SS 2023

Program Verification

Part 3 – Semantics of Functional Programs

René Thiemann

Department of Computer Science

Overview

- definition of a small functional programming language
- operational semantics
- a model in many-sorted logic
- derived inference rules

Functional Programming – Data Types

Data Type Definitions

- a functional program contains a sequence of data type definitions
- while processing the sequence, we determine the set of types $\mathcal{T}y$, the signature Σ, and the predicates $\mathcal{P}$, which are all initially empty
- each data type definition has the following form

$$\begin{aligned} \textsf{data}\ \tau = {} & c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau \\ \mid {} & \ldots \\ \mid {} & c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau \end{aligned}$$

where

  - $\tau \notin \mathcal{T}y$ — fresh type name
  - $c_1, \ldots, c_n \notin \Sigma$ and $c_i \neq c_j$ for $i \neq j$ — fresh and distinct constructor names
  - each $\tau_{i,j} \in \{\tau\} \cup \mathcal{T}y$ — only known types
  - exists c_i such that $\tau_{i,j} \in \mathcal{T}y$ for all j — non-recursive constructor
- effect: add type, constructors and equality predicate
  - $\mathcal{T}y := \mathcal{T}y \cup \{\tau\}$
  - $\Sigma := \Sigma \cup \{c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau, \ldots, c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau\}$
  - $\mathcal{P} := \mathcal{P} \cup \{=_\tau \subseteq \tau \times \tau\}$

Data Type Definitions: Examples

- $\mathcal{T}y = \Sigma = \mathcal{P} = \varnothing$
- data Nat = Zero : Nat | Succ : Nat → Nat
- processing updates $\mathcal{T}y = \{\mathsf{Nat}\}$,
  $\Sigma = \{\mathsf{Zero} : \mathsf{Nat}, \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}\}$
  and $\mathcal{P} = \{=_{\mathsf{Nat}} \subseteq \mathsf{Nat} \times \mathsf{Nat}\}$
- data List = Nil : List | Cons : Nat × List → List
- processing updates $\mathcal{T}y = \{\mathsf{Nat}, \mathsf{List}\}$,
  $\Sigma = \{\mathsf{Zero} : \mathsf{Nat}, \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{Nil} : \mathsf{List}, \mathsf{Cons} : \mathsf{Nat} \times \mathsf{List} \to \mathsf{List}\}$
  and $\mathcal{P} = \{=_{\mathsf{Nat}} \subseteq \mathsf{Nat} \times \mathsf{Nat}, =_{\mathsf{List}} \subseteq \mathsf{List} \times \mathsf{List}\}$
- data BList = NilB : BList | ConsB : Bool × BList → BList
  not allowed, since $\mathsf{Bool} \notin \mathcal{T}y$
- data LList = Nil : LList | Cons : List × LList → LList
  not allowed, since Nil and Cons are already in Σ
- data Tree = Node : Tree × Nat × Tree → Tree
  not allowed, since all constructors are recursive

Data Type Definitions: Standard Model

- while processing data type definitions we also build a model $\mathcal{M}$ for the functional program, called the standard model
- when processing

$$\begin{aligned} \mathsf{data}\ \tau = {} & c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau \\ | {} & \ldots \\ | {} & c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau \end{aligned}$$

  - define universe $\mathcal{A}_\tau$ for new type τ inductively via the following inference rules (one for each $1 \leq i \leq n$)

$$\frac{t_1 \in \mathcal{A}_{\tau_{i,1}} \quad \ldots \quad t_{m_i} \in \mathcal{A}_{\tau_{i,m_i}}}{c_i(t_1, \ldots, t_{m_i}) \in \mathcal{A}_\tau}$$

  - define $c_i^{\mathcal{M}}(t_1, \ldots, t_{m_i}) = c_i(t_1, \ldots, t_{m_i})$ — uninterpreted constructors
  - define $=_\tau^{\mathcal{M}} = \{(t, t) \mid t \in \mathcal{A}_\tau\}$ — equality

