

21 Semantics of Logic Programs

Semantics of logic programs is much simpler than for imperative programs

1. The Meaning of a Logic Program
2. Term Semantics
3. Finding Meaning
4. Approximating Meaning
5. Groundness Analysis

21.1 The Meaning of a Logic Program

- Meaning of logic program is what it makes true (ignore impurities, e.g. Input/Output, assert/retract)
- Meaning of a program can be shown as a set of unit clauses (facts)
- This program is its meaning:

```
capital(tas, hobart).    capital(vic, melbourne).  
capital(nsw, sydney).   capital(sa, adelaide).  
capital(act, canberra). capital(qld, brisbane).  
capital(nt, darwin).    capital(wa, perth).
```

The Meaning of a Logic Program (2)

```
parent(alice, harriet).  parent(alice, george).  
parent(bob, harriet).   parent(bob, george).  
parent(harriet, laura). parent(harriet, ken).  
gp(C, G) :- parent(C, P), parent(P, G).
```

Meaning is:

```
parent(alice, harriet).  parent(alice, george).  
parent(bob, harriet).   parent(bob, george).  
parent(harriet, laura). parent(harriet, ken).  
gp(alice, laura).       gp(alice, ken).  
gp(bob, laura).         gp(bob, ken).
```

The Meaning of a Logic Program (3)

- Meaning of a recursive predicate is also a set of clauses, e.g.:

`num(0).`

`num(s(N)) :- num(N).`

- `num/1` holds for numbers written in `s` notation, so meaning is:

`num(0).`

`num(s(0)).`

`num(s(s(0))).`

`num(s(s(s(0)))).`

`...`

21.2 Term Semantics

- Semantics of term is set of all instances of it
- Meaning of ground term is singleton set
- Meaning of non-ground term is infinite set of terms got by substituting every term for each variable
- Define:
 - term* is set of all terms
 - ground* is set of all ground terms
 - atom* is set of all atoms
 - clause* is set of all clauses
 - prog* is $\mathcal{P}(\textit{clause})$

21.2.1 Substitutions

- Formally, a substitution is a function $term \rightarrow term$
- Almost an identity function, but uniformly replaces certain variables with other terms
- Traditionally use small greek letters, e.g. θ, σ , written as postfix operator
- Denote particular substitution as set of $term/variable$ pairs

Substitutions (2)

- E.g., If $\theta = \{b/x, g(x)/y\}$, then

$$f(a, x, y)\theta = f(a, b, g(x))$$

- Note: simultaneous substitution, not repeated
- Function f is idempotent iff $\forall x. f(x) = f(f(x))$
- Generally only interested in idempotent substitutions; θ above is not idempotent
- Substitution θ is a unifier of terms t_1 and t_2 iff

$$t_1\theta = t_2\theta$$

21.3 Program Meaning

- Meaning of program P can be found using immediate consequence function T_P :

$$T : prog \rightarrow \mathcal{P}(atom) \rightarrow \mathcal{P}(atom)$$

$$T_P I = \{H\theta \mid (H : -G_1, \dots, G_n) \in P \wedge \\ H\theta \in ground \wedge \\ G_1\theta \in I \wedge \dots \wedge G_n\theta \in I\}$$

- This requires that θ unifies G_1, \dots, G_n with elements of I , and grounds all variables in H

Program Meaning (2)

- Take

$$T_P^2(I) = T_P(T_P(I))$$

$$T_P^3(I) = T_P(T_P(T_P(I)))$$

⋮

- Meaning of program P is $T_P^\infty(\emptyset)$
- \emptyset is the empty set of atoms

Example

- Take program $P = \{num(0), num(s(N)) : -num(N)\}$
- Then:

$$T_P^1(\emptyset) = \{num(0)\}$$

$$T_P^2(\emptyset) = \{num(0), num(s(0))\}$$

$$T_P^3(\emptyset) = \{num(0), num(s(0)), num(s(s(0)))\}$$

$$T_P^\infty(\emptyset) = \left\{ \begin{array}{l} num(0), num(s(0)), num(s(s(0))), \\ num(s(s(s(0)))), \dots \end{array} \right\}$$

