all constructs*

- `(defconcept C)`
 `(defrelation R)`
- `(retrieve ?x`
 `(:for-all (?z)`
 `(:and (R ?x ?z) (C ?z))))`

- *This will produce an error message*

- To successfully evaluate a universally quantified clause, the clause must contain at least one negated term. In this case, the clause
 `(|R|R ?X ?Z)`
 does not.

FAQ:

Proper use of “:for-all”

■ Logically speaking, the query

- (retrieve ?x
 (:for-all (?z)
 (:and (R ?x ?z) (C ?z))))

is extremely unlikely to be satisfied if ?z ranges over all individuals in the knowledge base. The query must be formulated to restrict the value of ?z

- (retrieve ?x
 (:for-all (?z)
 (:implies (R ?x ?z) (C ?z))))

or equivalently

- (retrieve ?x
 (:for-all (?z)
 (:or (:not (R ?x ?z)) (C ?z))))



FAQ:

Compiling Loom Code



- *Loom performs code generation and optimization during macro-expansion of the forms “tell”, “forget”, “ask” and “retrieve”*
- *The proper expansion of the code requires that all definitions referenced in the form be available*
 - *Definition files must therefore be loaded before assertion or query files are compiled*
 - *If definitions are in the same file, then they must be enclosed by an “eval-when” form specifying compile time evaluation. The last form in the eval-when should be a call to “finalize-definitions”*

FAQ:

Compiling Loom Code

- *Certain redefinitions (such as changing a relation from single to multipleneral rule, all code which uses definitions should be recompiled when those definitions change.*



- USCIS

- ```
(defrelation R)
(defconcept C)
(defconcept -C :is (:not C))
(defconcept A
 :is (:and C (:at-least 2 R C)
 (:at-least 2 R -C)))
```
- ```
(tell (A a1))
```
- ```
(ask (:about a1 (:at-least 2 R))) ==> T
(ask (:about a1 (:at-least 4 R))) ==> T
(ask (:about a1 (:at-least 5 R))) ==> NIL
```

- **Loom knows  $C$  and  $\neg C$  are disjoint, so there must be at least 4 fillers of  $R$  on any  $A$ .**

# FAQ:

## Combining Number Restrictions

■ *Inference is not complete in all cases*

■ *Example*

- (defrelation R)  
(defconcept C)  
(defconcept -C :is (:not C))  
(defconcept A  
:is (:and C (:at-least 2 R C)  
(:at-most 3 R)))
- (tell (A a1))
- (ask (:about a1 (:at-most 3 R))) ==> T  
(ask (:about a1 (:at-most 3 R C))) ==> T  
**WRONG!** (ask (:about a1 (:at-most 1 R -C))) ==> NIL  
(ask (:about a1 (:at-most 3 R -C))) ==> T

■ *Loom cannot infer the upper limit on -C fillers based on the upper limit on R and the lower limit on fillers of type C*



# *FAQ:*

## *Why Doesn't (:exactly 1 R) Clip?*

### ■ *Number restriction in concept definition*

- (defrelation R)
- (defconcept C  
:is-primitive (:exactly 1 R))

### ■ *Assertions*

- (tell (C c1) (R c1 3))
- (tell (R c1 4))

### ■ *Query*

- (retrieve ?x (R c1 ?x)) ==> (3 4)

### ■ *The assertion of C and of the two role fillers have equal weight. There is no logical preference for one over the other.*



# *FAQ: Why Doesn't (:exactly 1 R) Clip?*

- *To get clipping the relation itself must be asserted to be single-valued:*
  - (defrelation R  
:characteristics :single-valued)
- *Or Loom must be able to infer that R must be single-valued:*
  - (defrelation R :domain C)
  - (defconcept C  
:is-primitive (:exactly 1 R))
  - *Since the domain of R is C all instances that have R fillers must also be of type C. Since C only has 1 R, R must be single-valued.*



# FAQ:

## *Why Aren't Concepts Disjoint?*

### ■ *Example*

- (defrelation R :attributes :closed-world)  
(defconcept A)  
(defconcept B)  
(defconcept C :is (:and A (:all R A)))
- (tell (A a1) (B b1) (R a1 b1))
- (ask (C a1) ==> NIL (Good!))  
(ask (:not (C a1)) ==> NIL (Huh?))

