Overview

- Event Semantics
- FOPC
- Hierarchical Frames
- Frame Semantics
- Applying the Semantics

Semantic Processing Formalism

Recap

- Introduced a simple parser
  - Composed from meaning of parts
  - Semantics: meaning of whole
  - Parsing and Hierarchical Grouping
- Do interpretation on ASR output
  - Semantic building formalism might be too complex to fit into ASR
  - Need to map responses to semantics
- Need to interpret responses longer than single word
Splitting Semantics from Syntax

- Need to refer to semantics of State, Device, and Room

- Command \( \rightarrow \) turn (on \( ?, ?, ? \))

- Command \( \rightarrow \) turn on the light in the Room

- previous formalisms intermixed the semantic formalism with the syntactic rules

- Let's separate the syntactic rule from the semantic formalism

- Command \( \rightarrow \) turn State the Device in the Room

- 2nd line gives semantics for the syntactic rule

- Syntactic rule with non-terminals on right hand side

- Room \( \rightarrow \) living room

- livingroom

- Command \( \rightarrow \) turn (on \( ?, ?, ? \))

- Need to refer to semantics of State, Device, and Room

- Command \( \rightarrow \) turn State the Device in the Room

- previous formalisms intermixed the semantic formalism with

Splitting Semantics from Syntax

- Recap: Terminal Substitution

- Mechanism used to build meaning of the sentence

- Cannot apply any function on the semantics of children in order

- Cannot insert extra material into semantics

- Order of items in sentence must be order in semantic formalism

- The following would all have a different representation

- Mechnism used to build meaning of the sentence
Use semantic representation of turn(State,Device,Room)

Command <- turn Room Device State
Room <- Kitchen
Device <- heat
Device <- light

Referring to the Semantics

Syntactic rule with non-terminals on right hand side

Command <- turn(State,Device,Room)

Could just use non-terminals in semantic rule

\[ \text{Command} \rightarrow \text{turn(State,Device,Room)} \]

Using angle brackets around semantic names

This will make it easy to substitute the appropriate values using Tcl's regsub command

Syntactic rule with non-terminals on right hand side

**Home Control Example**

\[ \text{Command} \rightarrow \text{turn the Device State in the Room} \]
Banking Example

• Use semantic representation of transfer(Amount,From,To)

Account <- Det Type
Account <- Type

Command <- transfer Amount from Act to Act

Command <- transfer Amount to Act from Act

Command <- transfer from Act to Act Amount

• Use semantic representation of transfer(Amount,From,To)

Account <- Det Type account
Account <- Type account
Aside: Variations in Notation

- Exact formalism used for semantics does not matter.
  - We're using a form that is convenient for Tcl's `regsub`.

In Prolog:

- Would like advantage of its variable unification to make this easy.
  - We're using a form that is convenient for Prolog's `repl`.

Exact formalism used for semantics does not matter.

Duplicate Non-Terminals on Right-Hand Side

- Parallel syntactic and semantic rules.
- Terminal Substitution.
- Semantic building formalisms:
  - Give the exact same semantics.
  - Now can account for different transfer variations and have them.

```
Command <- transfer(Amount) from Act:From to Act:To
```

- Allow N's to refer to semantics of N.
  - Syntactic rule might have the same non-terminal twice.
Example Parse & Semantic Interpretation

Although parsing and semantics are separated in each rule, a parser will build semantics while it is applying parsing rules.

Example Parse & Semantic Interpretation
New Version

Create parse list with word sequence as only member
Create semantics list with word sequence as only member

While parse list is not empty
Pull out first alternative from parse list, call it \( p \)
Pull out first alternative from semantics list, call it \( s \)
For each rule \( r \)
Let \( r_b \) be the body of \( r \)
For \( n = 1 \) to length of \( p \)
Let \( p_r \) be subsequence of \( p \) from \( n \) to \( n + \) length of \( r_b \) - 1
Let \( s_r \) be subsequence of \( s \) from \( n \) to \( n + \) length of \( r_b \) - 1
If \( r_b = p_s \)
Create \( p' \) by rewriting \( p_r \) of \( p \) with rule's head
Create \( s' \) by rewriting \( s_r \) of \( s \) with appropriate semantics
If \( p' \) is start symbol
Halt with yes
Otherwise
Add \( p' \) to end of parse list
Add \( s' \) to end of semantics list

Augmenting the Bottom-Up Parser

• Old version: What do we need to change?

New Version

Create parse list with word sequence as only member
Create semantics list with word sequence as only member

While parse list is not empty
Pull out first alternative from parse list, call it \( p \)
For each rule \( r \)
Let \( r_b \) be the body of \( r \)
For \( n = 1 \) to length of \( p \)
Let \( p_r \) be subsequence of \( p \) from \( n \) to \( n + \) length of \( r_b \) - 1
Let \( s_r \) be subsequence of \( s \) from \( n \) to \( n + \) length of \( r_b \) - 1
If \( r_b = p_s \)
Create \( p' \) by rewriting \( p_r \) of \( p \) with rule's head
Create \( s' \) by rewriting \( s_r \) of \( s \) with appropriate semantics
If \( p' \) is start symbol
Halt with yes
Otherwise
Add \( p' \) to end of parse list
Add \( s' \) to end of semantics list

Halt with no
Rewrite rules during initialization to simplify building the semantics.