- Meaning of most programs is infinite

21.4 Approximating Meaning

- Can use semantics to verify correctness of logic program
- Check that two programs have same meaning: check that optimization is correct
- Can also approximate meaning of a program: gives information about program
- Approximation can be made finite: can actually be computed

Approximating Meaning (2)

- Called abstract interpretation
- Usual approach: replace data in program by an abstraction, then compute meaning
- Many interesting program properties are Boolean
- *E.g.*, parity (even/odd): abstract 0 by **T**
- Abstract $s(X)$ by $\neg x$, where x is approximation of X

Approximating Meaning (3)

- Abstracted `num` predicate:

$$num(\top)$$

$$num(\neg x) \leftarrow num(x)$$

$$T_P^1(\emptyset) = \{num(\top)\}$$

$$T_P^2(\emptyset) = \{num(\top), num(\text{F})\}$$

$$T_P^3(\emptyset) = \{num(\top), num(\text{F})\}$$

- Further repetitions will not add anything
- Result is finite
- Says even and odd numbers satisfy `num/1`

Approximating Meaning (4)

Program:

`plus(0, Y, Y).`

`plus(s(X), Y, s(Z)) :- plus(X, Y, Z).`

Abstracted version:

$\{plus(\top, y, y), plus(\neg x, y, \neg z) \leftarrow plus(x, y, z)\}$

Because y in first clause can be either T or F, this gives:

$$T_P^1(\emptyset) = \{plus(\top, \top, \top), plus(\top, \text{F}, \text{F})\}$$

Approximating Meaning (5)

$$\{plus(\top, y, y), plus(\neg x, y, \neg z) \leftarrow plus(x, y, z)\}$$

$$T_P^1(\emptyset) = \{plus(\top, \top, \top), plus(\top, \text{F}, \text{F})\}$$

$$T_P^2(\emptyset) = \left\{ \begin{array}{l} plus(\top, \top, \top), plus(\top, \text{F}, \text{F}), \\ plus(\text{F}, \top, \text{F}), plus(\text{F}, \text{F}, \top) \end{array} \right\}$$

$$T_P^3(\emptyset) = \left\{ \begin{array}{l} plus(\top, \top, \top), plus(\top, \text{F}, \text{F}), \\ plus(\text{F}, \top, \text{F}), plus(\text{F}, \text{F}, \top) \end{array} \right\}$$

Approximating Meaning (6)

- No further work needed; this is the answer
- Better shown as a table:

+	even	odd
even	even	odd
odd	odd	even

- Even such a simple analysis is able to discover interesting properties of programs

21.5 Groundness Analysis

- Similar analysis can be used to determine groundness of arguments to Prolog predicates
- Powerful abstract domain called Pos, abstracts terms to Boolean indicating whether they must always be ground
- Predicates abstracted to Boolean formulae
- Unification goal $X = f(X_1, \dots, X_n)$ abstracted to $x \leftrightarrow (x_1 \wedge \dots \wedge x_n)$
- Means X is ground iff all of X_1, \dots, X_n are

21.5.1 Append Example

$ap([], Y, Y).$

$ap([U|X], Y, [U|Z]) :- ap(X, Y, Z).$

Abstracted version:

$\{ap(T, y, y), ap(u \wedge x, y, u \wedge z) \leftarrow ap(x, y, z)\}$

Because y in first clause must be T or F:

$$T_P^1(\emptyset) = \{ap(T, T, T), ap(T, F, F)\}$$

Append Example (2)

$$\{ap(\top, y, y), ap(u \wedge x, y, u \wedge z) \leftarrow ap(x, y, z)\}$$

u can be \top or F ; clause 2 simplifies to either

$ap(x, y, z) \leftarrow ap(x, y, z)$ or $ap(F, y, F) \leftarrow ap(x, y, z)$

$$T_P^1(\emptyset) = \{ap(\top, \top, \top), ap(\top, F, F)\}$$

$$T_P^2(\emptyset) = \left\{ \begin{array}{l} ap(\top, \top, \top), ap(\top, F, F), \\ ap(F, \top, F), ap(F, F, F) \end{array} \right\}$$

$$T_P^3(\emptyset) = \left\{ \begin{array}{l} ap(\top, \top, \top), ap(\top, F, F), \\ ap(F, \top, F), ap(F, F, F) \end{array} \right\}$$