Data Type Definitions: Example and Standard Model

- data Nat = Zero : Nat | Succ : Nat → Nat
- processing creates universe $\mathcal{A}_{\mathsf{Nat}}$ via the inference rules

$$\frac{}{\mathsf{Zero} \in \mathcal{A}_{\mathsf{Nat}}} \qquad \frac{t \in \mathcal{A}_{\mathsf{Nat}}}{\mathsf{Succ}(t) \in \mathcal{A}_{\mathsf{Nat}}}$$

  i.e., $\mathcal{A}_{\mathsf{Nat}} = \{\mathsf{Zero}, \mathsf{Succ}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero})), \ldots\}$
- $\mathsf{Zero}^{\mathcal{M}} = \mathsf{Zero} \qquad \mathsf{Succ}^{\mathcal{M}}(t) = \mathsf{Succ}(t)$
- $=_{\mathsf{Nat}}^{\mathcal{M}} = \{(\mathsf{Zero}, \mathsf{Zero}), (\mathsf{Succ}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero})), \ldots\}$
- data List = Nil : List | Cons : Nat × List → List
- processing creates universe $\mathcal{A}_{\mathsf{List}}$ via the inference rules

$$\frac{}{\mathsf{Nil} \in \mathcal{A}_{\mathsf{List}}} \qquad \frac{t_1 \in \mathcal{A}_{\mathsf{Nat}} \quad t_2 \in \mathcal{A}_{\mathsf{List}}}{\mathsf{Cons}(t_1, t_2) \in \mathcal{A}_{\mathsf{List}}}$$

  i.e., $\mathcal{A}_{\mathsf{List}} = \{\mathsf{Nil}, \mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil}), \mathsf{Cons}(\mathsf{Succ}(\mathsf{Zero}), \mathsf{Nil}), \ldots\}$
- $=_{\mathsf{List}}^{\mathcal{M}} = \{(\mathsf{Nil}, \mathsf{Nil}), (\mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil}), \mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil})), \ldots\}$

Well-Definedness of Standard Model

- question: is the standard model really a model in the sense of many-sorted logic
  - is there a unique type for each $c_i \in \Sigma$ and $=_\tau \in \mathcal{P}$
  - are the definitions of $c_i^{\mathcal{M}}$ and $=_\tau^{\mathcal{M}}$ well-defined
  - are the definitions of $\mathcal{A}_\tau$ well-defined, i.e., $\mathcal{A}_\tau \neq \varnothing$
- recall: each data definition has the following form

$$\begin{aligned} \mathsf{data}\ \tau = {} & c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau \\ | {} & \ldots \\ | {} & c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau \end{aligned}$$

  where
  - $\tau \notin \mathcal{T}y$ — fresh type name
  - $c_1, \ldots, c_n \notin \Sigma$ and $c_i \neq c_j$ for $i \neq j$ — fresh and distinct constructor names
  - each $\tau_{i,j} \in \{\tau\} \cup \mathcal{T}y$ — only known types
  - exists c_i such that $\tau_{i,j} \in \mathcal{T}y$ for all j — non-recursive constructor
- what could happen if one of the conditions is dropped?

Non-Empty Universes

- without the last condition (non-recursive constructor) the following data type declaration would be allowed (assuming that Nat and Succ are fresh names)

$$\text{data Nat} = \text{Succ} : \text{Nat} \to \text{Nat}$$

with the universe defined as the inductive set $\mathcal{A}_{\text{Nat}}$

$$\frac{t \in \mathcal{A}_{\text{Nat}}}{\text{Succ}(t) \in \mathcal{A}_{\text{Nat}}}$$

- consequence: $\mathcal{A}_{\text{Nat}} = \varnothing$
- hence, non-recursive constructors are essential for having non-empty universes

Non-Empty Universes: Proof

Theorem

Let there be a list of data type declarations and an arbitrary type τ from this list. Then $\mathcal{A}_\tau \neq \varnothing$.

Proof

Let $\tau_1, \dots, \tau_n$ be the sequence of types that have been defined. We show

$$P(n) := \forall 1 \leq i \leq n.\ \mathcal{A}_{\tau_i} \neq \varnothing$$

by induction on n. This will entail the theorem.
In the base case we have to prove $P(0)$, which is trivially true. Now let us show $P(n+1)$ assuming $P(n)$. Because of $P(n)$, we only have to prove $\mathcal{A}_{\tau_{n+1}} \neq \varnothing$. By the definition of data types, there must be some $c_i : \tau_{i,1} \times \ldots \times \tau_{i,m_i} \to \tau_{n+1}$ where all $\tau_{i,j} \in \{\tau_1, \dots, \tau_n\}$. By the IH $P(n)$ we know that $\mathcal{A}_{\tau_{i,j}} \neq \varnothing$ for all j between 1 and m_i. Hence, there must be terms $t_1 \in \mathcal{A}_{\tau_{i,1}}, \ldots, t_{m_i} \in \mathcal{A}_{\tau_{i,m_i}}$. Consequently, $c_i(t_1, \dots, t_{m_i}) \in \mathcal{A}_{\tau_{n+1}}$, and hence $\mathcal{A}_{\tau_{n+1}} \neq \varnothing$.

