Semantics of Predicate Calculus

Robert Keller
1 December 2003

A Few Reasons why this stuff is of interest outside of pure logic for its own sake:

- **Databases**: DB structure and query languages are based on a concept of semantics.
- **Artificial Intelligence**: Requires representation of knowledge for purposes of world-modeling and interaction.
- **Programming languages**: Semantics of a language has many ideas in common with semantics of logic.
- **Software Development**: Formal specification of the meaning of programs is desirable.

Syntax vs. Semantics

- Predicate logic proofs, in a system such as natural deduction, focus on **syntax**: each formula in the derivation is **mechanically-checkable** at a **symbolic** level to be derivable from the given rules.
- The **semantics** or **meaning** of a formula is determined by separate considerations. Each formula is making a statement about some kind of **underlying mathematical structure**.

Interpretations of Formulas

- The structure(s) of interest in specific derivations are generally **not totally specified** in the system of derivation itself.
- Instead, we rely on certain formulas ("axioms") to **characterize** the properties of these structures that are of interest.
- It can then be proved separately that the syntactic rules are in agreement with the semantics of the intended **interpretation**.

What is an "Interpretation"?

- Each predicate logic formula can be viewed as making a statement (which could be true or false) about a structure. An **interpretation** for a set of formulas consists of:
  - A **domain**: that contains all individuals of interest.
  - A set of **constants** from the domain, one for each constant **symbol** in the logical language.
  - A set of **functions** mapping n-tuples of domain elements into domain elements, one for each function **symbol** in the logical language.
  - A set of **predicates** mapping n-tuples of domain elements into {T, F}, one for each predicate **symbol** in the logical language.

"Interpretation" vs. "Model"

- What we just called a interpretation the HR book calls a "model".
- But in most presentations, the latter term is reserved for interpretations in which a specific set of formulas is "true".
- We will reserve the term "model" for this use.

Example: Peano Axioms

- The Peano axioms are:
  - (□x) □(S(x) = 0)
  - (□x) (□y) ((S(x) = S(y)) □ (x = y))
  - (□[0/x] □ (□n) (□[n/x] □ □[S(n)/x])) □ (□n)□[n/x]

    for **every formula** □ where n is free for x.
- S is the only function symbol, = is the only predicate symbol.

Example: An Interpretation for **Peano Axioms**

- The **domain** is the set of natural numbers: {0, 1, 2, 3, …}.
- There is one **constant** symbol: **0**.
  The associated domain value is the number 0.
- There is one **function** symbol: **S**.
  The associated function is successor.
- There is one **predicate** symbol: =.
  The associated predicate is equality.

Equality Axioms
(These can be derived using N.D. inference rules)

- $(\forall x)\ (x = x)$
- $(\forall x)(\forall y)\ ((x = y) \rightarrow (y = x))$
- $(\forall x)(\forall y)(\forall z)(((x = y) \wedge (y = z)) \rightarrow (x = z))$
- $(\forall x_1) \ldots (\forall x_n)(\forall y_1) \ldots (\forall y_n)$

  $(((x_1=y_1) \wedge \ldots \wedge (x_n=y_n)) \rightarrow (\mathbf{f}(x_1,\ldots,x_n) = \mathbf{f}(y_1,\ldots,y_n)))$

  for each n-ary function symbol **f**.
- $(\forall x_1) \ldots (\forall x_n)(\forall y_1) \ldots (\forall y_n)$

  $(((x_1=y_1) \wedge \ldots \wedge (x_n=y_n)) \rightarrow (\mathbf{p}(x_1,\ldots,x_n) \rightarrow \mathbf{p}(y_1,\ldots,y_n)))$

  for each n-ary predicate symbol **p**.

ND Inference Rules for Equality

- $\dfrac{}{t = t}$ =i

  (t is any term)

- $\dfrac{t_1 = t_2 \quad \phi[t_1 / x]}{\phi[t_2 / x]}$ =e

  (t_1 and t_2 are free for x in ϕ)

Examples Deriving Equality "Axioms"

- Derive $\vdash \forall x\,(x = x)$

1.	x_0	
2.	$x_0 = x_0$	=i
3.	$\forall x\,(x = x)$	$\forall$x i 1-2

Examples Deriving Equality "Axioms"

- Derive $\vdash \forall x \forall y\,((x = y) \rightarrow (y = x))$

1.	x_0	
2.	y_0	
3.	$x_0 = y_0$	Assumption
4.	$x_0 = x_0$	=i
5.	$y_0 = x_0$	=e 3, 4 (with λ: $x = x_0$)
6.	$(x_0 = y_0) \rightarrow (y_0 = x_0)$	$\rightarrow$ i 3-5
7.	$\forall y\,((x_0 = y) \rightarrow (y = x_0))$	$\forall$y i 2-6
8.	$\forall x \forall y\,((x = y) \rightarrow (y = x))$	$\forall$x i 1-7

Examples Deriving Equality "Axioms"

- Derive $\vdash$ $(\forall x)(\forall y)(\forall z)(((x = y)\wedge(y = z)) \rightarrow (x = z))$

