

Logic (master program)

Georg Moser

Institute of Computer Science @ UIBK

Winter 2008



Definition

logspace reducibility

- for $A \subseteq \Sigma^*$, $B \subseteq \Delta^*$
- we write $A \leq_m^{\log} B$
if \exists logspace-computable function $\sigma: \Sigma^* \rightarrow \Delta^*$ with
$$x \in A \text{ if and only if } \sigma(x) \in B$$
- we say A is **logspace reducible** to B

Definition

hardness

- a set $A \subseteq \Sigma^*$ is **\leq_m^{\log} -hard** for a complexity class \mathcal{C} if
- $\forall B \in \mathcal{C}$ we have $B \leq_m^{\log} A$

Definition

completeness

- a set $A \subseteq \Sigma^*$ is **complete for \mathcal{C} with respect to \leq_m^{\log}** if
- 1 A is **\leq_m^{\log} -hard** for \mathcal{C}
 - 2 $A \in \mathcal{C}$

Summary of Last Lecture

Definition

logspace transducer

- a **total** deterministic logspace-bounded TM with output is called **logspace transducer**
- **total** means: it halts on all inputs
- **with output** means: \exists write-only output tape
- hence, a logspace transducer has:
 - 1 a **two-way read-only input tape**
 - 2 a **two-way read/write logspace-bounded worktape** initially blank
 - 3 a **write-only left-to-right output tape** initially blank
 - 4 Σ is the input alphabet
 - 5 Γ is the worktape alphabet
 - 6 Δ is the output alphabet

Definition

let \mathcal{S} a set of **finite** \mathcal{V} -structures for finite \mathcal{V}

- φ a \mathcal{V} -sentence
- \mathcal{M} a \mathcal{V} -structure in \mathcal{S}
- the **φ - \mathcal{S} problem** asks: $\mathcal{M} \models \varphi$

Definition

a second-order sentence φ is **existential** (\exists SO for short) if

$$\varphi \equiv \exists R_1^{n_1} \exists R_2^{n_2} \dots \exists R_k^{n_k} \psi \quad \psi \text{ is first-order}$$

Theorem

Fagin

let \mathcal{V} be a relational vocabulary, \mathcal{S} a set of finite \mathcal{V} -structure

- a \mathcal{V} -sentence φ is equivalent to a sentence in \exists SO if and only if φ - $\mathcal{S} \in \text{NP}$
- moreover the first-order part of φ is universal

Content

introduction, propositional logic, semantics, formal proofs, resolution (propositional)

first-order logic, semantics, structures, theories and models, formal proofs, Herbrand theory, completeness of first-order logic, properties of first-order logic, resolution (first-order)

introduction to computability, introduction to complexity, finite model theory

beyond first order: modal logics in a general setting, **higher-order logics**, **introduction to Isabelle**

Higher-Order Logic

Definition

higher-order logic (HOL) is conceivable as **functional programming** + **logic**; it features

- 1 datatypes
- 2 (total) recursive functions
- 3 logical operators

$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \forall \quad \exists \quad \dots$

Definition

types

- base types: `bool`, `nat`
- type constructors: `list`, for example `(nat) list`
- function types, denoted by \Rightarrow
- type variables: `'a`, `'b`

Definition

terms

- function application
if `f` is function of type $\tau_1 \Rightarrow \tau_2$, `t` term of type τ_1
then `f t` is **term** of type τ_2
- λ -abstraction
for example: `$\lambda x. x + 1$`
- basic constructs
 - if `b` then `t1` else `t2`
 - let `x = t` in `u` for example: `let x=0 in x+x`
 - ...