**Semantics**

Command <- transfer Amount from Act:From to Act:To

\[ \text{transfer(}<\text{Amount},<\text{From},<\text{To})} \]

Create \( s' \) by rewriting \( s \) with appropriate semantics.

Example:

Create \( s' \) by rewriting \( s \) with appropriate semantics.

What needs to be done in:

**Determining the Semantics**

Simplifying Computation of Semantics

- Rewrite rules during initialization to simplify building the semantics.

Command <- transfer Amount from Act:From to Act:To

\[ \text{command <- transfer Amount from Act to Act} \]

- Change semantic rules so it is just refers to the position of non-terminal in the syntactic rule.

\[ \text{command <- transfer Amount from Act:From to Act:To} \]
We have already seen predicate argument structure used for semantics.

- Banking Application could have three frame types:
  - Frame type
  - A fixed number of slots, which can have values.
  - Each argument has a specific meaning.
  - Predicate takes a fixed number of arguments.
  - turn (state, device, room)

- A generalization of this are Frames.

- Frame Semantics

Overview

Frame Semantics

- Event Semantics
- FOPC
- Hierarchical Frames
- Frame Semantics
- Applying the Semantics
- Semantic Processing Formalism
Notation for Frame Semantics

- Notation:
  - Frame name
  - Followed by list of slot & fillers

  - Order of slots doesn't matter
  - Followed by list of slot fillers
  - Frame name

Example (using Tcl lists):

- \{withdraw account=savings amount=20\}
- \{balance account=checking\}
- \{transfer from=savings to=checking amount=20\}

Frame Examples

Withdraw 20 dollars from my savings account
Give me 20 dollars from savings
20 dollars please from savings
Can you give me 20 dollars from my savings
Withdraw 20 dollars from my savings account

Withdraw Frame

Transfer Frame

Balance Frame
A Slightly Different Frame Representation

- Withdraws, transfers & deposits can be thought of as moves:
  - From/to can be cash, savings, checking, line of credit

- What frame inventory is best?
  - What is easiest for semantic processing
  - What does the back-end expect?

Adding Frame Semantics to Grammar

- We have standardized output to ALWAYS have slot names

  act <- bank::savings account
  command <- transfer from acct:from to acct:to amount:amt

  command <- transfer from acct:from to act:to amount:amt
  command <- transfer from acct:from to acct:to amount:amt

© P. Heeman, 2003
Toward More Complex Representation

- Could get cumbersome, especially for other applications
- Account type: checking, loan158, loan160
- Could do:
  - Transfer $20 from loan158 to savings
  - Transfer $20 from loan160 to savings
  - Withdraw $20 from John's checking account
  - Transfer $20 from my checking account to John's checking account
  - Withdraw $20 from John's checking account

When we want to transfer money between people's accounts

Overview

- Semantic Processing Formalism
- Applying the Semantics
- Frame Semantics
- Hierarchical Frames
- Event Semantics
- FOPC
Grammar Rules

- For hierarchical frames, need way to delimit them

```
{ ...
```

- Note: some frame slots can be mandatory; some optional

```
DetOpt <- my
DetOpt <- john's
```

Act <- DetOpt savings ActOpt

Hierarchical Frames

- Might want account to be a frame
- Would have a slot for account number
- Could have a slot for owner
- Could have a slot for type: checking, savings, loan, line of credit
- Might want account to be a frame

<table>
<thead>
<tr>
<th>From</th>
<th>Type</th>
<th>Owner</th>
<th>Account Frame</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>cash</td>
<td>loan</td>
<td>john</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Move Frame</td>
<td></td>
</tr>
</tbody>
</table>

From/To slots can have either a simple value as filler, or a frame
First Order Predicate Calculus (FOPC)

- Back end will figure out what X refers to, say account 109838
- We just refer to it with the variable X, and state what we know about it
- We don't need to know how backend refers to John's checking account, we just refer to it with the variable X, and state what we know about it

Example: transfer 20 dollars from John's checking to my savings

\[ \text{transfer}(20, X, Y) \wedge \text{type}(X, \text{checking}) \wedge \text{owner}(X, \text{John}) \wedge \text{type}(Y, \text{savings}) \]

- Allow conjunctions on variables X and Y
- When we do not know the name
- Lied to capture how arguments of different predicates related to each other
- Allow variables in predicates X, Y, ...
- Allow conjunctions of predicates \( \vee \)

Predicate argument structure can be expanded into FOPC
Lambda expressions are ideal for capturing meaning of arbitrary phrases.
- \( \lambda X \text{owner}(X, \text{john}) \) means we don't need a predicate \( \text{owner}(\text{john}) \).
- From a small set of predicates, we can build arbitrary predicates.