1.	x_0	
2.	y_0	
3.	z_0	
4.	$(x_0 = y_0)\wedge(y_0 = z_0)$	Assumption
5.	$(x_0 = y_0)$	$\wedge e_1$ 4
6.	$(y_0 = z_0)$	$\wedge e_2$ 4
7.	$(y_0 = x_0)$	previous deriv. 6
8.	$(x_0 = z_0)$	=e 7, 6 (Γ: $x = z_0$)
9.	$((x_0 = y_0)\wedge(y_0 = z_0)) \rightarrow (x_0 = z_0)$	$\rightarrow$ i 4-8
10.	$(\forall z)\ (((x_0 = y_0)\wedge(y_0 = z)) \rightarrow (x_0 = z))$	$\forall z$ i 3-9
11.	$(\forall y)(\forall z)\ (((x_0 = y)\wedge(y = z)) \rightarrow (x_0 = z))$	$\forall y$ i 2-10
12.	$(\forall x)(\forall y)(\forall z)\ (((x = y)\wedge(y = z)) \rightarrow (x = z))$	$\forall x$ i 1-11

Theories

- A **Theory** is a set of formulas called **theorems** derived from a basis set of **axioms**.
- To avoid re-listing all the axioms each time, we will use the notation:

  $\Gamma \vdash \varphi$

  where Γ is a **set** of formulas, to mean that φ can be derived from formulas in Γ.
- Note that Γ could be **infinite**. For example, in the case of the Peano axioms, there is an infinite set of induction axioms. In a given proof, however, only a finite set of axioms could actually be used.

Elementary Number Theory

- (∀x) ¬(S(x) = 0)
- (∀x) (∀y) ((S(x) = S(y)) → (x = y))
- (∀x) (x + 0) = x
- (∀x) (∀y) (x + S(y)) = S(x+y)
- (∀x) (x * 0) = 0
- (∀x) (∀y) (∀z) (x*S(y)) = ((x*y) + x)
- The function symbols are {S} (1-ary) and {+, *} (2-ary).
- The predicate symbol is =.
- Note that there is no induction.

Non-Uniqueness of Models

- Note that while some theories, such as number theory, may have a standard **intended** model, there is no guarantee that the model is unique.
- Many theories will intentionally have a **range** of models rather than a single model.

Linear Order Theory

- $(\forall x) \neg(x < x)$
- $(\forall x) (\forall y) (x < y) \quad (x = y) \quad (y < x)$
- $(\forall x) (\forall y) (\forall z) (((x < y) \wedge (y < z)) \rightarrow (x < z))$
- There are many possible linear orders, with different numbers of elements, including ones with both countable and uncountable numbers of elements.

Dense Linear Order Theory

- $(\forall x) \neg(x < x)$
- $(\forall x)(\forall y) (x < y)$
- $(\forall x) (\forall y) (x < y) \quad (x = y) \quad (y < x)$
- $(\forall x) (\forall y) (\forall z) (((x < y) \wedge (y < z)) \rightarrow (x < z))$
- $(\forall x) (\forall z) (x < z) \rightarrow (\exists y) (x < y) \wedge (y < z))$

Semi-Group Theory

- (□x)(□y)(□z) (x + (y + z)) = ((x + y) + z)

Monoid Theory

- (□x)(□y)(□z) (x + (y + z)) = ((x + y) + z)
- (□x) ((x + 0) = x)

Group Theory

- $(\forall x)(\forall y)(\forall z)\ (x + (y + z)) = ((x + y) + z)$
- $(\forall x)\ ((x + 0) = x)$
- $(\forall x)(\exists y)\ ((x + y) = 0)$

Ex: Proving a Theorem in Group Theory $\vdash$

- Show $\vdash$ $|\!-$ $(\forall x)(\forall y)(\forall z)\ (((y + x) = (z + x)) \rightarrow (y = z))$

1.	x_0	
2.	y_0	
3.	z_0	
4.	$(y_0 + x_0) = (z_0 + x_0)$	Assumption
5.	$(\forall x)(\forall y)\ ((x + y) = 0)$	Group Axiom
6.	$(\forall y)\ ((x_0 + y) = 0)$	$\forall$x e 5
7.	w_0 $(x_0 + w_0) = 0$	Assumption
8.	$(y_0 + x_0) + w_0 = (z_0 + x_0) + w_0$	Equality Rule
9.	$y_0 + (x_0 + w_0) = z_0 + (x_0 + w_0)$	Group Axiom & Eq. (twice)
10.	$y_0 + 0 = z_0 + 0$	Equality Rule
11.	$y_0 = z_0$	Group Axiom (twice)
12.	$y_0 = z_0$	$\forall$y e 7-11
13.	$(y_0 + x_0) = (z_0 + x_0) \rightarrow y_0 = z_0$	$\rightarrow$ i 4-12
14.	$(\forall z)\ (y_0 + x_0) = (z + x_0) \rightarrow y_0 = z$	$\forall$z i 3-13
15.	$(\forall y)\ (\forall z)\ (y + x_0) = (z + x_0) \rightarrow y = z$	$\forall$y i 2-14
16.	$(\forall x)(\forall y)(\forall z)\ (((y + x) = (z + x)) \rightarrow (y = z))$	$\forall$x i 1-15