Current State

- presented: data type definitions
- semantics
  - free constructors: each constructor is interpreted as itself
  - universe as inductively defined sets: no infinite terms, such as infinite lists $\text{Cons}(\text{Zero}, \text{Cons}(\text{Zero}, \ldots))$
    (modeling of infinite data structures would be possible via domain-theory)
- upcoming: functional programs, i.e., function definitions

Functional Programming – Function Definitions

Splitting the signature

- distinguish between
  - constructors, declared via data (capital letters in Haskell)
    e.g., Nil, Succ, Cons
  - defined functions, declared via equations (lowercase letters in Haskell)
    e.g., append, add, reverse
- formally, we have $\Sigma = \mathcal{C} \uplus \mathcal{D}$
- $\mathcal{C}$ is set of constructors, defined via data
  - constructors are written c, c_i, d in generic constructs such as data type definitions
  - start with uppercase letters in concrete examples (Succ, Cons)
- $\mathcal{D}$ is set of defined symbols, defined via function declarations
  - defined (function) symbols are written f, f_i, g in generic constructs such as function definitions
  - start with lowercase letters in concrete examples (append, reverse)
- we use F, G for elements of Σ whenever separation between $\mathcal{C}$ and $\mathcal{D}$ is not relevant
- note that in the standard model, $\mathcal{A}_\tau$ is exactly $\mathcal{T}(\mathcal{C})_\tau := \mathcal{T}(\mathcal{C}, \varnothing)_\tau$, which is the set of constructor ground terms of type τ

Functional Programming – Function Definitions

Notions for Preparing Function Definitions

- a pattern is a term in $\mathcal{T}(\mathcal{C}, \mathcal{V})$, usually written p or p_i
- a term t in $\mathcal{T}(\Sigma, \mathcal{V})$ is linear, if all variables within t occur only once
  - $\mathsf{reverse}(\mathsf{Cons}(x, \mathsf{Cons}(y, xs)))$ ✔
  - $\mathsf{reverse}(\mathsf{Cons}(x, \mathsf{Cons}(x, xs)))$ ✘
- the variables of a term t are defined as $\mathcal{V}ars(t)$
  - $\mathcal{V}ars(x) = \{x\}$
  - $\mathcal{V}ars(F(t_1, \dots, t_n)) = \mathcal{V}ars(t_1) \cup \ldots \cup \mathcal{V}ars(t_n)$

Functional Programming – Function Definitions

Function Definitions

- besides data type definitions, a functional program consists of a sequence of function definitions, each having the following form

$$\begin{aligned} f &: \tau_1 \times \ldots \times \tau_n \to \tau \\ \ell_1 &= r_1 \\ \ldots &= \ldots \\ \ell_m &= r_m \end{aligned}$$

where

- f is a fresh name and $\mathcal{D} := \mathcal{D} \cup \{f : \tau_1 \times \ldots \times \tau_n \to \tau\}$ (hence, f is also added to $\Sigma = \mathcal{C} \cup \mathcal{D}$)
- each left-hand side (lhs) ℓ_i is linear
- each lhs ℓ_i is of the form $f(p_1, \dots, p_n)$ with all p_j's being patterns
- each lhs ℓ_i and rhs r_i only use currently known symbols: $\ell_i, r_i \in \mathcal{T}(\Sigma, \mathcal{V})$
- each lhs ℓ_i and rhs r_i respect the type: $\ell_i, r_i \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$
- each equation $\ell_i = r_i$ satisfies the variable condition $\mathcal{V}ars(r_i) \subseteq \mathcal{V}ars(\ell_i)$

Functional Programming – Function Definitions

Function Definitions: Examples

- assume data types Nat and List have been defined as before (slide 5)

$$\begin{aligned} &\mathsf{add} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\ &\mathsf{add}(\mathsf{Zero}, y) = y \\ &\mathsf{add}(\mathsf{Succ}(x), y) = \mathsf{add}(x, \mathsf{Succ}(y)) \end{aligned}$$

$$\begin{aligned} &\mathsf{append} : \mathsf{List} \times \mathsf{List} \to \mathsf{List} \\ &\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys)) \\ &\mathsf{append}(xs, ys) = ys \end{aligned}$$

$$\begin{aligned} &\mathsf{head} : \mathsf{List} \to \mathsf{Nat} \\ &\mathsf{head}(\mathsf{Cons}(x, xs)) = x \end{aligned}$$

$$\begin{aligned} &\mathsf{zeros} : \mathsf{List} \\ &\mathsf{zeros} = \mathsf{Cons}(\mathsf{Zero}, \mathsf{zeros}) \end{aligned}$$

