CS 2740 Knowledge Representation
Lecture 12

Frame-based representation

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Knowledge representation

Many different ways of representing the same knowledge.
Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).

  – It’s easy to ask “What’s red?”

  – But we can’t ask “what is the color of car12?”

  Solution 2: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).

  – It’s easy to ask “What’s red?”

  – But we can’t ask “what is the color of car12?”

  Solution 2: Color (car12, red).

  – It’s easy to ask “What’s red?”

  – It’s easy to ask “What is the color of car12?”

  – Can’t ask “What property of pen7 has value red?”

  Solution 3: ?

Knowledge representation

Many different ways of representing the same knowledge. Representation may make inferences easier or more difficult.

Example:

- How to represent: “Car #12 is red.”

  Solution 1: Red(car12).
  - It’s easy to ask “What’s red?”
  - But we can’t ask “what is the color of car12?”

  Solution 2: Color (car12, red).
  - It’s easy to ask “What’s red?”
  - It’s easy to ask “What is the color of car12?”
  - Can’t ask “What property of pen7 has value red?”

  Solution 3: Prop(car12, color , red).
  - It’s easy to ask all these questions.

Knowledge representation

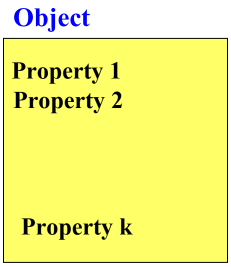
- Prop(Object, Property, Value)
- **Called:** object-property-value representation
- In **FOL statements** about the world, e.g. statements about objects are scattered around
- If we merge many properties of the object of the same type into one structure we get the object-centered representation:

Prop(Object, Property1, Value1)

Prop(Object, Property2, Value2)

…

Prop(Object, Property-n, Value-n)



Object-centered representations

Objects: a natural way to organize the knowledge about

- **physical objects:**
  - a desk has a surface-material, # of drawers, width, length, height, color, procedure for unlocking, etc.
  - some variations: no drawers, multi-level surface
- **situations:**
  - a class: room, participants, teacher, day, time, seating arrangement, lighting, procedures for registering, grading, etc.
  - leg of a trip: destination, origin, conveyance, procedures for buying ticket, getting through customs, reserving hotel room, locating a car rental etc.

Important: Objects enable grouping of procedures for determining the properties of objects, their parts, interaction with parts

---

Frames

Predecessor of object-oriented systems

Two types of frames:

- **individual frames**
  - represent a single object like a person, part of a trip
- **generic frames**
  - represent categories of objects, like students

Example:

- A generic frame: Europian city
- Individual frames: Paris, London, Prague

Frames

- An individual frame is a named list of buckets called **slots**.
- What goes in the bucket is called a **filler of the slot**.

  (*frame-name*

  <slot-name1 filler1>

  <slot-name2 filler2 > …)

Frames

Individual frames have a special slot called : INSTANCE-OF whose filler is the name of **a generic frame**:

Example:

(toronto % lower case for individual frames
<:**INSTANCE-OF** CanadianCity>
<:Province ontario>
<:Population 4.5M>…)

Generic frames may have IS-A slot that includes generic frame

- (CanadianCity % upper case for generic frames
  <:**IS-A** City>
  <:Province CanadianProvince>
  <:Country canada>…)

Frames – inference control

Slots in **generic frames** can have associated procedures that are executed 'control' inference

Two types of procedures:

- **IF-NEEDED procedure**; executes when no slot filler is given and the value is needed

  (Table

    <:Clearance [**IF-NEEDED** computeClearance]> …)
- **IF-ADDED procedure**. If a slot filler is given its effect may propagate to other frames (say to assure constraints)

  (Lecture

    <:DayOfWeek WeekDay>

    <:Date [**IF-ADDED** computeDayOfWeek]> …)
- the filler for :DayOfWeek will be calculated when :Date is filled

---

Frames – defaults

(CanadianCity

<:**IS-A** City>

<:Province CanadianProvince>

<:Country canada>…)

(city134

<:**INSTANCE-OF** CanadianCity>

..)

- A country filler is:

Frames – defaults

(CanadianCity
<:**IS-A** City>
<:Province CanadianProvince>
<:Country canada>…)
(city134
<:**INSTANCE-OF** CanadianCity>
..)