Abelian Group Theory

- (∀x)(∀y)(∀z) (x + (y + z)) = ((x + y) + z)
- (∀x) ((x + 0) = x)
- (∀x)(∃y) ((x + y) = 0)
- (∀x)(∀y) ((x + y) = (y + x))

Ring Theory

- (∀x)(∀y)(∀z) (x + (y + z)) = ((x + y) + z)
- (∀x) ((x + 0) = x)
- (∀x)(∃y) ((x + y) = 0)
- (∀x)(∀y) ((x + y) = (y + x))
- (∀x)(∀y)(∀z) ((x * (y * z)) = ((x * y) * z))
- (∀x)(∀y)(∀z) ((x * (y + z)) = ((x * y) + (x * z)))
- (∀x)(∀y)(∀z) ((y + z)*x) = ((y * x) + (z * x)))

Commutative Ring Theory

- (∀x)(∀y)(∀z) (x + (y + z)) = ((x + y) + z)
- (∀x) ((x + 0) = x)
- (∀x)(∃y) ((x + y) = 0)
- (∀x)(∀y) ((x + y) = (y + x))
- (∀x)(∀y)(∀z) ((x * (y * z)) = ((x * y) * z))
- (∀x)(∀y)(∀z) ((x * (y + z)) = ((x * y) + (x * z)))
- (∀x)(∀y) ((x * y) = (y * x))

Field Theory

- (∀x)(∀y)(∀z) (x + (y + z)) = ((x + y) + z)
- (∀x) ((x + 0) = x)
- (∀x)(∃y) ((x + y) = 0)
- (∀x)(∀y) ((x + y) = (y + x))
- (∀x)(∀y)(∀z) ((x * (y * z)) = ((x * y) * z))
- (∀x)(∀y)(∀z) ((x * (y + z)) = ((x * y) + (x * z)))
- (∀x)(∀y) ((x * y) = (y * x))
- (∀x) ((x * 1) = x)
- (∀x) (¬(x = 0) → ((∃y) ((x *y) = 1)))
- ¬(1 = 0)

Something to Notice

- As we add more formulas that need to be satisfied,

  we subset the interpretations that satisfy the characterization.

- In a sense, every formula serves as an "interpretation filter".

Filter Example

- In the Peano axioms, these two act together to filter out any interpretations with finite domains.
  - (∀x) ¬(S(x) = 0)
  - (∀x) (∀y) ((S(x) = S(y)) → (x = y))
- S(0) must be distinct from 0 by the first axiom
- S(S(0)) must be distinct from S(0) by the second axiom and the previous statement, and from 0 by the first axiom.
- S(S(S(0))) must be distinct from S(S(0)) by the second axiom and the previous statement, from S(0) by the second axiom and the statement before that, and from 0 by the first axiom.
- . . .

Definition of Truth

- In propositional logic, truth was defined as "true for all valuations".
- A valuation assigned a {T, F} value to each proposition, and hence to an entire formula.
- In predicate logic, the role of propositions is replaced by atomic formulas (predicates applied to terms).
- We can't arbitrarily pick truth values for these, because there are **inter-relationships** to be taken into account, depending on the **interpretation**.

Definition of Truth

- We have already defined an **interpretation** as determining a domain, and mappings Φ from the constant, function, and predicate symbols to constants, functions, and predicates in the domain. If c is a constant symbol, Φ(c) will be the corresponding constant, if f is a function symbol, Φ(f) will be the corresponding function, and if p is a predicate symbol, Φ(p) will be the corresponding predicate.
- An "assignment" (aka "environment") for a formula maps each free variable into the domain of a structure.
- In so-doing, it determines a domain element for every term, and thus a truth value for each atomic formula.

Assignments

- An **assignment** □ (aka "environment"), which is always relative to an **interpretation** □, maps each **free** variable of a formula into the domain of the interpretation. Extend an assignment to map arbitrary terms into domain values and arbitrary formulas into {T, F} as follows:
  - For any variable v, □(v) is the value assigned by the assignment.
  - For any constant c, □(c) = □(c), the value assigned by the interpretation.
  - For any term of the form $f(t_1,\ldots, t_n)$, □(f($t_1,\ldots, t_n$)) = □(f)(□(t_1),…, □(t_n)).
  - For any atomic formula $p(t_1,\ldots, t_n)$ □ determines a truth value by □(p($t_1,\ldots, t_n$)) = □(p)(□(t_1),…, □(t_n)).
  - For any formula of the form (□ □), □(□ □) = T iff □(□) = T or □(□) = T.
  - For any formula of the form (□ □ □), □(□ □ □) = T iff □(□) = T and □(□) = T.
  - For any formula of the form (□ □ □), □(□ □ □) = T iff □(□) = F or □(□) = T. For any formula of the form (□□), □(□□) = T iff □(□) = F.
  - The **next slide** shows how to determine □((□v)□) and □((□v)□).