Function Definitions: Non-Examples

- assume program from previous slides + data Bool = True | False

$$\begin{aligned}
&\mathsf{even} : \mathsf{Nat} \to \mathsf{Bool} \\
&\mathsf{even}(\mathsf{Zero}) = \mathsf{True} \\
&\mathsf{even}(\mathsf{Succ}(x)) = \mathsf{odd}(x) && ✘ \\
&\mathsf{odd} : \mathsf{Nat} \to \mathsf{Bool} \\
&\mathsf{odd}(\mathsf{Zero}) = \mathsf{False} \\
&\mathsf{odd}(\mathsf{Succ}(x)) = \mathsf{even}(x) && ✘ \\
&\mathsf{random} : \mathsf{Nat} \\
&\mathsf{random} = x && ✘ \\
&\mathsf{minus} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\
&\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) = \mathsf{minus}(x, y) \\
&\mathsf{minus}(x, \mathsf{Zero}) = x \\
&\mathsf{minus}(x, x) = \mathsf{Zero} && ✘ \\
&\mathsf{minus}(\mathsf{add}(x, y), x) = y && ✘
\end{aligned}$$

Semantics for Function Definitions

- problem: given a function definition

$$\begin{aligned}
f &: \tau_1 \times \ldots \times \tau_n \to \tau \\
\ell_1 &= r_1 \\
\ldots &= \ldots \\
\ell_m &= r_m
\end{aligned}$$

we need to extend the semantics in the standard model, i.e., define the function

$$f^{\mathcal{M}} : \mathcal{A}_{\tau_1} \times \ldots \times \mathcal{A}_{\tau_n} \to \mathcal{A}_\tau$$

or equivalently

$$f^{\mathcal{M}} : \mathcal{T}(\mathcal{C})_{\tau_1} \times \ldots \times \mathcal{T}(\mathcal{C})_{\tau_n} \to \mathcal{T}(\mathcal{C})_\tau$$

- idea: define $f^{\mathcal{M}}(t_1, \ldots, t_n)$ as

  the result of $f(t_1, \ldots, t_n)$ after evaluation w.r.t. equations in program

Semantics for Function Definitions – Continued

- required: $f^{\mathcal{M}} : \mathcal{T}(\mathcal{C})_{\tau_1} \times \ldots \times \mathcal{T}(\mathcal{C})_{\tau_n} \to \mathcal{T}(\mathcal{C})_\tau$
- idea: define $f^{\mathcal{M}}(t_1, \ldots, t_n)$ as

  the result of $f(t_1, \ldots, t_n)$ after evaluation w.r.t. equations in program
- several issues:
  - how is term evaluation defined?
    - briefly: replace instances of lhss by instances of rhss as long as possible
  - is result unique?
  - is result element of $\mathcal{T}(\mathcal{C})_\tau$?
  - does evaluation terminate?

Function Definitions: Examples

- consider previous program, type declarations omitted

$$\begin{aligned}
\mathsf{add}(\mathsf{Zero}, y) &= y && (1) \\
\mathsf{add}(\mathsf{Succ}(x), y) &= \mathsf{add}(x, \mathsf{Succ}(y)) && (2) \\
\mathsf{append}(\mathsf{Cons}(x, xs), ys) &= \mathsf{Cons}(x, \mathsf{append}(xs, ys)) && (3) \\
\mathsf{append}(xs, ys) &= ys && (4) \\
\mathsf{head}(\mathsf{Cons}(x, xs)) &= x && (5) \\
\mathsf{zeros} &= \mathsf{Cons}(\mathsf{Zero}, \mathsf{zeros}) && (6)
\end{aligned}$$

- is result unique? no: consider $t = \mathsf{append}(\mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil}), \mathsf{Nil})$

  then $t \overset{(3)}{=} \mathsf{Cons}(\mathsf{Zero}, \mathsf{append}(\mathsf{Nil}, \mathsf{Nil})) \overset{(4)}{=} \mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil})$

  and $t \overset{(4)}{=} \mathsf{Nil}$
- is result element of $\mathcal{T}(\mathcal{C})_\tau$? no: $\mathsf{head}(\mathsf{Nil})$ cannot be evaluated
- does evaluation terminate? no: $\mathsf{zeros} = \mathsf{Cons}(\mathsf{Zero}, \mathsf{zeros}) = \ldots$
- solution: further restrictions on function definitions

