First Order Predicate Logic

Formal Definitions and Specifications

Wolfgang Schreiner and Wolfgang Windsteiger
Wolfgang.(Schreiner|Windsteiger)@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University (JKU), Linz, Austria
http://www.risc.jku.at

Defining and Specifying

Specifying problems and domains is a core activity of computer science.

- Goal: the adequate specification of a certain "problem" or "type".
  - A computation to be performed.
  - A domain of values to be represented.
  - Specification is to be expressed using the notions of some "model".
- Given: a "model", i.e., a collection of notions (functions/predicates).
  - For example, the model "set" with the usual set operations.
  - The interpretation of these notions is universally understood.
- Issue: the given model is not up to the task.
  - Its notions are on a too low level of abstraction.
  - The specification would become too cumbersome to write and too difficult to understand.

We need a model that is on an appropriate level of abstraction.

Refinement and Abstraction

How to overcome the gap between the given model and the intended one?

- Top-down (↓): refinement.
  - Start with the intended model.
  - Reduce its notions to lower-level notions.
  - Iterate, until the lowest level of the given notions is reached.
- Bottom-up (↑): abstraction.
  - Start with the given model.
  - Iteratively combine the given notions to higher-level notions.
  - Iterate, until the highest level of the intended notions is reached.
- Bottom-up and top-down (↕):
  - Combination of refinement and abstraction steps.
  - Iterate, until the refined notions "meet" the abstracted ones.

With the help of newly defined notions, problems and types may be adequately specified.

Illustration

Databases

File Systems Text Files XML Documents

Character Strings Floating Point Numbers

Arrays Lists Trees Integers

Refinement Abstraction

Natural Numbers

Functions Relations

Products Sums

Sets

Some Standard Models

- Products: $T_1 \times \ldots \times T_n$
  - Let $x_1 \in T_1, \ldots, x_n \in T_n, t \in T_1 \times \ldots \times T_n$.
  - Tuple construction: $(x_1, \ldots, x_n) \in T_1 \times \ldots \times T_n$.
  - Element selection: $t.1 \in T_1, \ldots, t.n \in T_n$ (or: $t_1 \in T_1, \ldots, t_n \in T_n$).
- Functions: $T_1 \times \ldots \times T_n \to T$
  - Let $x_1 \in T_1, \ldots, x_n \in T_n, f \in T_1 \times \ldots \times T_n \to T$.
  - Function definition: see later.
  - Function application: $f(x_1, \ldots, x_n) \in T$.
  - $domain(f) = T_1 \times \ldots \times T_n$, $range(f) \subseteq T$.
- Relations/Predicates: $\mathcal{P}(T_1 \times \ldots \times T_n)$

  $$\mathcal{P}(T): \textit{the powerset (the set of all subsets) of } T.$$

  - Let $x_1 \in T_1, \ldots, x_n \in T_n, p \in \mathcal{P}(T_1 \times \ldots \times T_n)$ ($p \subseteq T_1 \times \ldots \times T_n$).
  - Predicate definition: see later.
  - Predicate application: $p(x_1, \ldots, x_n)$ denotes a truth value.
  - $domain(p) = T_1 \times \ldots \times T_n$.

Can be reduced to set-theoretic notions.

Some Standard Models

- Infinite Sequences: $T^\omega = \mathbb{N} \to T$
  - Let $s \in T^\omega, i \in \mathbb{N}$.
  - Element access $s(i) \in T$ (or: $s_i \in T$).
- Sequences of Length n: $T^n = \mathbb{N}_n \to T$
  - Index domain: $\mathbb{N}_n = \{i \in \mathbb{N} \mid i < n\}$.
  - Let $s \in T^n, i \in \mathbb{N}_n$.
  - Element access: $s(i) \in T$ (or: $s_i \in T$).
- Finite Sequences: $T^* = \bigcup_{n \in \mathbb{N}} T^n$
  - Let $s \in T^*$, i.e., $s \in T^n$ for some $n \in \mathbb{N}$, let $i \in \mathbb{N}_n$.
  - Sequence length: $length(s) = n$.
  - Element access: $s(i) \in T$ (or: $s_i \in T$).