Assignment Values for Quantification

- For formulas (□v)□ and (□v)□, the truth value for an assignment □ is determined as follows:
  - □((□v)□) = T provided that □'(□) = T for **each** assignment □' for □ that agrees with □ on all free variables in (□v)□.
  - □((□v)□) = T provided that □'(□) = T for **some** assignment □' for □ that agrees with □ on all free variables in (□v)□.
- Note that the extension of an assignment for a formula with **no** free variables simply evaluates the formula to T or F, depending only on the **interpretation**.

Example of Interpretation and Assignment

- Consider the formula
  $(\exists x)\ ((x = 0) \quad (\exists y)(x = S(y))$
- An interpretation gives us a domain D and a function $\mathcal{I}(S)$: D $\rightarrow$ D.
- Suppose the domain is {0, 1, 2} with $\mathcal{I}(0) = 0$ and $\mathcal{I}(S)$ is defined by $\mathcal{I}(S)(0) = 1$, $\mathcal{I}(S)(1) = 2$, $\mathcal{I}(S)(2) = 1$.
- The sub-formula x = 0 will have $\mathcal{I}(x = 0) = T$ exactly in cases where $\mathcal{I}(x) = 0$.
- The sub-formula x = S(y) will have $\mathcal{I}(x = S(y)) = T$ exactly in three cases:
  - $\mathcal{I}(x) = 1$ and $\mathcal{I}(y) = 0$
  - $\mathcal{I}(x) = 1$ and $\mathcal{I}(y) = 2$
  - $\mathcal{I}(x) = 2$ and $\mathcal{I}(y) = 1$

Example of Interpretation and Assignment

- Still considering the formula
  $(\exists x)\ ((x = 0) \quad (\exists y)(x = S(y))$
- From the previous slide,
  sub-formula $((x = 0) \quad (\exists y)(x = S(y))$ will have
  $\mathcal{I}(((x = 0) \quad (\exists y)(x = S(y))) = T$
  exactly in cases where $\mathcal{I}(x) \in \{0, 1, 2\}$, since
  - $\mathcal{I}(x = 0) = T$ when $\mathcal{I}(x) = 0$ and
  - $\mathcal{I}((\exists y)(x = S(y))) = T$ when $\mathcal{I}(x) \in \{1, 2\}$.
- Since every $\mathcal{I}$ must have $\mathcal{I}(x) \in \{0, 1, 2\}$ (because that is the entire domain), we have
  $\mathcal{I}(((x = 0) \quad (\exists y)(x = S(y))) = T$
  for every assignment $\mathcal{I}$.
- Therefore $\mathcal{I}((\exists x)((x = 0) \quad (\exists y)(x = S(y))) = T$ for any assignment.

Note on Extending Assignments to Valuations

- If the formula in question has **no** free variables, the rules of extension given determine a **constant** value, either T or F, for the formula, **independent** of the assignment.

Another Example and Assignment

- Consider the formula
  $(\forall y)(0 = S(y))$
- using the same interpretation as before: domain is {0, 1, 2} with $\mathcal{I}(0) = 0$ and $\mathcal{I}(S)$ is defined by
  $\mathcal{I}(S)(0) = 1$, $\mathcal{I}(S)(1) = 2$, $\mathcal{I}(S)(2) = 1$.
- The sub-formula $0 = S(y)$ will have $\mathcal{I}(0 = S(y)) = T$ for **no** assignments σ.
- Therefore for any assignment σ, we have
  $\mathcal{I}((\forall y)(0 = S(y))) = F$.
- Note that there could be ***some other interpretation*** (with a different meaning for S) for which the formula evaluates to T.

Formalizing |=

- When $\alpha_1, \ldots, \alpha_n, \beta$ are predicate calculus formulas,

  $\alpha_1, \ldots, \alpha_n \models \beta$

  means:

  For every interpretation $\mathcal{I}$ for the formulas (taken collectively)

  for every assignment σ such that for each i $\mathcal{I}(\alpha_i)$ = T,

  it must also be the case that $\mathcal{I}(\beta)$ = T.

- In most cases, the formulas will be closed (no free variables), but the notation works in general.

Models

- An interpretation for a set of formulas S such that

  for every assignment σ, $\mathcal{I}(\sigma)$ = T for each $\sigma \in$ S

  is called a **model** for S.

|= in predicate calculus vs. propositional

- The predicate version of $\alpha_1, \ldots, \alpha_n \models \beta$ is a **very broad** statement:
  - The set of applicable structures is generally infinite.
  - If a given domain is infinite, so is the set of assignments.
- Intuitively there is much less likely to be an algorithm to check whether $\alpha_1, \ldots, \alpha_n \models \beta$ in the way there is for the propositional calculus.

Soundness and Completeness

- We are not going to prove it here, but soundness and completeness both hold for the natural deduction rules for the predicate calculus:

  $\alpha_1, \ldots, \alpha_n \vdash \beta$

  iff

  $\alpha_1, \ldots, \alpha_n \models \beta$
- Put more succinctly:

  $\vdash$ = $\models$

Validity and Satisfiability

- If the left-hand side of
  $\Gamma_1, \ldots, \Gamma_n \models \Delta$
  is empty:

  $\models \Delta$

  then we say that Δ is **valid**. In other words, $\Im(\Delta) = T$ for every interpretation and every assignment $\Im$.