- A country filler is: canada

(city135
<:**INSTANCE-OF** CanadianCity>
<:Country holland>)

- A country filler is:

Frames – defaults

(CanadianCity
<:**IS-A** City>
<:Province CanadianProvince>
<:Country canada>…)
(city134
<:**INSTANCE-OF** CanadianCity>
..)

- A country filler is: canada

(city135
<:**INSTANCE-OF** CanadianCity>
<:Country holland>)

- A country filler is: holland

Frames – inheritance

- Procedures and fillers of more general frame are applicable to more specific frame through the inheritance mechanism

  (CoffeeTable
      <:**IS-A** Table> ...)
  (MahoganyCoffeeTable
      <:**IS-A** CoffeeTable> ...)

  (Elephant
      <:**IS-A** Mammal>
      <:Colour gray> ...)
  (RoyalElephant
      <:**IS-A** Elephant>
      <:Colour white>)
  (clyde
      <:**INSTANCE-OF** RoyalElephant>)

Frames – reasoning

Basic reasoning goes like this:

1. user instantiates a frame, i.e., declares that an object or situation exists
2. slot fillers are inherited where possible
3. inherited **IF-ADDED** procedures are run, causing more frames to be instantiated and slots to be filled.

If the user or any procedure **requires the filler of a slot** then:

1. if there is a filler, it is used
2. otherwise, an inherited **IF-NEEDED** procedure is run, potentially causing additional actions

Frames – reasoning

Global reasoning:

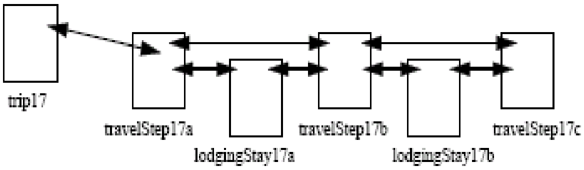
- make frames be major situations or object-types you need to flesh out
- express constraints between slots as **IF-NEEDED** and **IF-ADDED** procedures
- fill in default values when known

Frames – example

A system to **assist in travel planning**

Basic frame types:

- a Trip - be a sequence of TravelSteps, linked through slots
- a TravelStep - terminates in a LodgingStay
- a LodgingStay linked to arriving and departing TravelStep(s)
- TravelSteps includes LodgingStays of their origin and destination



```
(trip17
   <:INSTANCE-OF Trip>
   <:FirstStep travelStep17a>
   <:Traveler  ronB> ...)
```

Frames - examples

TravelSteps and LodgingStays share some properties (e.g., :BeginDate, :EndDate, :Cost, :PaymentMethod), so we might create a more general category as the parent frame for both of them:

```
(Trip
     <:FirstStep TravelStep>
     <:Traveler  Person>
     <:BeginDate  Date>
     <:TotalCost Price> ...)

(TravelStep
     <:IS-A  TripPart>
     <:Means>
     <:Origin>   <:Destination>
     <:NextStep>   <:PreviousStep>
     <:DepartureTime>   <:ArrivalTime>
     <:OriginLodgingStay>
     <:DestinationLodgingStay> ...)

(TripPart
     <:BeginDate>
     <:EndDate>
     <:Cost>
     <:PaymentMethod> ...)

(LodgingStay
     <:IS-A  TripPart>
     <:ArrivingTravelStep>
     <:DepartingTravelStep>
     <:City>
     <:LodgingPlace> ...)
```

Frames - example

Embellish frames with defaults and procedures

```
(TravelStep
     <:Means  airplane> ...)
(TripPart
     <:PaymentMethod  visaCard> ...)
(TravelStep
     <:Origin [IF-NEEDED {if no SELF:PreviousStep then newark}]>)
(Trip
     <:TotalCost
        [IF-NEEDED
          { x←SELF:FirstStep;
           result←0;
           repeat
            { if exists x:NextStep
              then
                 { result←result + x:Cost +
                           x:DestinationLodgingStay:Cost;
                   x←x:NextStep }
             else return result+x:Cost }}]>)
```

assume this is 0 if there is no LodgingStay

Program notation (for an imaginary language):

- *SELF* is the current frame being processed
- if *x* refers to an individual frame, and *y* to a slot, then *xy* refers to the filler of the slot

Frames - example

```
(TravelStep
    <:NextStep
        [IF-ADDED
         {if  SELF:EndDate ≠ SELF:NextStep:BeginDate
          then
             SELF:DestinationLodgingStay ←
                 SELF:NextStep:OriginLodgingStay ←
                     create new LodgingStay
                         with :BeginDate = SELF:EndDate
                         and with :EndDate = SELF:NextStep:BeginDate
                         and with :ArrivingTravelStep = SELF
                         and with :DepartingTravelStep = SELF:NextStep
               …}]>
  …)
```