Definition

formulas

formulas are of type `bool`

- truth constants and propositional connectives

`True` `False` \neg \wedge \vee \rightarrow

- equality `=` of type `'a \Rightarrow 'a \Rightarrow bool`
`=` is also interpreted as 'if and only if'
- quantifiers: $\forall x. P$, $\exists x. P$

Mini-Tutorial Isabelle/HOL

System Architecture

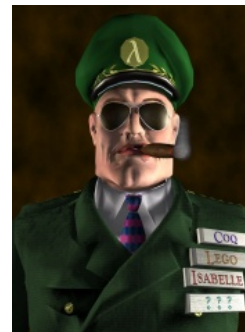
- 1 **standard ML**
- 2 **Isabelle**
- 3 **Isabelle/HOL**
- 4 **ProofGeneral**

implementation language

generic theorem prover

Isabelle instance for HOL

interface



```
theory MyTh
imports ImpTh1, ..., ImpThn
begin
declarations, definitions, theorems, proofs, ...
end
```

Demo: Theory

```

theory Demo
imports Main
begin

datatype 'a list = Nil                ("[]")
| Cons 'a "'a list"                 (infixr "#" 65)

primrec app :: "'a list => 'a list => 'a list"
where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

primrec rev :: "'a list => 'a list"
where
"rev Nil = Nil" |
"rev (Cons x xs) = app (rev xs) (Cons x Nil)"

```

Schema

```

lemma <name>: " ... "
apply (...)
apply (...)

done

```

```

lemma <name>[simp]: " ... "

```

Methods

- 1 structural induction: `apply (induct_tac x)`
 x free variable in first subgoal, type of x needs to be a datatype
- 2 simplification: `apply (auto)`
tries to solve as many subgoals as possible, ignores quantified formulas, logical connectives
- 3 `sorry`: “proves” anything
- 4 `quickcheck`: tries to find counterexamples

Demo: Theorem Proving

```

lemma app_Nil[simp]:
"app xs [] = xs"
sorry

lemma app_assoc[simp]:
"app (app xs ys) zs = app xs (app ys zs)"
sorry

lemma rev_app[simp]:
"rev (app xs ys) = app (rev ys) (rev xs)"
sorry

theorem rev_rev: "rev (rev xs) = xs"
apply (induct_tac xs)
apply auto
done

```

Datatypes

Example

```

datatype 'a list = Nil | Cons 'a "'a list"

```

Definition

datatype

$$\text{datatype } (\alpha_1, \dots, \alpha_n) t = \begin{array}{l} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ \dots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

- 1 **distinctness** `Nil` \neq `Cons x xs`
- 2 **injectivity** `(Cons x xs = Cons y ys)` \Rightarrow `x = y` \wedge `xs = ys`

Example

```

datatype boolex = Const bool | Var nat | Neg boolex
| And boolex boolex

```

Demo: Trees

```

datatype 'a tree = Tip
| Node "'a tree" 'a "'a tree"

lemma "Tip ~ = Node l x r"
sorry

consts mirror :: "'a tree => 'a tree"

primrec
"mirror Tip = Tip"
"mirror (Node l x r) =
Node (mirror r) x (mirror l)"

lemma [simp]: "mirror(mirror t) = t"
apply (induct_tac t)
apply (auto)
done

```

Definition

primitive recursion

$$\text{primrec } f \ x_1 \ \dots \ (C \ y_1 \ \dots \ y_k) \ \dots \ x_n = r$$

- 1 C a constructor
- 2 all recursive calls to f in r are of form $f \ \dots \ y_i \ \dots$

Example

```
case xs of [] => [] | y # ys => y
```

Definition

case expressions

$$\text{case } e \text{ of } \begin{array}{l} C_1 \tau_{1,1} \ \dots \ \tau_{1,n_1} \Rightarrow e_1 \\ \dots \\ C_k \tau_{k,1} \ \dots \ \tau_{k,n_k} \Rightarrow e_k \end{array}$$

Example

```
lemma '(case xs of [] => [] | y # ys => xs) = xs'
apply (case_tac xs)
done

```

Definition

nat

```
datatype nat = 0 | Suc nat
```

Example

```

consts sum :: "'nat => nat'"
primrec 'sum 0 = 0'
      'sum (Suc n) = Suc n + sum n'

```

nat is overloaded, needs type annotations: `1 :: nat`

Definition

list

```

datatype 'a list = "[]" (" []")
| "#" 'a ('a list) (infixr 65)

```

syntactic sugar: $[x_1, \dots, x_n]$

primitive functions: `hd`, `tl`, `map`, `length`, `filter`, `set`, `nth`, `take`, `drop`, `distinct`, ...