Sequences (arrays, lists, ...) of arbitrary length can be modeled as functions over an index domain.

Formal Definitions and Specifications

- Explicit Function Definitions.
- Explicit Predicate Definitions.
- Implicit Function Definitions.
- Algebraic Data Type Definitions.
- Defining by Structural Induction.
- Problem Specifications.

Explicit Function Definitions

A new function my be introduced by describing its value.

- An explicit function definition

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(x_1, \ldots, x_n) := t$$

  consists of
  - a new n-ary function constant f,
  - a type signature $T_1 \times \ldots \times T_n \to T$ withs sets $T_1, \ldots, T_n, T$,
  - a list of variables $x_1, \ldots, x_n$ (the parameters), and
  - a term t (the body) whose free variables occur in $x_1, \ldots, x_n$;
  - case $n = 0$: the definition of a value constant $f : T, f := t$.
- We have to show for the newly introduced function f

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : t \in T$$

  and then know

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : f(x_1, \ldots, x_n) = t$$

The body of an explicit function definition may only refer to *previously* defined functions (no recursion).

Examples

$$sqrtsum : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$$
$$sqrtsum(x, y) := \sqrt{x} + \sqrt{y}$$

$$sumsquared : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$sumsquared(x, y) := \textbf{let } z = x + y \textbf{ in } z^2$$

$$sqrtsumsquared : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$$
$$sqrtsumsquared(x, y) := sqrtsum(x, y)^2$$

$$sqrtsumset : \mathbb{N} \to \mathcal{P}(\mathbb{R})$$
$$sqrtsumset(n) := \{sqrtsum(x, y) \mid x, y \in \mathbb{N} \wedge 1 \leq x, y \leq n\}$$

Explicit Predicate Definitions

A new predicate my be introduced by describing its truth value.

- An explicit predicate definition

$$p \subseteq T_1 \times \ldots \times T_n$$
$$p(x_1, \ldots, x_n) :\Leftrightarrow F$$

  consists of

  - a new n-ary predicate constant p,
  - a type signature $T_1 \times \ldots \times T_n$ with sets $T_1, \ldots, T_n$
  - a list of variables $x_1, \ldots, x_n$ (the parameters), and
  - a formula F (the body) whose free variables occur in $x_1, \ldots, x_n$.
  - case $n = 0$: definition of a truth value constant $p :\Leftrightarrow F$.
- We then know for the newly introduced predicate p:

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : p(x_1, \ldots, x_n) \leftrightarrow F$$

The body of an explicit predicate definition may only refer to *previously* defined predicates (no recursion).

Definitions with Side Conditions

- A definition may occur in the context of a side condition.

  Let $x_1 \in T_1, \ldots, x_n \in T_n$ *be such that* $c(x_1, \ldots, x_n)$. *We define*

$$f(x_1, \ldots, x_n) := t$$
$$p(x_1, \ldots, x_n) :\Leftrightarrow F$$

- We then know for the newly introduced function/predicate

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : c(x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n) = t$$
$$\forall x_1 \in T_1, \ldots, x_n \in T_n : c(x_1, \ldots, x_n) \rightarrow (p(x_1, \ldots, x_n) \leftrightarrow F)$$

Applications of the function/predicate to arguments that violate the side condition are meaningless.

Examples

$$| \subseteq \mathbb{N} \times \mathbb{N}$$
$$x|y :\Leftrightarrow \exists z \in \mathbb{N} : x \cdot z = y$$

$$\mathit{isprime} \subseteq \mathbb{N}$$
$$\mathit{isprime}(x) :\Leftrightarrow x \geq 2 \wedge \forall y \in \mathbb{N} : 1 < y \wedge y < x \rightarrow \neg(y|x)$$

$$\mathit{isprimefactor} \subseteq \mathbb{N} \times \mathbb{N}$$
$$\mathit{isprimefactor}(p, n) :\Leftrightarrow \mathit{isprime}(p) \wedge p|n$$

Implicit Function Definitions

We may also introduce a new function by describing what condition its result must satisfy.

