Formal Methods and Functional Programming - Week 2 (Lectures)

- Author: Ruben Schenk
- Date: 08.03.2021
- Contact: ruben.schenk@inf.ethz.ch

Substitution

`Substitution` describes replacing in A all occurrences of a *free variable* x with some term t. We write $A[x/t]$ to indicate that we substitute x by t in A.

Example:
$A \equiv \exists y.\, y * x = x * z \to A[x/2-1] \equiv \exists y.\, y * (2-1) = (2-1 * z)$

Universal quantification

We introduce the following two rules:

$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \forall - I^*, \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x/t]} \forall - E$

For the insertion rule, the side condition * denotes that x cannot be free in any assumption Γ.

Example:

$$\dfrac{\dfrac{\dfrac{\overline{\forall x.\, p(x) \vdash \forall z.\, p(z)}\ \textit{axiom}}{\forall x.\, p(x) \vdash p(f(y))}\ \forall\text{-}E}{\forall x.\, p(x) \vdash \forall y.\, p(f(y))}\ \forall\text{-}I}{\vdash (\forall x.\, p(x)) \to (\forall y.\, p(f(y)))}\ \to\text{-}I$$

implicit α-conversion

with $x \equiv z$ and $t \equiv f(y)$

y not free in $\forall x.\, p(x)$

Existential quantification

We introduce the following two rules:
$\frac{\Gamma \vdash A[x/t]}{\Gamma \vdash \exists x - A} \exists - I, \qquad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \exists - E^*$

For the elimination rule, the side condition * denotes that x is neither free in B nor free in Γ.

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \exists z.\, p(z)}\,\textit{axiom} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \forall w.\, p(w) \to q}\,\textit{axiom}}{\Gamma' \vdash \quad p(z) \quad \to q}\,\forall\text{-}E \qquad \dfrac{}{\Gamma' \vdash \quad p(z)}\,\textit{axiom}}{\Gamma, p(z) \vdash q}\,\to\text{-}E}{\forall x.\, p(x) \to q, \exists y.\, p(y) \vdash q}\,\exists\text{-}E\ ^{*}}{\forall x.\, p(x) \to q \vdash (\exists y.\, p(y)) \to q}\,\to\text{-}I}{\vdash (\forall x.\, p(x) \to q) \to ((\exists y.\ p(y)) \to q)}\,\to\text{-}I$$

where $\Gamma \equiv \forall x.p(x) \to q, \exists y.\, p(y)$ and $\Gamma' \equiv \Gamma, p(z)$

Equality

First order logic with equality

Equality is logical symbol with associated proof rules. We define it with:

- Extended language: $t_1 = t_2 \in \mathit{Form}$ if $t_1, t_2 \in \mathit{Term}$
- Extended definition of $\models$: $\mathcal{I} \models t_1 = t_2$ if $\mathcal{I}(t_1) = \mathcal{I}(t_2)$

Equality

`Equality` is an equivalence rule, shown by the following three rules:

$$\frac{}{\Gamma \vdash t = t}\, ref, \qquad \frac{\Gamma \vdash t = s}{\Gamma \vdash s = t}\, sym, \qquad \frac{\Gamma \vdash t = s \quad \Gamma \vdash s = r}{\Gamma \vdash t = r}\, trans$$

Equality is also a `congruence` on terms and all definable relations:

$$\frac{\Gamma \vdash t_1 = s_1 \quad \cdots \quad \Gamma \vdash t_n = s_n}{\Gamma \vdash f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)}\, \mathrm{cong}_1$$

$$\frac{\Gamma \vdash t_1 = s_1 \quad \cdots \quad \Gamma \vdash t_n = s_n \quad \Gamma \vdash p(t_1, \ldots, t_n)}{\Gamma \vdash p(s_1, \ldots, s_n)}\, \mathrm{cong}_2$$

Correctness

Correctness is important for programs. But what does `correctness` mean? What properties should hold?

- `Termination`: Important for many, but not all programs
- `Functional behavior`: Function should return "correct" values.

Termination

If f is defined in terms of functions $g_1, \ldots, g_k$, and each g_j terminates, then so does f. If we work with recursion, a sufficient condition for termination is: Arguments are smaller along a well-founded order of function's domain.

