Abstract Specification of a Function

- Consider a function \( f : \text{Dom} \rightarrow \text{CoDom} \)
- How to describe in an abstract way its behavior?
- Abstraction: No implementation details.
- Specification: A relation \( \text{Spec}_f \) between inputs and outputs of \( f \)

\[
\text{Spec}_f((\text{In}, \text{Out})) \subseteq \text{Dom} \times \text{CoDom}
\]

- What is a suitable (natural) formalism for describing such a relation?

### Logic-based Specification Language

- Example: Specification of the Append function:

\[
\text{Spec}_{\text{Append}}(\ell_1, \ell_2, \ell) = \\
|\ell| = |\ell_1| + |\ell_2| \land \\
\forall i \in \text{Nat}. (i < |\ell_1|) \Rightarrow \ell[i] = \ell_1[i] \land \\
\forall i \in \text{Nat}. (i < |\ell_2|) \Rightarrow \ell[|\ell_1| + i] = \ell_2[i]
\]

where:

\[
\forall \ell \in \text{List}[\ast]. \forall i \in \text{Nat}. \forall e \in \ast. \ell[i] = e \iff \\
(i < |\ell|) \land \\
\exists \ell'. (\ell = a \cdot \ell' \land \\
((i = 0 \land e = a) \lor (i > 0 \land \ell'[i - 1] = e)))
\]

- 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 \( \text{Op} \) be a set of operations interpreted as functions over \( D \)
  - Let \( \text{Pred} \) be a set of predicates interpreted as relations over \( D \)
- Remark: Here the set \( \text{Op} \) may include constants, seen as operators of arity 0.
- Domain of interpretation is a triple \((D, \text{Op}, \text{Ref})\).
- Examples of domains of interpretation:
  - \((\text{Bool}, \{\text{tt, ff, not, or, and, }\}, \{=\})\)
  - \((\text{Nat}, \{0, s, +\}, \{\leq\})\)
  - \((\text{List}[\ast], \{[\ast, @, \emptyset], \{=\}\})\)
First Order Logic over a Data Domain

- Let \((D, Op, Pred)\) be a domain of interpretation.
- Let \(Var\) be a set of variables.
- Terms:
  \[ t ::= v \in Var \mid op(t, \ldots, t) \]
  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 \(\nu\) be a valuation of the variables.
  - Then, \((t)_{\nu}\) is the value in \(D\) obtained by the evaluation of \(t\), using \(\nu\) as valuation of the variables.
  - Example: Given \(\nu = \{(x, 2), (y, 1), (z, 4)\}\), 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

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

\[
\begin{align*}
  x[\nu] & = \nu(x) \\
  p(t_1, \ldots, t_n)[\nu] & = p(t_1[\nu], \ldots, t_n[\nu]) \\
  (\phi \land \phi')[\nu] & = \phi[\nu] \land \phi'[\nu] \\
  (\phi \lor \phi')[\nu] & = \phi[\nu] \lor \phi'[\nu] \\
  (\forall v. \phi)[\nu] & = \forall v. \phi[\nu] \\
  (\exists v. \phi)[\nu] & = \exists v. \phi[\nu] \\
  (\phi \Rightarrow \phi')[\nu] & = \phi[\nu] \Rightarrow \phi'[\nu]
\end{align*}
\]

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

First Order Logic: Semantics of formulas

- Formulas \((\rho \in Pred\) and \(v \in Var\)).
  \[
  \phi ::= T \mid \bot \mid p(t_1, \ldots, t_n) \mid \phi \lor \phi \mid \phi \land \phi \mid \exists v. \phi \mid \forall v. \phi \mid \phi \Rightarrow \phi
  \]
- Abbreviations: \(\neg \phi ::= (\phi \Rightarrow \bot)\), \(\phi \iff \phi' ::= \phi \land \phi' \Rightarrow \phi \land \phi' \Rightarrow \phi\)
- An occurrence of a variable \(x\) is bound in a formula \(\phi\) 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 \((x = 0 \lor 3x, x = 1\) is not well-formed). A variable is free in \(\phi\) if its occurrences in \(\phi\) are unbound. A formula is closed if it has no free variables.
- Examples:
  - \(\phi_1 = \forall x, y. x \leq y \Rightarrow \exists z. (x \leq z \land z < y)\) is a closed formula.
  - \(\phi_2 = \exists x. \forall y. x \leq y\) is a closed formula.
  - \(\phi_3 = \forall y. x \leq y\) is an open formula. It has \(x\) as free variable.
  - \(\phi_4 = x \leq y \land \exists z. y \leq z \land z \leq 5\) is an open formula. Its free variables are \(x\) and \(y\).