- If $\Im(\Delta) = T$ for **some** interpretation and some assignment $\Im$, we say that Δ is **satisfiable**.
- In other words, a closed formula Δ is satisfiable if it has a model.

- Thus for a closed formula Δ, $\models \Delta$ iff $\neg\Delta$ is **un**satisfiable.

Decidability (Solvability)

- **Analogous** to the propositional calculus, if we could devise an algorithm that would determine $\Gamma_1, \ldots, \Gamma_n \models \Delta$ for any set of formulas, then we could determine $\Gamma_1, \ldots, \Gamma_n \vdash \Delta$ as well, thanks to soundness and completeness.

- However, **unlike** the propositional calculus, there is **no algorithm for the predicate calculus**; thus neither semantic entailment nor derivability can be determined algorithmically.

Proof that validity is undecidable

- We are going to reduce Post's Correspondence Problem to the problem of determining validity:
- For each instance J of PCP, there is a formula □(J) such J has a solution iff |= □(J).

Proof that validity is undecidable

- We are following HR.
- Without loss of generality, we will assume that the alphabet for PCP is always {0, 1}. Any other alphabet's characters could be encoded into this alphabet.
- Let the instance of PCP be the pairs $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$, where each $s_i, t_i \in \{0, 1\}^*$.

The Logic Language

- The logic language will contain:
  - A constant symbol e.
  - Two 1-ary function symbols f_0, and f_1.
  - A 2-ary predicate symbol P.

Encoding words as terms.

- For each strings $b_1 b_2 \ldots b_m \in \{0, 1\}^*$ define a corresponding term

  $f_{b_m}(\ldots(f_{b_2}(f_{b_1}(e)))\ldots)$

  and ***abbreviate*** this term as $f_{b_m \ldots b_2 b_1}(e)$.

- Since we will be interested in strings from the set $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$

  f_{s_i} and f_{t_i} will denote the corresponding function compositions.

The Formula □(J)

- For the instance J of the PCP, defined by the pairs $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$, □(J) will be:

  $\square_1 \wedge \square_2 \wedge \square_3$

  where $\square_1$, $\square_2$, $\square_3$ are given on the following slides, and are constructed so that:

  $\models \square(J)$ iff J has a solution.

The Sub-Formula $\square_3$ of □(J)

- $\square_3$ is $(\forall z)P(z, z)$
- The intention is that P will be constrained (by $\square_1$ and $\square_2$) so that P(u, v) is forced to evaluate to T whenever (u, v) is a pair of terms corresponding to a **partial** solution to J, i.e. if one of of the following two strings is a **prefix** of the other

  $s_{i1}\, s_{i2} \ldots s_{im}$ vs. $t_{i1}\, t_{i2} \ldots t_{im}$

  then

  $P(f_{sim}(\ldots(f_{si2}(f_{si1}(e)))\ldots), f_{tim}(\ldots(f_{ti2}(f_{ti1}(e)))\ldots))$

- So $(\forall z)P(z, z)$ will evaluate to T iff the corresponding partial solution is indeed a solution.

The Sub-Formula Φ_1 of $\Phi(J)$

- Φ_1 is the "starter kit" for the partial solutions:

  $P(f_{s1}(e), f_{t1}(e)) \wedge P(f_{s2}(e), f_{t2}(e)) \ldots \wedge P(f_{sk}(e), f_{tk}(e))$

- Every partial solution must begin with one of the pairs of the instance J. This requires P(u, v) to be T for all such initial pairs (u, v).

The Sub-Formula Φ_2 of $\Phi(J)$

- Φ_2 is the "extender kit" for partial solutions:

  $(\forall u)(\forall v)\ (P(u, v) \rightarrow$

  $(P(f_{s1}(u), f_{t1}(v)) \wedge P(f_{s2}(u), f_{t2}(v)) \wedge \ldots \wedge P(f_{sk}(u), f_{tk}(v)))$

- This says that if (u, v) is a pair of terms corresponding to a partial solution, then so is $f_{si}(u), f_{ti}(v))$ for each i.

Proof that |= □(J) implies J has a solution (1 of 2)

- |= □(J) means that □(J) evaluates to T for **every** interpretation (assignment is irrelevant because □(J) is closed), which includes the following **specific** interpretation:
  - The domain is the set {0, 1}*.
  - □(e) = □, the null string
  - □(f_0)(x) = x0, the function that suffixes 0 to its argument
  - □(f_1)(x) = x1, the function that suffixes 1 to its argument
  - □(P)(u, v) is true exactly when (u, v) is a partial solution to J

Proof that |= □(J) implies J has a solution (2 of 2)

- It is easy to see that this specific interpretation is such that for the components of the formula □(J) (□$_1$ □ □$_2$ □ □$_3$):
  - □(□$_1$) = T, the "starter kit" is true:

    □(P(f_{s1} (e), f_{t1} (e)) □ P(f_{s2} (e), f_{t2} (e)) . . . □ P(f_{s2} (e), f_{t2} (e))) = T
  - □(□$_2$) = T, the "extender kit" is true:

    □((□u)(□v) (P(u, v) □
    (P(f_{s1}(u), f_{t1}(v)) □ P(f_{s2}(u), f_{t2}(v)) □ … □ P(f_{sk}(u), f_{tk}(v)))) =T
- Since □(□(J)) = T by assumption, the meaning of □ requires that □(□$_3$) = T,

  □((□z)P(z, z)) = T

  which means that J **has a solution**, namely one corresponding to a term z that makes P(z, z) true.