21.5.2 Naive Reverse Example

- So after a call $ap(X,Y,Z)$, X and Y will be ground iff Z is
- Use this to analyze calls to ap :

```
rev([], []).  
rev([U|X], Y) :-  
    rev(X, Z),  
    ap(Z, [U], Y).
```

- Abstracted version:

$$\left\{ \begin{array}{l} rev(T, T), \\ rev(u \wedge x, y) \leftarrow rev(x, z) \wedge ap(z, u, y) \end{array} \right\}$$

Naive Reverse Example (2)

Using what we've learned about ap:

$$\left\{ \begin{array}{l} rev(\top, \top), \\ rev(u \wedge x, y) \leftarrow rev(x, z) \wedge ((z \wedge u) \leftrightarrow y) \end{array} \right\}$$

Again u is \top or F , so clause 2 is either

$$rev(x, y) \leftarrow rev(x, z) \wedge (z \leftrightarrow y) \text{ or } rev(F, y) \leftarrow rev(x, z) \wedge (F \leftrightarrow y)$$

This simplifies to: $rev(x, y) \leftarrow rev(x, y)$ or
 $rev(F, F) \leftarrow rev(x, z)$

Naive Reverse Example (3)

$$\left\{ \begin{array}{l} rev(T, T), \\ rev(x, y) \leftarrow rev(x, y), \\ rev(F, F) \leftarrow rev(x, z) \end{array} \right\}$$

$$T_P^1(\emptyset) = \{rev(T, T)\}$$

$$T_P^2(\emptyset) = \{rev(T, T), rev(F, F)\}$$

$$T_P^3(\emptyset) = \{rev(T, T), rev(F, F)\}$$

21.6 Conclusions

- So after $\text{rev}(X,Y)$, X is ground iff Y is: $x \leftrightarrow y$
- This analysis gives quite precise results
- Many other such abstractions have been proposed
- If abstract domain is correctly implemented, abstract interpretation guaranteed to give correct analysis
- This is an area of active research

22 Review

A brief recap of what we've covered this semester

1. Abstraction
2. Object oriented programming
3. Functional Programming
4. Logic Programming
5. Semantics

22.1 Abstraction

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

— Edsgar Dijkstra

- Languages determine what abstractions are available to programmers
- Abstractions we have seen include: ADTs, inheritance, polymorphism, functions, higher order functions, type systems, nondeterminism, constraint solving

22.1.1 Language Issues

- Binding time: when a decision is made
- *E.g.*, run, compile, coding, language implementation or language design time
- Applies to variable type, value, allocation, deallocation, procedure choice, branching
- Other issues: readability, familiarity, consistency, orthogonality, simplicity, substitutivity

22.2 Object Oriented Programming

- Based on ADTs and Inheritance
- Abstract Data Type: encapsulation of type and operations on that type in single unit, allowing instances of type to be created
- In Java, class defines an ADT:

```
class Something {  
    ...  
}
```

- ... part includes both data member and method definitions

22.2.1 Inheritance

- Inheritance: define one ADT by specifying only how it differs from another
- In Java, specify inheritance with `extends` keyword:

```
class Something extends SomethingElse {  
    ...  
}
```
- `Something` class inherits data members and methods from `SomethingElse`

22.2.2 Shadowing vs. Overriding

- Class may have definition of same member as superclass; two possible resolutions:
- Shadowing: subclass's member hides the inherited member; declared type of object determines which is used
- Overriding: subclass's member replaces inherited member; actual type determines which is used
- Shadowing more efficient, overriding more powerful

22.2.3 Information Hiding

- OOP commonly also supports information hiding: hiding implementation details from class user
- In Java: put keyword `public` or `private` (or `protected`) before member definition
- Key idea: clearly distinguish public, unchanging interface part from fluid implementation part
- Protects user from confusing detail; protects implementor from lock-in to bad design decisions

22.2.4 Multiple Inheritance

- Some object oriented programming languages allow multiple superclasses; Java does not
- MI can cause conflicts when same member inherited from multiple ancestors, or when same member inherited along multiple paths
- Java supports multiple interface inheritance: class can implement multiple interfaces
- `interface` is like fully abstract class: specifies only interface, no implementation