First Order Logic: Semantics of formulas

- Given a valuation \(\nu\), we say \(\nu\) satisfies \(\phi\) if and only if \(\phi[\nu]\) is true, i.e., when interpreting the formula using \(\nu\), 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 \([\phi]\) be the set of valuations \(\nu\) which satisfy \(\phi\).
- 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:

- \( p(t_1, \ldots, t_n) \): by hypothesis, or definition of \( p \).
- \( \neg \phi \): assume \( \phi \), prove contradiction (\( \bot \))
- \( \phi \lor \phi' \): prove \( \phi \) or prove \( \phi' \).
- \( \phi \land \phi' \): prove both \( \phi \) and \( \phi' \).
- \( \exists v. \phi \): provide a witness \( t \) for \( v \) and show \( \phi[v \mapsto t] \).
- \( \forall v. \phi \): assume a variable \( v \) and show \( \phi \).
- \( \phi \Rightarrow \phi' \): assume an hypothesis \( H \): \( \phi \) and show \( \phi' \).

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 \( \bot \). The hypothesis \( H \) is equivalent to assuming \( x, n \) and an hypothesis:
\[ x + s(n) = 0 \iff s(x + n) = 0. \]

However, \( \forall n, s(n) \neq 0 \iff \forall n, s(n) = 0 \Rightarrow \bot \), hence a contradiction.

Valid, invalid, satisfiable or unsatisfiable?

- \( \phi_1 = \forall x, y. x \leq y \Rightarrow \exists z. (x \leq z \land z < y) \)
- \( \phi_2 = \exists x. \forall y. x \leq y \)
- \( \phi_3 = \forall y. x \leq y \)
- \( \phi_4 = \forall x, y. x \leq y \)
- \( \phi_5 = x \leq y \land \exists z. y \leq z \land z \leq 5 \)
- \( \phi_6 = x = 3 \)
- \( \phi_7 = \exists x, x = y \)
- \( \phi_8 = \forall x, y. x \leq y \Rightarrow x < y \lor x = y \)
- \( \phi_9 = \exists x, y. x < y \land x > y \)
- \( \phi_{10} = \exists x, x < y \)

Example: The head and tail functions

- head function:
  \[ head : \text{List}[*] \to * \]
  \[ \text{Spec} \_ \text{head}(\ell, a) = \exists \ell' \in \text{List}[*]. \ell = a \cdot \ell' \]

- tail function:
  \[ tail : \text{List}[*] \to \text{List}[*] \]
  \[ \text{Spec} \_ \text{tail}(\ell, \ell') = \exists a \in *. \ell = a \cdot \ell' \]

Multi-sorted Logics

- In general we need to reason about several data domains simultaneously.
- We will consider domains of interpretation of the form
  \[ (D_1, \ldots, D_n, Op, Rel) \]
  where the operations and relations are defined over one or several of the data domains \( D_1, \ldots, D_n \).
- Example: \( (\text{List}[*], \text{Nat}, \{[], \cdot, @, \text{Lgth}, \text{At}, 0, s, +\}, \{=, \leq\}) \)
Specifying a sorting function

Define an Input-Output relation $\text{Spec}_\text{Sort}(\ell, \ell')$?

- The output list is ordered:
  \[ \forall i, j \in \text{Nat}. \ (i < j < |\ell| \implies \ell[i] \leq \ell[j]) \]
- Is it complete?

The output list is ordered:

\[ \text{Ordered}(\ell) = \forall i, j \in \text{Nat}. \ (i < j < |\ell| \implies \ell[i] \leq \ell[j]) \]

Is it complete?

M. Sozeau (Inria & UP7)
L2: Logic-based Program Specification
October 2014 13 / 1

Specifying a sorting function (cont.)

The output list is a permutation of the input list.

Can we express this property in $\text{FO(List}[\star], \text{Nat}, [\{\}, \cdot, \text{@}, \text{Lgth}, \text{At}, 0, s, +], \{=, \leq\})$?

Every element in the input appears in the output, and vice-versa:

\[
\forall i \in \text{Nat}. \ i < |\ell_1| \implies \exists j \in \text{Nat}. \ (j < |\ell_2| \land \ell_1[i] = \ell_2[j]) \\
\land \forall i \in \text{Nat}. \ i < |\ell_2| \implies \exists j \in \text{Nat}. \ (j < |\ell_1| \land \ell_1[i] = \ell_2[j])
\]

Still not sufficient: $\ell_1 = [2, 5, 2]$ and $\ell_2 = [2, 5]$

The input and output lists have the same length: $|\ell_1| = |\ell_2|$

Counter-example: $\ell_1 = [2, 5, 2]$ and $\ell_2 = [5, 2, 5]$

We must count the number of occurrences of each element!