Proof that J has a solution implies |= □(J) (1 of 3)

- Suppose that instance J of PCP has a solution

  $s_{i1} s_{i2} \ldots s_{im} = t_{i1} t_{i2} \ldots t_{im}$.

  We have to show that, for **any** interpretation □ of □(J), □(□(J)) = T.

- By the meaning of □ , this is trivially true if either □(□$_1$) = F or □(□$_2$) = F.
- So proceed assuming □(□$_1$) = □(□$_2$) = T.
- We now must show □(□$_3$) = T as well.

Proof that J has a solution implies |= □(J) (2 of 3)

- Let D be the domain of □. Define a function I:{0, 1}* □ D by induction as follows:
  - I(□) = □(e).
  - I(x0) = □(f_0)(I(x))
  - I(x1) = □(f_1)(I(x))
- Hence for any string $b_1 b_2 \ldots b_m$ we will have

  $I(b_1 b_2 \ldots b_m) = □(f_{bm})(\ldots □(f_{b2}(□(f_{b1}(□(e)))))\ldots)$.

Proof that J has a solution implies |= □(J) (3 of 3)

- Due to □(□$_1$) = □(□$_2$) = T, we see by induction on m that for any **partial solution** (s_{i1} s_{i2} … s_{im}, t_{i1} t_{i2} … t_{im}),

  □(P) must be such that

  □(P)(I(s_{i1} s_{i2} … s_{im}), I(t_{i1} t_{i2} … t_{im})) = T.

- For a solution (s_{i1} s_{i2} … s_{im}, t_{i1} t_{i2} … t_{im}), we will **also** have

  I(s_{i1} s_{i2} … s_{im}) = I(t_{i1} t_{i2} … t_{im}),

  so that if there is a solution, □(P)((□z)P(z, z)) = □(□$_3$) = T.

- Hence □(□(J)) = T.

- But since the interpretation was chosen arbitrarily, |= □(J).

Partial Decidability

- The undecidability of |= (and hence |□) means that there is no algorithm that will determine whether or not □$_1$, …, □$_n$ |= □ for a finite set of formulas {□$_1$, …, □$_n$, □}.

- In other words, for any fixed set of formulas {□$_1$, …, □$_n$}, the language {□ | □$_1$, …, □$_n$ |= □} is **not recursive**.

- However the language {□ | □$_1$, …, □$_n$ |□ □} **is recursively enumerable**.

- We can **enumerate** all the formulas derivable from {□$_1$, …, □$_n$} much in the same way we can enumerate the strings derivable in a grammar.

Automated Reasoning

- "Automated Reasoning" is the term used to describe methods that increase efficiency of determining whether $\Gamma_1, \ldots, \Gamma_n \mid\!\sim \Delta$, in the sense that if this sequent is true, then the algorithm will determine that fact. (The algorithm cannot conclude that it is false in general.)
- Many automated reasoning techniques are based on "**resolution theorem proving**", which is great for machines, but not so easy for humans.
- A special case of resolution (for Horn-clause logic) is the basis for the **Prolog** programming language.

Decidable Theories

- We have seen how it is not decidable in the general case whether a formula is derivable.
- However, for certain **special categories of formulas** (cf. W. Ackerman, W. Solvable Cases of the Decision Problem. North-Holland Publishing Co., 1954), or for **certain theories**, the question can be decided algorithmically.
- What Makes a Theory Decidable?
  - Restricting the language to specific functions and predicates.
  - Axioms that constrain the possible models to ones where questions can be answered by computation.

Some Known Decidable Theories

- Presburger Arithmetic (natural numbers with just +, no *, and with a form of induction)
- Robinson Arithmetic (natural numbers with +, *, <, but no induction)
- Dense Linear Orders
- Real Closed Fields
- Boolean Algebras

Presburger Arithmetic

- (∀x) ¬(0 = x + 1)
- (∀x)(∀y) ¬(x = y) ⇒ ¬(x + 1 = y + 1)
- (∀x) x + 0 = x
- (∀x)(∀y) (x + y) + 1 = x + (y + 1)
- If P is any formula involving a single free variable x:

  (P[0/x] ∧ (∀x) (P ⇒ P[(x + 1)/x])) ⇒ (∀x) P

- Formulas can be decided based on a finite automata-like construction, as shown by M. Presburger in 1929!
- In 1974, Fischer and Rabin proved that any decision procedure requires double exponential time (2^2^n) where n is the length of the formula.

