Theme 1: Abstract Reasoning

Lecture 2: Logic-based Program Specification

Matthieu Sozeau

Inria & Paris Diderot University, Paris 7

October 2014

Abstract Specification of a Function

- Consider a function
$$f : Dom \to CoDom$$
- How to describe in an abstract way its behavior ?
- Abstraction: No implementation details.
- Specification: A relation *Spec_f* between inputs and outputs of f
$$Spec\_f(In; Out) \subseteq Dom \times CoDom$$
- What is a suitable (natural) formalism for describing such a relation?

Logic-based Specification Language

- Example: Specification of the Append function:
$$\begin{array}{l} Spec\_Append(\ell_1; \ell_2; \ell) = \\ \quad |\ell| = |\ell_1| + |\ell_2| \ \wedge \\ \quad \forall i \in Nat: (i < |\ell_1|) \Rightarrow \ell[i] = \ell_1[i] \ \wedge \\ \quad \forall i \in Nat: (i < |\ell_2|) \Rightarrow \ell[|\ell_1| + i] = \ell_2[i] \end{array}$$
where:
$$\begin{array}{l} \forall \ell \in List[\alpha]: \forall i \in Nat: \forall e \in \alpha: \ \ell[i] = e \iff \\ \quad (i < |\ell|) \wedge \\ \quad \exists \ell': \ \ell = a \cdot \ell' \wedge \\ \quad\quad ((i = 0 \wedge e = a) \vee (i > 0 \wedge \ell'[i-1] = e)) \end{array}$$
- $\Rightarrow$ First-order logic over data domains (natural numbers, lists, etc.), with recursive predicates.

Domains of Interpretation

- Data domain with a set of operations and predicates
  - Consider a data domain D
  - Let *Op* be a set of operations interpreted as functions over D
  - Let *Pred* be a set of predicates interpreted as relations over D
- Remark:
  Here the set Op may include constants, seen as operators of arity 0.
- Domain of interpretation is a triple $(D; Op; Rel)$.
- Examples of domains of interpretation:
  - $(Bool; \{\texttt{tt}; \texttt{ff}; \texttt{not}; \texttt{or}; \texttt{and}\}; \{=\})$
  - $(Nat; \{0; s; +\}; \{\leq\})$
  - $(List[\alpha]; \{[]; \cdot; \texttt{@}\}; \{=\})$

First Order Logic over a Data Domain

- Let $(D; Op; Pred)$ be a domain of interpretation.
- Let *Var* be a set of variables.
- Terms:

$$t ::= v \in Var \mid op(t; : : : ; t)$$

  where $v \in Var$ and $op \in Op$.
- Examples: x, 2, $x + 2$, $x + y + 3$, and $2x$ as an abbreviation of $x + x$.
- Terms are interpreted as elements of the domain D:
  - Let $\;: Var \to D$ be a valuation of the variables.
  - Then, $\langle t \rangle_\nu$ is the value in D obtained by the evaluation of t, using as valuation of the variables.
  - Example: Given $= \{(x; 2); (y; 1); (z; 4)\}$, we have

$$\langle x \rangle_\nu = 2 \quad \langle x + 2y \rangle_\nu = 4 \quad \langle (x * z) + (y + 1) \rangle_\nu = 10$$

First Order Logic: Syntax of formulas

- Formulas ($p \in Pred$ and $v \in Var$).

$$::= \top \mid \bot \mid p(t_1; : : : ; t_n) \mid \quad \vee \quad \mid \quad \wedge \quad \mid \exists v: \quad \mid \forall v: \quad \mid \quad \Rightarrow$$

- Abbreviations: $\neg \quad ::= (\quad \Rightarrow \bot); \quad \iff \quad ' ::= \quad \Rightarrow \quad ' \wedge \quad ' \Rightarrow$
- An occurrence of a variable x is bound in a formula if it is under a quantifier $\exists x$ or $\forall x$. We consider only well-formed formulas where all occurrences of a variable are either bound or unbound ($x = 0 \vee \exists x; x = 1$ is not well-formed). A variable is *free* in if its occurrences in are unbound. A formula is *closed* if it has no free variables.
- Examples:
  - $_1 = \forall x; y: x \leq y \Rightarrow \exists z: (x \leq z \wedge z < y)$ is a closed formula.
  - $_2 = \exists x: \forall y: x \leq y$ is a closed formula.
  - $_3 = \forall y: x \leq y$, is an open formula. It has x as free variable.
  - $_4 = x \leq y \wedge \exists z: y \leq z \wedge z \leq 5$ is an open formula. Its free variables are x and y.