M. Sozeau (Inria & UP7)
L2: Logic-based Program Specification
October 2014 14 / 1

Multisets

- The domain of multisets/bags: $\text{Multiset}[\star] \equiv \star \rightarrow \text{Nat}$
- Operations on multisets:
  - $\emptyset : \text{Multiset}[\star]$
  - $\text{Sg} : \star \rightarrow \text{Multiset}[\star]$
  - $\cup : \text{Multiset}[\star] \times \text{Multiset}[\star] \rightarrow \text{Multiset}[\star]$

Definitions:

- $\emptyset = \lambda x \in \star. \ 0$
- $\text{Sg}(a) = \lambda x \in \star. \ \text{if } x = a \ \text{then } 1 \ \text{else } 0$
- $M_1 \cup M_2 = \lambda x \in \star. \ M_1(x) + M_2(x)$

Example:

$\text{Sg}(0) \cup (\text{Sg}(5) \cup \text{Sg}(0)) = \lambda x \in \text{Nat}. \ \text{if } x = 0 \ \text{then } 2 \ \text{else } (\text{if } x = 5 \ \text{then } 1 \ \text{else } 0)$

Multisets: Properties

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

M. Sozeau (Inria & UP7)
L2: Logic-based Program Specification
October 2014 15 / 1

M. Sozeau (Inria & UP7)
L2: Logic-based Program Specification
October 2014 16 / 1
From Lists to Multisets

- Abstracting order in a list:
  \[ Ms : List[^\star] \rightarrow Multiset[^\star] \]

- Definition:
  \[ Ms([]) = \emptyset \]
  \[ Ms(a \cdot \ell) = Sg(a) \uplus Ms(\ell) \]

- Example: \[ Ms(b \cdot a \cdot b \cdot []) = \lambda x \in \star. \text{if } x = a \text{ then } 1 \text{ else if } x = b \text{ then } 2 \text{ else } 0 \]

From Lists to Multisets (cont.): Properties

- \[ Ms(\ell_1 \circ \ell_2) = Ms(\ell_2 \circ \ell_1) = Ms(\ell_1) \uplus Ms(\ell_2) \]
- \[ Ms(Rev(\ell)) = Ms(\ell) \]

Proofs: Induction the structure of lists.

Specifying a sorting function (cont.)

\[ Spec\_Sort(\ell, \ell') = \forall i, j, \in Nat. (i < j < |\ell| \Rightarrow \ell'[i] \leq \ell'[j]) \]
\[ \land Ms(\ell) = Ms(\ell') \]

Inductive Predicates

Inductive predicates give an alternative way to define relations in logic. To specify evenness we can define \( even : Nat \rightarrow prop \) inductively by:

\[
\begin{align*}
even0 & : even\ 0 \\
evenS & : \forall n, even\ n \Rightarrow even\ s(n)
\end{align*}
\]

Compare with the definition: \( even\ n = \exists k, 2 \cdot k = n \)

- 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: \( 1 = s(0) = s(n') \), by contradiction on \( 0 = s(n') \).
Inductive Predicates: less-than

For example, $\lt : \text{Nat} \to \text{Nat} \to \text{prop}$ can be defined inductively as:

$$\lt 0 : \forall x, 0 < s(x)$$
$$\lt S : \forall x \ y, x < y \Rightarrow s(x) < s(y)$$

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

**Proof:** By induction on $x$:
- Case $x = 0$. We must prove $0 < s(0)$. By $\lt 00 : 0 < s(0)$.
- Case $x = s(x')$. We have the induction hypothesis $x' < s(x')$.
  We must prove $s(x') < s(s(x'))$.
  We can apply $\lt S x' s(x') : x' < s(x') \Rightarrow s(x') < s(s(x'))$ to simplify this to $x' < s(x')$.
  This is the induction hypothesis.

Inductive Predicates: permutation

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

$$\text{permnil} : \text{perm} \ [ \ ] \ [ \ ]$$
$$\text{permskip} : \forall x \ l \ l', \text{perm} l \ l' \Rightarrow \text{perm} (x \cdot l) (x \cdot l')$$
$$\text{permswap} : \forall x \ y \ l, \text{perm} (x \cdot y \cdot l) (y \cdot x \cdot l)$$
$$\text{permtrans} : \forall l \ l' \ l'', \text{perm} l \ l' \Rightarrow \text{perm} l' \ l'' \Rightarrow \text{perm} l \ l''$$

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

Proofs:
- $\Rightarrow$ by induction on $\ell$ and $\ell'$ and case analysis.
- $\Leftarrow$ by induction on the proof $\text{perm} \ \ell \ \ell'$

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 $\forall x \ y : \text{Nat}, x < y \Rightarrow 2 \ast x < 2 \ast y$:

**Proof:** By induction on the hypothesis $x < y$:
- Case $\lt 0 : x = 0, y = s(y')$. We must prove $2 \ast 0 < 2 \ast s(y')$, by simplification we must prove $0 < s(y' + s(y'))$. By $\lt 0$.
- Case $\lt S : x = s(x'), y = s(y')$ and induction hypothesis:
  $2 \ast x' < 2 \ast y'$ and induction hypothesis:
  $2 \ast x' < 2 \ast y' \iff x' + x' < y' + y'$.
  We must prove $2 \ast s(x') < 2 \ast s(y')$. By simplification we must prove:
  $s(x' + s(x')) < s(y' + s(y'))$.
  We can apply $\lt S : x' + s(x') < y' + s(y') \Rightarrow s(x' + s(x')) < s(y' + s(y'))$.
  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 $\lt S$ and the induction hypothesis to conclude.

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.