22.2.5 Object Oriented Style

- Usually focus on one argument of function call as recipient of message
- Type of recipient determines which method is executed
- Common object oriented syntax:
Object.message(arg, ...)
- Similar to syntax for accessing data member:
Object.member

22.3 Functional Programming

- Based on mathematical function: given same inputs, always produces same output
- Many functional languages are impure, allowing functions to return different results for same inputs
- Haskell is pure

22.3.1 Function Types

- Haskell is strongly typed: compiler will not accept program unless types are consistent
- Each function can be declared with type signature specifying type of that function, *e.g.*

```
f :: Int -> Int -> Int
```

specifies that `f` expects an integer as input and returns a function from integer to integer as result

22.3.2 Currying

- Also think of this as function mapping 2 integers to an integer
- Currying: function takes single argument and returns (curried) function mapping remaining arguments to result
- Allows partial function application: supply some arguments to function and pass result where function is expected
- Any function can be curried

22.3.3 Algebraic Type Systems

- Haskell supports algebraic types: type is declared as set of alternatives
- Each alternative has unique constructor and 0 or more typed arguments
- Each constructor is then (curried) function yielding value of defined type
- Unique constructors make this a discriminated union type: can tell which alternative each value is

22.3.4 Parametric Polymorphism

- Types can take parameters (other types)
- Can use those parameters in type definition to specify other types, *e.g.*:

```
data Tree a =  
  Empty | Node a (Tree a) (Tree a)
```

- Specifies a tree of any constituent type, but every element is of exactly that type
- Statically checked by compiler
- Equivalent to generics in other languages

22.3.5 Higher Order Functions

- Higher order functions: functions taking functions as arguments or yielding functions as values
- Currying makes many functions higher order
- Common higher order functions: compose, map, filter, fold

22.3.6 Laziness

- Haskell functions are call by need: arguments are not evaluated until they are needed
- Values are only evaluated once; result is remembered if needed again
- Allows infinite computations, only part of which is needed, *e.g.*:

```
repeat      :: a -> [a]
repeat x    = xs where xs = x:xs
```

```
replicate   :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

22.3.7 Type Classes

- Constrained polymorphism: type parameter can be any type that supports certain operations
- *E.g.*, tree elements can be any type that supports comparison operation
- Type class is like Java interface: allow specification of signatures of needed operations, without giving implementations
- Type classes can inherit from others and can provide default implementations for operations

22.4 Logic Programming

- Based on predicate calculus
- Program is set of predicate definitions
- Predicate can be defined by multiple clauses
- Each clause specifies some conditions to make predicate true

22.4.1 Terms

- Prolog data are terms
- Term is either integer, float, atom, variable or compound term
- Compound term is functor (uninterpreted function symbol; an atom) applied to some terms called arguments
- Variable is first class term: can be used anywhere a term can
- Variable can be bound to any term as program proceeds

22.4.2 Resolution and Unification

- Logic programming solves queries by finding bindings for variables to make query true
- Resolution: match query with clause head, then solve clause body by resolution
- Nondeterministic: multiple clause heads may match; if solving body fails, then try other clauses
- Match query with clause head by unification: bind variables as much as needed to make identical

22.4.3 Modes

- Predicates can be invoked in different modes: some args unbound variables, others bound
- Prolog uses bound args to solve for unbound
- Some Prolog builtins (*e.g.*, arithmetic) cannot be used in all modes
- In some modes, Prolog code hits infinite loops
- Sometimes hard to make code work in all modes
- Builtins like `var/1` can be used to do this, but are extra-logical and make conjunction not commutative

22.5 Semantics

- Can define meaning of program data to be mathematical values (e.g., numbers, tuples, sets, functions)
- Interpretation maps computer values to their meanings
- Types in real programming languages approximate their mathematical counterparts with some errors
- Infinite mathematical types cannot be fully supported on computers

22.5.1 Denotational Semantics

- Denotational Semantics uses functions to map program to meaning
- For imperative (and OO) language, must consider changes to computation state throughout
- Much simpler for pure language
- Produce recursive functions; compute fixed point to find actual meaning
- Can approximate program meaning to analyze program: abstract interpretation