- An order $>$ on a set $\mathcal{S}$ is `well-founded` iff. there is no infinite decreasing chain chain $x_1 > x_2 > x_3 > \cdots$, for $x_i \in \mathcal{S}$
- We write $>_{\mathcal{S}}$ to indicate the domain $\mathcal{S}$, i.e. $>_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$

Well-founded relations

We can construct new well-founded relations from existing ones the following way:

Let R_1 and R_2 be binary relations on a set S. The composition of R_1 and R_2 is defined as

$$R_2 \circ R_1 \equiv \{(a,\, c) \in S \times S \,|\, \exists b \in S.\, a\, R_1\, b \wedge b\, R_2\, c\}.$$

Remark: For a binary relation R, we write $a\, R\, b$ for $(a,\, b) \in R$.

Let $R \subseteq S \times S$. Define:

- $R^1 \equiv R$
- $R^{n+1} \equiv R \circ R^n$, for $n \geq 1$
- $R^+ \equiv \bigcup_{n \geq 1} R^n$

So $a\, R^+\, b$ iff. $a\, R^i\, b$ for some $i \geq 1$.

> Lemma: Let $R \subseteq S \times S$. Let $s_0, s_i \in S$ and $i \geq 1$. Then $s_0\, R^i\, s_i$ iff- there are $s_1, .., s_{i-1} \in S$ such that $s_0\, R\, s_1\, R \ldots R\, s_{i-1}\, R\, s_i$.

> Theorem: If $>$ is a well-founded order on a set S, then $>^+$ is also well-founded on S.

Termination: Example

Consider the following two Haskell snippets for calculating the factorial of a (positive) integer:

```
fac 0 = 1
fac n = n * fac (n-1)

-------------------------

fac2 (0, a) = a
fac2 (n, a) = fac2 (n-1, n*a)
```

We know look at the above examples in terms of termination:

- function `fac`:
  - `fac n` has only `fac (n-1)` as a recusrive call and $n > n - 1$, hence $>$ is the standard ordering over the natural numbers.
- function `fac2`:
  - `fac2 (n, a)` has only `fac2 (n-1, n*a)` as a recursive call and the first argument is always smaller under $>$.

Reasoning

Equational reasoning

`Equational reasoning` include proofs based on the simple idea, that functions are equations.

For example we can have the following simple Haskell program:

```haskell
swap :: (Int, Int) -> (Int, Int)
swap (a, b) = (b, a)
```

More formally:

$\forall a \in \mathcal{Z}, \forall b \in \mathcal{Z}, \text{swap}(a, b) = (b, a)$

Reasoning by cases

Consider the following Haskell snippet:

```haskell
maxi :: Int -> Int -> Int
maxi n m
    | n >= m    = n
    | otherwise = m
```

Can we prove that $\text{maxi}\, n\, m \geq n$?

- We have that $n \geq m \vee \neg(n \geq m)$
- We now show $\text{maxi}\, n\, m \geq n$ for both cases:
    - Case 1: $n \geq m$, then $\text{maxi}\, n\, m = n$ and $n \geq n$
    - Case 2: $\neg(n \geq m)$, then $\text{maxi}\, n\, m = m$. But $m > n$, so $\text{maxi}\, n\, m \geq n$

Proof by induction

We use a domino principle formulated by induction proof rule.

Example: To prove $\forall n \in \mathcal{N}. P$

- Base case: Prove $P[n/0]$
- Step case: For an arbitrary m not free in P, prove $P[n/m+1]$ under the assumption $P[n/m]$.

Well-founded induction

The induction schema for well-founded induction is given by:

- To prove P for all natural numbers n
- Well-founded step: For an arbitrary m (not free in P), prove $P[n/m]$ under the assumption that $P[n/l]$ holds, for all $l < m$ (where also l is not free in P).

List and Abstraction

List type

Lists require a new type constructor:

- If T is a type, then $[T]$ is a type.

The elements of $[T]$ are given by:

- Empty list: $[] :: [T]$
- Non-empty list: $(x : xs) :: [T]$, if $x :: T$ and $xs :: [T]$

In Haskell we can use the following shorthand: $1 : (2 : (3 : []))$ written as $[1,\ 2,\ 3]$.

Functions on lists

We should always include the case where the list is empty!

Example: `sumList`

```
sumList []      = 0
sumList (x:xs)  = x + sumList xs
```

Patterns

Pattern matching has two purposes:

1. checks if an argument has the proper form
2. binds values to variables

Example: $(x : xs)$ matches with $[2,\ 3,\ 4] \to x = 2,\ xs = [3,\ 4]$.

Patterns are inductively defined with:

- Constants: $-2,\ '1',\ True,\ [], \ldots$
- Variables: $x,\ foo, \ldots$
- Wild card: _
- Tuples: $(p_1, \ldots, p_k)$, where p_i are patterns
- Non-empty lists: $(p_1 : p_2)$, where p_i are patterns

Moreover, patterns require to be `linear`, this means that each variable can occur at most once.