First Order Logic: Semantics of formulas

Given a valuation $\;: Var \to D$ of the *free* variables, we can define an interpretation $[\;]$ which replaces every occurrence of a free variable by its associated value:

$$
\begin{aligned}
x[\;] &= \quad (x) \\
p(t_1; : : : ; t_n)[\;] &= p(t_1[\;]; : : : ; t_n[\;]) \\
(\quad \wedge \quad ')[\;] &= \quad [\;] \wedge \quad '[\;] \\
(\quad \vee \quad ')[\;] &= \quad [\;] \vee \quad '[\;] \\
(\forall v: \quad)[\;] &= \forall v: \quad [\;] \\
(\exists v: \quad)[\;] &= \exists v: \quad [\;] \\
(\quad \Rightarrow \quad ')[\;] &= \quad [\;] \Rightarrow \quad '[\;]
\end{aligned}
$$

- $(x = 3)[x \mapsto 2] = 2 = 3$
- $(\exists x; x = y)[y \mapsto 3] = \exists x; x = 3$

First Order Logic: Semantics of formulas

- Given a valuation , we say satisfies if and only if $[\;]$ is true, i.e., when interpreting the formula using , the formula is valid.
- Formulas are interpreted as relations over D, i.e., the sets of valuations of the variables that satisfy the formula.
- Let $[\![\;]\!]$ be the set of valuations which satisfy .
- A formula is *valid* if it is satisfied by all valuations. A formula is *satisfiable* if there exists at least one valuation that satisfies it.
- Remark:

  Closed formulas are either valid or not: Their value does not depend on the variable valuation. Either all variable valuations satisfy them, or none of the valuations can satisfy them.
- Question: what can we say about the formulas in the previous slides?

First Order Logic: proving validity

To show the validity of a quantified formula, we must formally *prove* it:

- $p(t_1; : : : ; t_n)$: by hypothesis, or definition of p.
- $\neg$: assume , prove contradiction ($\perp$)
- $\vee$ $'$: prove or prove $'$.
- $\wedge$ $'$: prove both and $'$.
- $\exists v$: : provide a witness t for v and show $[v \mapsto t]$.
- $\forall v$: : assume a variable v and show .
- $\Rightarrow$ $'$: assume an hypothesis H : and show $'$.

Example: $\neg \exists x; x < 0$. Where $x < y$ is defined by $\exists n; x + s(n) = y$.
Proof: Assume $H : \exists x; x < 0$. We must prove a $\perp$. The hypothesis H is equivalent to assuming x, n and an hypothesis:
$x + s(n) = 0 \iff s(x + n) = 0$. However,
$\forall n; s(n) \neq 0 \iff \forall n; s(n) = 0 \Rightarrow \perp$, hence a contradiction. □

Valid, invalid, satisfisable or unsatisfiable?

- $_1 = \forall x; y: x \leq y \Rightarrow \exists z: (x \leq z \wedge z < y)$
- $_2 = \exists x: \forall y: x \leq y$
- $_3 = \forall y: x \leq y$
- $_4 = \forall x\ y: x \leq y$
- $_5 = x \leq y \wedge \exists z: y \leq z \wedge z \leq 5$
- $_6 = x = 3$
- $_7 = \exists x; x = y$
- $_8 = \forall x\ y; x \leq y \Rightarrow x < y \vee x = y$
- $_9 = \exists x\ y; x < y \wedge x > y$
- $_{10} = \exists x; x < y$

Example: The head and tail functions

- head function:

$$head \;:\; List[?] \to ?$$
$$Spec\_head(`; a) \;=\; \exists `' \in List[?]: ` = a \cdot `'$$

- tail function:

$$tail \;:\; List[?] \to List[?]$$
$$Spec\_tail(`; `') \;=\; \exists a \in ?: ` = a \cdot `'$$

Multi-sorted Logics

- In general we need to reason about several data domains simultaneously.
- We will consider domains of interpretation of the form

$$(D_1; : : : ; D_n; Op; Rel)$$

  where the operations and relations are defined over one or several of the data domains $D_1; : : : ; D_n$.
- Example: $(List[?]; Nat; \{[]; \cdot; @; Lgth; At; 0; s; +\}; \{=; \leq\})$

Specifying a sorting function

Define an Input-Output relation Spec_Sort(`; `′) ?