Note: default :City of LodgingStay, etc. can also be calculated:

```
(LodgingStay
    <:City [IF-NEEDED  {SELF:ArrivingTravelStep:Destination}]…>   …)
```

Frames - example

Propose a trip to Toronto on Dec. 21, returning Dec. 22

```
(trip18
    <:INSTANCE-OF  Trip>
    <:FirstStep travelStep18a>)
```

the first thing to do is to create the trip and the first step

```
(travelStep18a
    <:INSTANCE-OF  TravelStep>
    <:BeginDate 12/21/98>
    <:EndDate 12/21/98>
    <:Means>
    <:Origin>
    <:Destination toronto>
    <:NextStep> <:PreviousStep>
    <:DepartureTime>  <:ArrivalTime>)
```

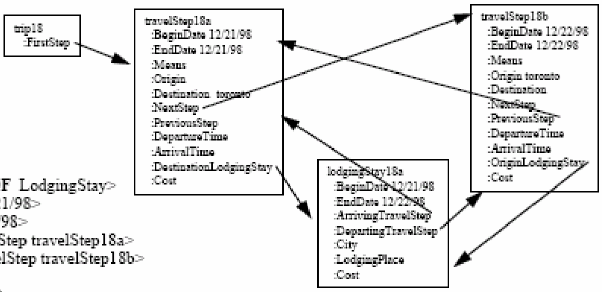
```
(travelStep18b
    <:INSTANCE-OF  TravelStep>
    <:BeginDate 12/22/98>
    <:EndDate 12/22/98>
    <:Means>
    <:Origin toronto>
    <:Destination>
    <:NextStep>
    <:PreviousStep travelStep18a>
    <:DepartureTime>  <:ArrivalTime>)
```

the next thing to do is to create the second step and link it to the first by changing the :NextStep

```
(travelStep18a
    <:NextStep travelStep18b>)
```

Frames - example

IF-ADDED on :NextStep then creates a LodgingStay:



(lodgingStay18a
<:**INSTANCE-OF** LodgingStay>
<:BeginDate 12/21/98>
<:EndDate 12/22/98>
<:ArrivingTravelStep travelStep18a>
<:DepartingTravelStep travelStep18b>
<:City>
<:LodgingPlace>)

If requested, **IF-NEEDED** can provide :City for lodgingStay18a (toronto)

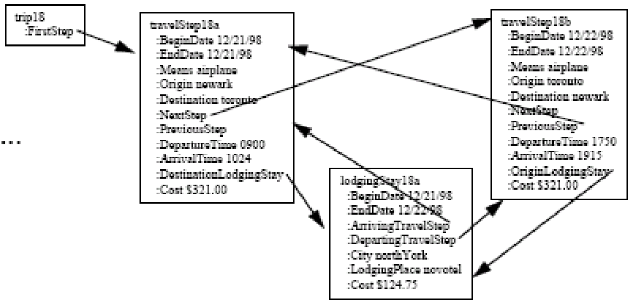
which could then be overridden by hand, if necessary
(e.g. usually stay in North York, not Toronto)

Similarly, apply default for :Means and default calc for :Origin

---

Frames - example

So far...



Finally, we can use :TotalCost **IF-NEEDED** procedure (see above) to calculate the total cost of the trip:

- result← 0, x←travelStep18a, x:NextStep=travelStep18b
- result←0+$321.00+$124.75; x← travelStep18b, x:NextStep=NIL
- return: result=$445.75+$321.00 = $766.75

Using a frame-based system

Main purpose of the above:

- embellish a sketchy description with defaults, implied values
- maintain consistency
- use computed values to:
  - allow derived properties to look explicit
  - avoid up front, potentially unneeded computation

Monitoring

- hook to a DB, watch for changes in values
- like an ES somewhat, but monitors are more object-centered, inherited

---

Frames

- **Declarative vs procedural representation**
  - **Frames allow both declarative and procedural control**
- **Inference is control via procedures**
  - **Can be very tightly controlled, much like an object oriented programming**

- **Differences from OOP:**
  - Frames control via: instantiate/ inherit/trigger cycles
  - OOP: objects sending messages