- An implicit function definition

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(x_1, \ldots, x_n) := \textbf{such } y : F$$

  consists of

  - a new n-ary function constant f,
  - a type signature $T_1 \times \ldots \times T_n \to T$ withs sets $T_1, \ldots, T_n, T$,
  - a list of variables $x_1, \ldots, x_n$ (the parameters),
  - a variable y (the result variable),
  - a formula F (the result condition) whose free variables occur in $x_1, \ldots, x_n, y$.

- We then know for the newly introduced function f

$$\forall x_1 \in T_1, \ldots, x_n \in T_n :$$
$$(\exists y \in T : F) \to (\exists y \in T : F \wedge y = f(x_1, \ldots, x_n))$$

If there is some value that satisfies the result condition, the function result is one such value (otherwise, it is undefined).

Examples

$quotient : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ // undefined for n=0, otherwise unique

$quotient(m,n) :=$ **such** $q : \exists r \in \mathbb{N} : m = n \cdot q + r \wedge r < n$

$root : \mathbb{R} \to \mathbb{R}$ // defined for non-negative x, but not unique

$root(x) :=$ **such** $r : r^2 = x$

$someprimefactor : \mathbb{N} \to \mathbb{N}$ // may be undefined; if defined, may not be unique

$someprimefactor(n) :=$ **such** $p : isprimefactor(p,n)$

$maxprime : \mathbb{N} \to \mathbb{N}$ // may be undefined; if defined, it is unique

$maxprime(n) :=$ **such** $p : isprime(p) \wedge p \leq n \wedge$

$$(\forall q \in \mathbb{N} : isprime(q) \wedge q \leq n \to q \leq p)$$

The result of an implicitly specified function is not necessarily uniquely defined (and may be also completely undefined).

Implicit Unique Function Definitions

But sometimes the result is uniquely defined by an implicit definition.

- An implicit unique function definition

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(x_1, \ldots, x_n) := \textbf{the } y : F$$

  consists of the same elements as an unique function definition.

- We have to prove that the function result is defined and unique

$$\begin{array}{l} \forall x_1 \in T_1, \ldots, x_n \in T_n : \\ \quad (\exists y \in T : F) \land \\ \quad (\forall y_1 \in T, y_2 \in T : F[y_1/y] \land F[y_2/y] \to y_1 = y_2) \end{array}$$

  from which we know for the newly introduced function f

$$\begin{array}{l} \forall x_1 \in T_1, \ldots, x_n \in T_n : \\ \quad (\exists y \in T : F \land y = f(x_1, \ldots, x_n)) \land \\ \quad (\forall y \in T : F \to y = f(x_1, \ldots, x_n)) \end{array}$$

The function result is the only value that satisfies the result condition.

Examples

$$quot : \mathbb{R}_{\geq 0} \times \mathbb{R}_{>0} \to \mathbb{R}_{\geq 0} \text{ // defined and unique}$$
$$quot(a, b) := \textbf{the } q : a = b \cdot q$$

$$posroot : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \text{ // defined and unique}$$
$$posroot(x) := \textbf{the } r : r^2 = x \wedge r \geq 0$$

$$minimum : \mathcal{P}(\mathbb{N}) \backslash \{\emptyset\} \to \mathbb{N} \text{ // defined and unique}$$
$$minimum(S) := \textbf{the } m : m \in S \wedge \forall m' \in S : m' \geq m$$

Informal Definitions

- Definition: A *tomcat* is a male cat.

$$tomcat(x) :\Leftrightarrow cat(x) \wedge male(x).$$

- Definition: Let x, y be positive integers. Then $gcd(x, y)$ denotes the greatest positive integer that divides both x and y.

$$gcd : \mathbb{Z}_{>0} \times \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$$
$$gcd(x, y) := \textbf{the } z : z|x \wedge z|y \wedge \forall z' \in \mathbb{Z}_{>0} : z'|x \wedge z'|y \to z' \leq z$$