- The output list is ordered:

$$Ordered(`) = \forall i; j; \in Nat{:}\ (i < j < |`| \Rightarrow `[i] \leq `[j])$$

- Is it complete ?

Specifying a sorting function (cont.)

- The output list is a permutation of the input list.
- Can we express this property in $FO(List[?]; Nat; \{[]; \cdot; @; Lgth; At; 0; s; +\}; \{=; \leq\})$?
- Every element in the input appears in the output, and vice-versa:
  $\forall i \in Nat{:}\ i < |`_1| \Rightarrow \exists j \in Nat{:}\ (j < |`_2| \wedge `_1[i] = `_2[j])$
  $\wedge\ \forall i \in Nat{:}\ i < |`_2| \Rightarrow \exists j \in Nat{:}\ (j < |`_1| \wedge `_1[i] = `_2[j])$
- Still not sufficient: $`_1 = [2; 5; 2]$ and $`_2 = [2; 5]$
- The input and output lists have the same length: $|`_1| = |`_2|$
- Counter-example: $`_1 = [2; 5; 2]$ and $`_2 = [5; 2; 5]$
- We must count the number of occurrences of each element!

Multisets

- The domain of multisets/bags: $Multiset[?] \equiv ? \rightarrow Nat$
- Operations on multisets:
  - $\emptyset : Multiset[?]$
  - $Sg : ? \rightarrow Multiset[?]$
  - $\uplus : Multiset[?] \times Multiset[?] \rightarrow Multiset[?]$
- Definitions:
  - $\emptyset = \ x \in ?{:}\ 0$
  - $Sg(a) = \ x \in ?{:}\ \texttt{if}\ x = a\ \texttt{then}\ 1\ \texttt{else}\ 0$
  - $M_1 \uplus M_2 = \ x \in ?{:}\ M_1(x) + M_2(x)$
- Example:
  $Sg(0) \uplus (Sg(5) \uplus Sg(0)) =$
  $x \in Nat{:}\ \texttt{if}\ x = 0\ \texttt{then}\ 2\ \texttt{else}\ (\texttt{if}\ x = 5\ \texttt{then}\ 1\ \texttt{else}\ 0)$

Multisets: Properties

- Neutral element: $\emptyset \uplus M = M \uplus \emptyset = M$
- Commutativity: $M_1 \uplus M_2 = M_2 \uplus M_1$
- Associativity: $M_1 \uplus (M_2 \uplus M_3) = (M_1 \uplus M_2) \uplus M_3$
- Proofs: Use properties of natural numbers.

From Lists to Multisets

- Abstracting order in a list:

$$Ms : List[?] \ \to Multiset[?]$$

- Definition:

$$\begin{array}{rcl} Ms([]) & = & \emptyset \\ Ms(a \cdot `) & = & Sg(a) \uplus Ms(`) \end{array}$$

- Example: $Ms(b \cdot a \cdot b \cdot []) =$
  $x \in ?$: if $x = a$ then 1 else if $x = b$ then 2 else 0

From Lists to Multisets (cont.): Properties

- $Ms(`_1 @ `_2) = Ms(`_2 @ `_1) = Ms(`_1) \uplus Ms(`_2)$
- $Ms(Rev(`)) = Ms(`)$

- Proofs: Induction the structure of lists.

Specifying a sorting function (cont.)