### ■ *Why can't Loom conclude that a1 is not a C?*

### ■ *Because concepts are not disjoint by default.*

*Just because b1 is a B, it doesn't preclude it from being an A as well.*



# *FAQ: Why Aren't Concepts Disjoint?*

## ■ *Example*

- (defrelation R :attributes :closed-world)  
(defconcept A)  
(defconcept B)  
(defconcept C :is (:and A (:all R A)))

## ■ *Alternate Fixes*

- (defconcept A :implies (:not B))
- (defconcept A  
:characteristics :closed-world)
- *Make A and B members of a partition.*

## ■ *Solution*

- (tell (A a1) (B b1) (R a1 b1))
- (ask (C a1) ==> NIL  
(ask (:not (C a1)) ==> T)





# *FAQ: My Concept Name Changed!*

## ■ *Consider these definitions*

- (defrelation R)  
(defconcept A)  
(defconcept B :is (:and A (:some R A)))  
(defconcept C :is (:and A (:some R A)))

## ■ *Note identical definitions of B and C.*

- (tell (C c1))  
(get-types 'c1) ==> (|C|B |C|A |C|THING)

## ■ *What happened to the concept C?*



# *FAQ: My Concept Name Changed!*

## ■ *Consider these definitions*

- ```
(defrelation R)
(defconcept A)
(defconcept B :is (:and A (:some R A)))
(defconcept C :is (:and A (:some R A)))
```

■ *Note identical definitions of B and C.*

- ```
(tell (C c1))
(get-types 'c1) ==> (|C|B |C|A |C|THING)
```

## ■ *What happened to the concept C? It merged!*

- ```
(ask (C c1)) ==> T
(find-concept 'c) ==> |C|C
```

■ *Loom can find and use it under either name, but only one name is used for display.*



FAQ:

Multiple Value Roles & Defaults

■ *Definitions*

- (defrelation R)
- (defconcept C
:defaults (:filled-by R 5))

■ *Assertions*

- (tell (C c1) (C c2) (R c2 4))

■ *Queries*

- (retrieve ?x (R c1 ?x))
- (retrieve ?x (R c2 ?x))



FAQ:

Multiple Value Roles & Defaults

■ *Definitions*

- (defrelation R)
- (defconcept C
:defaults (:filled-by R 5))

■ *Assertions*

- (tell (C c1) (C c2) (R c2 4))

■ *Queries*

- (retrieve ?x (R c1 ?x)) ==> (5)
- (retrieve ?x (R c2 ?x)) ==> (5 4)



FAQ:

Multiple Value Roles & Defaults



- **Problem:** *You can't easily get rid of default fillers on multiple-value roles*
- **Solution:** *Consider only using them on single-value roles*

FAQ:

Multiple Value Roles & Defaults



- **Problem:** *You can't easily get rid of default fillers on multiple-value roles*
- **Solution:** *Consider only using them on single-value roles*
- **Non-solution:** *Use forget to get rid of default value. Doesn't work because forget just withdraws support for assertions. Loom can prove the value a different way (default inference)*
- **Solution 2:** *Assert the negation. Note that this is very clumsy and not our first choice recommendation*
 - `(tell (:not (R c2 5)))`

FAQ: My New Inverse Doesn't Work!

- *If I define a new inverse relation, the old assertions don't work properly:*

```
(defrelation R)
(tell (R Fred Sue) (R Bill Sue))
(defrelation R-1 :is (:inverse R))
(retrieve ?x (R-1 Sue ?x)) => NIL
```

- *Why weren't Bill and Fred returned?*



FAQ: My New Inverse Doesn't Work!

- *If I define a new inverse relation, the old assertions don't work properly:*

```
(defrelation R)
(tell (R Fred Sue) (R Bill Sue))
(defrelation R-1 :is (:inverse R))
(retrieve ?x (R-1 Sue ?x)) => NIL
```

- *Why weren't Bill and Fred returned?*

- *Loom implements inverse relations by explicitly asserting the inverse relation*
- *Since R-1 did not exist when "R Fred Sue" was asserted, the inverse assertion was not made*



FAQ: Can't Strings Have Inverses?

■ *Why doesn't the following work?*

```
(defrelation Name)
(defrelation Name-of
  :is (:inverse Name))
(tell (R Sue "Sue Jones")) => Error
```



FAQ: Can't Strings Have Inverses?

■ *Why doesn't the following work?*

```
(defrelation Name)
(defrelation Name-of
  :is (:inverse Name))
(tell (R Sue "Sue Jones")) => Error
```

■ *The inverse assertion can't be made!*

- *Built-in types (such as numbers, strings and symbols) cannot have assertions made about them.*
- *The objects are too primitive to support assertions*
- *Inverses are implemented as assertions*



FAQ: Loom vs. CLOS



- *Loom has multiple slots with the same name*
- *Loom “type” hierarchies are determined structurally*
- *Loom relation names have significance in determining the type of objects*
- *Loom instances can have slots added on the fly without redefinition*
- *Loom has a query language*