- Definition: A *prime factorization* of $n > 1$ is a product $p_1^{e_1} \cdots p_n^{e_n} = n$ with primes $p_1 < \ldots < p_n$ and exponents $e_i > 0$.

$$isPrimeFactorization \subseteq \mathbb{N} \times (\mathbb{N} \times \mathbb{N})^*$$
$$isPrimeFactorization(n, p) :\Leftrightarrow$$
$$\textbf{let } l = length(p) \textbf{ in}$$
$$n = \prod_{i=0}^{l-1} (p(i).1)^{(p(i).2)} \wedge$$
$$(\forall i \in \mathbb{N}_l : prime(p(i).1)) \wedge$$
$$(\forall i \in \mathbb{N}_{l-1} : p(i).1 < p(i+1).1)$$

It is important to recognize the formal content of informal definitions.

Predicates versus Functions

A predicate gives also rise to functions.

- A predicate:

$$isprimefactor \subseteq \mathbb{N} \times \mathbb{N}$$
$$isprimefactor(p, n) :\Leftrightarrow isprime(p) \wedge p|n$$

- An implicitly defined function:

$$someprimefactor : \mathbb{N} \rightarrow \mathbb{N}$$
$$someprimefactor(n) := \textbf{such } p : isprime(p) \wedge p|n$$

- A function whose result is a set:

$$allprimefactors : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$$
$$allprimefactors(n) := \{p \in \mathbb{N} \mid isprime(p) \wedge p|n\}$$

The preferred style of definition is a matter of taste and purpose.

Type Definitions

Frequently sets used as types (value domains) have a particular structure.

- The def. of a type with (free) constructors (an algebraic type)

$$\textbf{type } T \; := \; c_1(T_{\sqcup}, \ldots, T_{\sqcup}) \; + \; \ldots \; + \; c_n(T_{\sqcup}, \ldots, T_{\sqcup})$$

  consists of new function constants $c_1, \ldots, c_n$ (the constructors) where each constructor c_i receives type signature $c_i : T_{\sqcup} \times \ldots \times T_{\sqcup} \rightarrow T$.
  - A constructor $c()$ is written as c and has type signature $c : T$.
- T is then the set of terms that can be built from the constructors.
  - Different terms $c_i(\ldots)$ denote different elements of T.
- T itself may appear in the type signatures of the constructors.
  - Types with infinitely many values may be constructed.
- Multiple types may be simultaneously defined.
  - The type signatures of their constructors may refer to any of the defined types.

An algebraic data type is the set of constructor terms.

Examples

- **type** $Bool := true + false$
  - Constructors: $true, false : Bool$.
  - $Bool = \{true, false\}$
- **type** $Nat := 0 + s(Nat)$
  - Constructors: $0 : Nat, s : Nat \rightarrow Nat$
  - $Nat = \{0, s(0), s(s(0)), \ldots\}$
- **type** $NatList := empty + cons(Nat, NatList)$
  - Constructors: $empty : NatList, cons : Nat \times NatList \rightarrow NatList$
  - $NatList = \{empty, cons(0, empty), \ldots, cons(s(0), cons(0, empty)), \ldots\}$
- **type** $Tree(E) := nil + node(E, Tree(E), Tree(E))$
  - Constructors: $nil : Tree(E), node : E \times Tree(E) \times Tree(E) \rightarrow Tree(E)$
  - $Tree(E) = \{\ldots, node(e_1, node(e_2, nil, nil), nil), \ldots\}$
- **types** $A := a + r(B), B := b + s(A)$
  - Constructors: $a : A, r : B \rightarrow A, b : B, s : A \rightarrow B$
  - $A = \{a, r(b), r(s(a)), \ldots\}, B = \{b, s(a), s(r(b)), \ldots\}$

All data types whose values can be described by terms.