$Spec\_Sort(`; `') =$

$$\begin{array}{c} \forall i; j; \in Nat: (i < j < |`| \Rightarrow `'[i] \leq `'[j]) \\ \wedge \\ Ms(`) = Ms(`') \end{array}$$

Inductive Predicates

Inductive predicates give an alternative way to define relations in logic.
To specify evennees we can define *even* : *Nat* $\to$ *prop* inductively by:

$$\begin{array}{rcl} even0 & : & \text{even } 0 \\ evenS & : & \forall n; \text{even } n \Rightarrow \text{even } s(s(n)) \end{array}$$

Compare with the definition: even $n = \exists k; 2 * k = n$

- The two definitions are equivalent.
- Using one or the other depends on the *statement* we want to prove.
- Some properties are easier to express and reason about as inductive predicates.

One can show negative properties easily, i.e. $\neg$even(1): Suppose H : even 1 and try to prove $\bot$. By case analysis on H:

- Case even0: $1 = 0$, by contradiction.
- Case even0: $1 (= s(0)) = s(s(n'))$, by contradiction on $0 = s(n')$.

Inductive Predicates: less-than

For example, $<: Nat \rightarrow Nat \rightarrow prop$ can be defined inductively as:

$$\begin{aligned} lt0 &: \forall x; 0 < s(x) \\ ltS &: \forall x\ y; x < y \Rightarrow s(x) < s(y) \end{aligned}$$

To prove properties about inductive predicates, we can use induction: For example, to prove $\forall x : Nat; x < s(x)$:

Proof: By induction on x:

- Case $x = 0$. We must prove $0 < s(0)$. By $lt00 : 0 < s(0)$.
- Case $x = s(x')$. We have the induction hypothesis $x' < s(x')$.
  We must prove $s(x') < s(s(x'))$.
  We can *apply* $ltS\ x'\ s(x') : x' < s(x') \Rightarrow s(x') < s(s(x'))$ to simplify this to $x' < s(x')$.
  This is the induction hypothesis.

□

Inductive Predicates: less-than

One can also use induction directly on the predicate, in which case we get one case for each constructor of the inductive predicate:
Proving $\forall x\ y : Nat; x < y \Rightarrow 2 * x < 2 * y$:
Proof: By induction on the *hypothesis* $x < y$:

- Case $lt0 : x = 0; y = s(y')$. We must prove $2 * 0 < 2 * s(y')$, by simplification we must prove $0 < s(y' + s(y'))$. By lt0.
- Case $ltS : x = s(x'); y = s(y')$ and induction hypothesis:
  $2 * x' < 2 * y' \leftrightarrow x' + x' < y' + y'$.
  We must prove $2 * s(x') < 2 * s(y')$. By simplification we must prove:
  $s(x' + s(x')) < s(y' + s(y'))$.
  We can apply
  $ltS : x' + s(x') < y' + s(y') \Rightarrow s(x' + s(x')) < s(y' + s(y'))$
  To simplify this to $x' + s(x') < y' + s(y')$.
  By lemmas on addition this is equivalent to $s(x' + x') < s(y' + y')$
  We can apply ltS and the induction hypothesis to conclude.

□

Inductive Predicates: permutation

$\text{perm} : List[?] \rightarrow List[?] \rightarrow prop$ can also be defined inductively using:

$$\begin{aligned} permnil &: \text{perm } [] \ [] \\ permskip &: \forall x\ l\ l'; \text{perm } l\ l' \Rightarrow \text{perm } (x \cdot l)\ (x \cdot l') \\ permswap &: \forall x\ y\ l; \text{perm } (x \cdot y \cdot l)\ (y \cdot x \cdot l) \\ permtrans &: \forall l\ l'\ l''; \text{perm } l\ l' \Rightarrow \text{perm } l'\ l'' \Rightarrow \text{perm } l\ l'' \end{aligned}$$

$$Ms(`) = Ms(`') \iff \text{perm } `\ `'$$

Proofs:

- $\Rightarrow$ by induction on $`$ and $`'$ and case analysis.
- $\Leftarrow$ by induction on the proof perm $`\ `'$

Conclusion

- Specifications are abstract definitions of the effect of functions
- No implementation details are imposed.
- Logic is a natural language for the abstract description of input-output relations
- Abstraction allows modular design:
  - The user of a function needs only to know its specification.
  - The implementor must ensure the satisfaction of the specification.
- There might be different ways to express the same specification, using recursive or inductive predicates.