Forms of Completeness

- Earlier we claimed that
  $\Gamma_1, \ldots, \Gamma_n \models \Delta$ implies $\Gamma_1, \ldots, \Gamma_n \vdash \Delta$ ("Completeness")
- Recall that $\Gamma_1, \ldots, \Gamma_n \models \Delta$ means that Δ is true in **every** interpretation for which **each** Γ_i is true, while $\Gamma_1, \ldots, \Gamma_n \vdash \Delta$ means that Δ is derivable from $\Gamma_1, \ldots, \Gamma_n$.
- A closed formula Δ could be true in **some, but not all**, **interpretations** for which each **each** Γ_i is true. By soundness, we would **not** expect such a Δ to be provable from $\Gamma_1, \ldots, \Gamma_n$.

Forms of Completeness

- For a formula Δ that is not true in all models, it might be the case that the **negation** $\neg\Delta$ is true in all models, in which case $\neg\Delta$ would be provable.
- A ***theory*** is called **complete** if for **any** closed formula Δ, either
  - Δ is provable within the theory or
  - $\neg\Delta$ is provable within the theory.
- From the discussion above, whether or not a **theory** is complete is **independent** of the notion of completeness used earlier:
  - Completeness as used on previous pages is **completeness of the logical proof formalism**.
  - Completeness as defined on **this** page is **completeness of a specific theory**.
- Sometimes the form of completeness for a theory is called "**negation-completeness**".

Consistency: A needed technicality

- Could it ever be that **both** φ and $\neg\varphi$ are provable in the same theory?
- Yes. Such a theory is called **inconsistent**. From it we can derive $\bot$, and, from that, **anything**. The theorems of such a theory would thus carry no information content, so the theory is worthless.
- A theory is **consistent** if there is no formula φ such that both φ and $\neg\varphi$ are provable.

Completeness vs. Decidability

- A little reflection will reveal that **completeness for a theory** is a very strong requirement, not to be expected to hold in an arbitrary theory.
- We already mentioned the **set of theorems** of a theory is **recursively-enumerable**, provided the set of axioms is finite.
- More generally, the set of theorems will be recursively-enumerable even if the set of axioms is just **recursive**, but not necessarily finite.
- Such a theory is called an "**recursively-axiomatized**" theory.

Completeness vs. Decidability

- If a consistent, recursively-axiomatized, theory also happens to be **complete**, then something special is gained: Its set of theorems is **recursive**, not just recursively-enumerable.
- In the latter case, since we know that for any closed formula □ either □ or □□ is derivable, there is an algorithm for determining whether or not □ is derivable:
  - In parallel, run two computations:
    - Computation A accepts if □ is derivable from the axioms.
    - Computation B accepts if □□ is derivable from the axioms.
  - Whichever computation terminates with a "yes" answer will determine whether □ is derivable or not.
    - By consistency, we know both can't say "yes".
    - By completeness, we known that one of them will say "yes".
- Such a theory is called a **decidable** theory.

Extensions of Theories

- If we add more axioms to a theory, new theorems might be derivable that weren't before.
- One theory **extends** another if the axioms of the former include those of the latter.
- It is easy to see that the theorems of the extension must therefore include the theorems of the original theory, but not necessarily conversely.

Three (Meta-)Theorems by Gödel

- Completeness Theorem:
  - Relates to logic framework being complete
- Incompleteness Theorem:
  - Relates to theories being incomplete
- Second Incompleteness Theorem:
  - Relates to theories being unable to prove their own consistency

Gödel's Completeness Theorem

- The completeness of first-order predicate calculus as a **logic** was first established by Kurt Gödel in his dissertation in 1929.
- Kurt Gödel, "Über die Vollständigkeit des Logikkalküls", doctoral dissertation, University Of Vienna, 1929.
- Kurt Gödel, "Die Vollständigkeit der Axiome des logischen Funktionen-kalküls", Monatshefte für Mathematik und Physik 37 (1930), 349-360. This article contains the same material as the doctoral dissertation, in a rewritten and shortened form.

Gödel's Incompleteness Theorem
(as refined by Rosser)

- **No consistent recursively-axiomatized extension of number theory is complete.**
- In other words, we can *try* to build up an all-powerful mathematical theory. At a minimum, it must include number theory, which is not asking very much. But such a theory will always be **incomplete**.
- This result, published in 1931, meant that Hilbert's idea of mechanizing all of mathematics could **never** be achieved.
- Kurt Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I.* Monatshefte für Mathematik und Physik, 38 (1931), pp. 173-198. Translated in van Heijenoort: *From Frege to Gödel*. Harvard University Press, 1971., online at http://home.ddc.net/ygg/etext/godel/
- B. Rosser: *Extensions of some theorems of Gödel and Church*. Journal of Symbolic Logic, 1 (1936), N1, pp. 87-91.

Contrapositive

- A complete, consistent theory must be weak, in that in cannot even include all true formulas for about the natural numbers as theorems.

But ***which*** "Number Theory"?

- Essentially one with +, *, and induction, usually called **Peano Arithmetic** (not simply Peano Axioms).
- (∀x) ¬(S(x) = 0)
- (∀x) (∀y) ((S(x) = S(y)) → (x = y))
- (∀x) (x + 0) = x
- (∀x) (∀y) (x + S(y)) = S(x+y)
- (∀x) (x * 0) = 0
- (∀x) (∀y) (∀z) (x*S(y)) = ((x*y) + x)
- (φ[0/x] ∧ (∀n) (φ[n/x] → φ[S(n)/x])) → (∀n)φ[n/x]

  for **every formula** φ where n is free for x.