Defining by Structural Induction

- A primitive recursive definition on $\mathbb{N}$:

$$
\begin{aligned}
&! : \mathbb{N} \rightarrow \mathbb{N} \\
&0! := 1 \\
&(n+1)! := (n+1) \cdot n!
\end{aligned}
$$

- The domain $\mathbb{N}$ of natural numbers is isomorphic to the algebraic type

$$\textbf{type } \mathit{Nat} := 0 + s(\mathit{Nat})$$

- We can define on "Nat" by structural induction the analog function

$$
\begin{aligned}
&\mathit{fact} : \mathit{Nat} \rightarrow \mathit{Nat} \\
&\mathit{fact}(0) := s(0) \\
&\mathit{fact}(s(n)) := \mathit{mult}(s(n), \mathit{fact}(n))
\end{aligned}
$$

Primitive recursion is a special case of structural induction.

Defining by Structural Induction

Given an algebraic data type

$$\textbf{type } T \ := \ c_1(\ldots) \ + \ \ldots \ + c_n(\ldots)$$

with n constructors one may define a function $f : \ldots \times T \times \ldots \to \ldots$ by n equations of form

$$\begin{aligned} & f(\ldots, c_1(\ldots), \ldots) := t_1 \\ & \ldots \\ & f(\ldots, c_n(\ldots), \ldots) := t_n \end{aligned}$$

with n terms $t_1, \ldots, t_n$ where

- only distinct variables occur in the positions "..." of each "pattern" $f(\ldots, c_i(\ldots), \ldots)$,
- the free variables in each term t_i occur as variables of the corresponding pattern,
- in every term t_i the function f must not be applied to the constructor term $c_i(\ldots)$ on the left side (but only to some variable inside).

Defining a function (possibly recursively) by "pattern matching".

Meaning

- Given an algebraic type

$$\textbf{type } T \;:=\; c_1(\ldots) \;+\; \ldots \;+ c_n(\ldots)$$

- by structural induction with defining equations

$$\begin{aligned} & f(\ldots, c_1(\ldots), \ldots) := t_1 \\ & \ldots \\ & f(\ldots, c_n(\ldots), \ldots) := t_n \end{aligned}$$

- a new function f is introduced that satisfies the axioms

$$\begin{aligned} & \forall \ldots :\; f(\ldots, c_1(\ldots), \ldots) = t_1, \\ & \ldots, \\ & \forall \ldots :\; f(\ldots, c_n(\ldots), \ldots) = t_n \end{aligned}$$

where $\forall \ldots$ binds all variables that appear in the respective pattern.

The function value is uniquely defined for all arguments.

Defining by Structural Induction

Given an algebraic data type

$$\textbf{type}\ T\ :=\ c_1(\ldots)\ +\ \ldots\ + c_n(\ldots)$$

with n constructors one may define a predicate $p \subseteq \ldots \times T \times \ldots$ by n equivalences of form

$$p(\ldots, c_1(\ldots), \ldots) :\Leftrightarrow F_1$$
$$\ldots$$
$$p(\ldots, c_n(, \ldots), \ldots) :\Leftrightarrow F_n$$

with n formulas $F_1, \ldots, F_n$ where

- only distinct variables occur in the positions "…" of each "pattern" $p(\ldots, c_i(\ldots), \ldots)$,
- the free variables in each formula F_i occur as variables of the corresponding pattern,
- in every formula F_i, the predicate p must not be applied to the term $c_i(\ldots)$ on the left side (but only to some variable inside).

Defining a predicate (possibly recursively) by "pattern matching".

Meaning

- Given an algebraic type with constructors

$$\textbf{type}\ T \ := \ c_1(\ldots) \ + \ \ldots \ + c_n(\ldots)$$

- by structural induction with defining equivalences

$$\begin{aligned} & p(\ldots, c_1(\ldots), \ldots) :\Leftrightarrow F_1 \\ & \ldots \\ & p(\ldots, c_n(\ldots), \ldots) :\Leftrightarrow F_n \end{aligned}$$

- a new predicate p is introduced that satisfies the axioms

$$\begin{aligned} & \forall \ldots : \ p(\ldots, c_1(\ldots), \ldots) \leftrightarrow F_1 \\ & \ldots, \\ & \forall \ldots : \ p(\ldots, c_n(\ldots), \ldots) \leftrightarrow F_n \end{aligned}$$

where $\forall \ldots$ binds all variables that appear in the respective pattern.