Lambda expressions are a way to build an arbitrary predicate.

\( \lambda X \text{owner}(X, \text{john}) \)

- We can capture this intuition via use of lambda expressions.
- The \( X \) that is being referred to is owned by \( \text{john} \).
- It really should be something closer to \( \lambda \text{owner}(X, \text{john}) \).
- \( \text{owner}(X, \text{john}) \) seems weird as semantics of \( \text{john} \) is stranded on its own.

First Attempt at Mapping to FOPC

\begin{align*}
\text{DetOpt} & \leftarrow \text{my} \\
\text{DetOpt} & \leftarrow \text{the} \\
\text{DetOpt} & \leftarrow \epsilon \\
\text{DetOpt} & \leftarrow \text{john's} \\
\text{Act} & \leftarrow \text{DetOpt} \land \text{ActOpt} \\
\text{Command} & \leftarrow \text{transfer Amt from Act:From to Act:To} \\
& \land \text{type}(X, \text{savings}) \\
& \land \text{type}(X, \text{checking}) \\
& \land \text{owner}(X, \text{john})
\end{align*}

- Semantics of "transfer fifty from savings to john's checking"

\begin{align*}
\text{act} & \leftarrow \text{ DetOpt } \\
\text{command} & \leftarrow \text{ transfer Amt from Act:From to Act:To} \\
\text{type} & \leftarrow \text{type}(X, \text{savings}) \\
\text{owner} & \leftarrow \text{owner}(X, \text{john}) \\
\text{DetOpt} & \leftarrow \text{DetOpt} \\
\text{DetOpt} & \leftarrow \epsilon \\
\text{DetOpt} & \leftarrow \text{the} \\
\text{DetOpt} & \leftarrow \text{my}
\end{align*}
Applying Lambda Expressions

• Predicate symbol can be applied to value to produce predicate:
  ownedByJohn applied to boat224 gives ownedByJohn(boat224)

• Can do the same with lambda expressions:
  \( \lambda X \owner(X,john) \) applied to boat224 gives \( \owner(boat224,john) \)
  which can be simplified to \( \owner(boat224,john) \)

• Use in Semantic Rule:
  When we want to apply a lambda expression, write
  \( \text{apply} \) (Expression, Value)

Example Revisited

• Previous:
  DetOpt <- john's \( \owner(X,john) \)
  Act <- DetOpt savings ActOpt
  type(X,savings) \& \& DetOpt
  Command <- transfer Amt from Act:From to Act:To
  transfer(Amt,X,Y)
  \& From \& To

• New:
  DetOpt <- john's \( \lambda X \owner(X,john) \)
  Act <- DetOpt savings ActOpt
  \& \& \&
  Command <- transfer Amt from Act:From to Act:To
  transfer(Amt,X,Y)
  \& From \& To

Predicat symbol can be applied to value to produce predicate.
In building the semantics, anytime an 'apply' is seen, the expression should be simplified to remove the 'apply'.

Example Revisited Again

• Previous

\[ \text{DetOpt} \leftarrow \text{john's owner}(X, \text{john}) \]
\[ \text{Act} \leftarrow \text{DetOpt savings ActOpt} \]
\[ \text{type}(X, \text{savings}) \land \text{DetOpt} \]
\[ \text{Command} \leftarrow \text{transfer Amt from Act:From to Act:To} \]
\[ \text{transfer}(\text{Amt}, X, Y) \land \text{apply}(\text{From}, X) \land \text{apply}(\text{To}, X) \]

• New

\[ \text{DetOpt} \leftarrow \lambda(X, \text{owner}(X, \text{john})) \]
\[ \text{Act} \leftarrow \text{DetOpt savings ActOpt} \]
\[ \lambda(X, \text{type}(X, \text{savings}) \land \text{apply}(\text{DetOpt}, X)) \]
\[ \text{Command} \leftarrow \text{transfer Amt from Act:From to Act:To} \]
\[ \text{transfer}(\text{Amt}, X, Y) \land \text{apply}(\text{From}, X) \land \text{apply}(\text{To}, X) \]
Event Semantics

- Have a variable that represents the event, say $e$
- Has a type (similar to slots of frame)
  $\text{type}(e) = \text{move}$
- Has roles (similar to slots of frame)
  $\text{from}(e) = \text{savings}$, $\text{to}(e) = \text{checking}$
- Whole thing is
  $\exists E \text{s.t. type}(E) = \text{move} \land \text{from}(E) = \text{savings} \land \text{to}(E) = \text{checking}$
- Depending on the richness of the domain, you might do
  - Lambda expressions can be used for expressing partial meanings

Overview

- Semantic Processing Formalism
- Applying the Semantics
- Frame Semantics
- Hierarchical Frames
- FOPC
- Event Semantics