How did Gödel's proof work?

- Similar to the encoding of Turing machine tapes as numbers and Turing machines as partial recursive functions, Gödel showed how to:
  - encode formulas as numbers
  - encode proofs as numbers
  - create a number-theoretic formula □ (for "provable"), where

    □(p, f, a)

    means: for any formula φ(v) with v representing the one and only free variable:

    f is the encoding of **formula** φ

    p is the encoding of a **proof** of formula φ(a), where

    a is an **argument**, substituted for v in φ(v).

How did Gödel's proof work?

- Consider the following formula **□(f)** incorporating formula □:
  (□p) □□(p, f, f)
- For an **arbitrary** formula □ with encoding f, □(f) says

  there is **no** proof of □(f).

- Let d be the encoding of □(f). So **□(d)** says

  there is **no** proof of □(d).

- Now is □(d) provable? If it is, then a **non-truth** is provable, because □(d) says "□(d) is not provable".
- More precisely, if □(d) is provable, then letting p encode the proof, we have □(p, d, d), which is at odds with with (□p) □□(p, d, d), which would give us □□(p, d, d) by the □p elimination rule, which would mean the theory is **inconsistent**.
- So if the theory is consistent, □(d) is **not provable**.

How did Gödel's proof work?

- Recall □(d) says

  there is **no** proof of □(d).

- Since we've established that □(d) is not provable, □(d) is **true**. So we have a formula that is **true for the natural numbers, *reasoned informally*,** but **not provable** within the theory about the natural numbers.
- Is □□(d) provable? If the theory is consistent, it had better not be, because □□(d) is false and a false statement would contradict the soundness of the logical system.
- Since neither □(d) nor □□(d) is provable, we are forced to conclude that **number theory, *if consistent*, cannot also be complete**.

Connection with Decidability

- In particular, what does Gödel's incompleteness theorem have to with **decidability** (= solvability)?
- First, the construction of □(d) should look roughly similar to the proof of unsolvability of the halting problem:
  - □(d) asserts there is **no** proof of □(d)
  - $T_k(x_k)$ converges iff T_k diverges on x_k
- Second (and this is not obvious in our brief exposition) the functions used to build up □, from which is derived, are the **partial recursive functions**. This is because these are the natural functions that can be constructed within number theory. In fact, it was Gödel who developed the concept of partial recursive functions, largely for this purpose.

Number Theory is Undecidable

- A Turing machine computation can be uniformly encoded as a number theoretic functions. In slides on partial-recursive functions, we stated:
  - Halting in i steps is expressed by:

    $\exists i\ [P(T(i, x_0)) = 0]$
- So the logic formula that expresses whether a machine, coded as primitive-recursive functions P and T, halts is:

  $\exists i\ [P(T(i, x_0)) = 0]$
- This formula is either true or false. If we could determine which, we'd be solving the halting problem.

Gödel's Second Incompleteness Theorem

- Within any consistent-extension of number theory, there is a formula that expresses the **consistency** of the theory but which is not provable in the theory.
- (and by now, this should be no surprise.)

Connections Between Logic, Computability, and Complexity

- A decision problem is in **NP** if it can be solved by a **Non-deterministic Turing machine** in time **Polynomial** in the length of the input.
- A decision problem is in **P** if it can be solved by a **deterministic** Turing machine in Polynomial time.
- A problem is **NP-complete** if it is NP and any other problem in NP **reduces** to it by a polynomial-time reduction.
- The theory of NP came, in part, from Stephen Cook's investigation of complexity of **automatic theorem proving**: The problem of determining whether a **propositional** formula is **satisfiable** is NP-complete.
- It is **unknown** whether P = NP. (It seems **unlikely**, but no one has been able to prove this yet.)
- If P = NP, every NP complete problem (of which there are many) has a polynomial-time algorithm.
- For a list of 88 NP-complete problems, see: http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html

Logic and Computing Timeline

- 1822 Babbage: Difference Engine
- 1847 Boole: Laws of Thought
- 1866 Boolean Algebra applied to Switching
- 1890 Hollerith Tabulating Machines
- 1900 Fleming invents Vacuum tube
- 1929 Gödel: Completeness Theorem
- 1931 Gödel: Incompleteness Theorem
- 1935 Church: An unsolvable problem of elementary number theory, lambda calculus
- 1936 Kleene: General recursive functions, unsolvability
- 1936 Post: computing processes (essentially equivalent to Turing machines)
- 1937 Turing invents the Turing machine, uncomputability
- 1939 Stored-program electronic computer
- 1947 First transistor
- 1948 Univac: First commercial electronic computer
- 1954 Fortran
- 1958 First integrated circuit
- 1963 Sutherland's Sketchpad
- 1963 LINC minicomputer
- 1969 Arpanet
- 1971 First microprocessor
- 1972 Pocket calculator
- 1973 Xerox alto, point-and-click
- 1975 Ethernet
- 1976 Apple I
- 1981 IBM PC
- 1983 Visicalc
- 1984 Macintosh