The predicate value is uniquely defined for all arguments.

Example

A list of elements of type T.

$$\textbf{type } List(T) := empty + cons(T, List(T))$$

$$length : List(T) \to \mathbb{N}$$
$$length(empty) := 0$$
$$length(cons(x, l)) := 1 + length(l)$$

$$append : List(T) \times List(T) \to List(T)$$
$$append(empty, l2) := l2$$
$$append(cons(x, l1), l2) := cons(x, append(l1, l2))$$

$$has \subseteq List(T) \times T$$
$$has(empty, e) :\Leftrightarrow \bot$$
$$has(cons(x, l), e) :\Leftrightarrow x = e \lor has(l, e)$$

Specifying Problems

An important role of logic in computer science is to specify problems.

- The specification of a (computational) problem

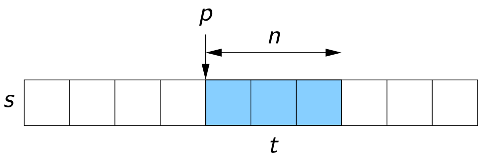
$$\textbf{Input: } x_1 \in T_1, \ldots, x_n \in T_n \quad \textbf{where } I$$
$$\textbf{Output: } y_1 \in U_1, \ldots, y_m \in U_m \textbf{ where } O$$

  consists of
  - a list of input variables $x_1, \ldots, x_n$ with types $T_1, \ldots, T_n$,
  - a formula I (the input condition) whose free variables occur in $x_1, \ldots, x_n$,
  - a list of output variables $y_1, \ldots, y_m$ with types $U_1, \ldots, U_m$, and
  - a formula O (the output condition) whose free variables occur in $x_1, \ldots, x_n, y_1, \ldots, y_m$.

The specification is expressed with the help of functions and predicates that have been previously defined to describe the problem domain.

Example

Extract from a finite sequence s of natural numbers a subsequence of length n starting at position p.



Input: $s \in \mathbb{N}^*, n \in \mathbb{N}, p \in \mathbb{N}$ **where**

$n + p \leq length(s)$

Output: $t \in \mathbb{N}^*$ **where**

$length(t) = n \wedge$

$\forall i \in \mathbb{N}_n : t(i) = s(i + p)$

The resulting sequence must have appropriate length and content.

Implementing Problem Specifications

The ultimate goal of computer science is to implement specifications.

- The specifications demands the definition of a function $f : T_1 \times \ldots \times T_n \to U_1 \times \ldots \times U_m$ such that

$$\forall x_1 \in T_1, \ldots, x_n \in T_n : I \to \\ \textbf{let } (y_1, \ldots, y_m) = f(x_1, \ldots, x_n) \textbf{ in } O$$

  - For all arguments $x_1, \ldots, x_n$ that satisfy the input condition,
  - the result $(y_1, \ldots, y_m)$ of f satisfies the output condition.
- The specification itself already implicitly defines such a function:

$$f(x_1, \ldots, x_n) := \textbf{such } y_1, \ldots, y_m : I \to O$$

- However, the specification is actually implemented only by an explicitly defined function (computer program).

  The correctness of the implementation with respect to the specification has to be verified (e.g. by a formal proof).

Our goal is to adequately specify informal problems, to implement formal specifications, and to verify the correctness of the implementations.

The Java Modeling Language (JML)

A language for specifying the contracts of Java functions.

```
/*@ requires s != null && 0 <= p && 0 <= n && p+n <= s.length;
  @ ensures  \result != null && \result.length == n &&
  @          (\forall int i; 0 <= i && i < n;
  @             \result[i] == s[i+p]);
  @*/
/*@ pure @*/ static int[] subarray(int[] s, int p, int n) {
  int[] t = new int[n];
  for (int i=0; i<n; i++)
    t[i] = s[i+p];
  return t;
}
```

The Java function implements the specified contract.