Functional Programming – Operational Semantics

Functional Programming: Operational Semantics

- operational semantics: formal definition on how evaluation proceeds step-by-step
- main operation: applying a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ to a term, can be defined recursively
  - $x\sigma = \sigma(x)$
  - $F(t_1, \ldots, t_n)\sigma = F(t_1\sigma, \ldots, t_n\sigma)$
- one-step evaluation relation $\hookrightarrow \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$ defined as inductive set

$$\frac{\ell = r \text{ is equation in program}}{\ell\sigma \hookrightarrow r\sigma} \text{ root step}$$

$$\frac{F \in \Sigma \quad s_i \hookrightarrow t_i}{F(s_1, \ldots, s_i, \ldots, s_n) \hookrightarrow F(s_1, \ldots, t_i, \ldots, s_n)} \text{ rewrite in context}$$

- given a term t and a lhs ℓ, for checking whether a root-step is applicable one needs matching: $\exists \sigma. \ell\sigma = t$ (and also deliver that σ)
- same evaluation as in functional programming (lecture), except that order of equations is ignored and here it becomes formal

Matching

- we define matching as an operation on a set of pairs $P = \{(\ell_1, t_1), \ldots, (\ell_n, t_n)\}$ and the task is to decide: $\exists \sigma. \ell_1\sigma = t_1 \wedge \ldots \wedge \ell_n\sigma = t_n$, i.e.,
  - either return the required substitution σ in the form of a set of pairs $\{(x_1, s_1), \ldots, (x_m, s_m)\}$ with all x_i distinct which can then be interpreted as the substitution σ defined by

$$\sigma(x) = \begin{cases} s_i, & \text{if } x = x_i \text{ for some } i \\ x, & \text{otherwise} \end{cases}$$

  - or return $\bot$ indicating that no such substitution exists
- matching algorithm
  - if P contains a pair $(F(\ell_1, \ldots, \ell_n), F(t_1, \ldots, t_n))$, then replace this pair by the n pairs $(\ell_1, t_1), \ldots, (\ell_n, t_n)$ — decompose
  - if P contains $(F(\ldots), G(\ldots))$ with $F \neq G$, then return $\bot$ — clash
  - if P contains $(F(\ldots), x)$ with $x \in \mathcal{V}$, then return $\bot$ — fun-var
  - if P contains (x, s) and (x, t) with $x \in \mathcal{V}$ and $s \neq t$, then return $\bot$ — var-clash
  - if none of the above rules is applicable, then return P

Matching – Example

- we want to test whether there is a root step possible for the term $t = \mathsf{append}(\mathsf{Cons}(y, \mathsf{Nil}), \mathsf{Cons}(y, ys))$ w.r.t. the equation $(\ell = r) = (\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys)))$
- setup matching problem $\{(\ell, t)\}$ $P = \{(\mathsf{append}(\mathsf{Cons}(x, xs), ys), \mathsf{append}(\mathsf{Cons}(y, \mathsf{Nil}), \mathsf{Cons}(y, ys)))\}$
- decomposition: $P = \{(\mathsf{Cons}(x, xs), \mathsf{Cons}(y, \mathsf{Nil})), (ys, \mathsf{Cons}(y, ys))\}$
- decomposition: $P = \{(x, y), (xs, \mathsf{Nil}), (ys, \mathsf{Cons}(y, ys))\}$
- obtain substitution $\sigma(z) = \begin{cases} y, & \text{if } z = x \\ \mathsf{Nil}, & \text{if } z = xs \\ \mathsf{Cons}(y, ys), & \text{if } z = ys \\ z, & \text{otherwise} \end{cases}$
- so, $t = \ell\sigma \hookrightarrow r\sigma = \mathsf{Cons}(x, \mathsf{append}(xs, ys))\sigma = \mathsf{Cons}(y, \mathsf{append}(\mathsf{Nil}, \mathsf{Cons}(y, ys)))$

Matching – Verification and Termination Proof

- matching algorithm
  - whenever P contains a pair $(F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n))$, replace this pair by the n pairs $(\ell_1,t_1),\dots,(\ell_n,t_n)$ decompose
  - ...
- soundness = termination + partial correctness
- termination: in each step, the sum of the size of terms (# symbols) is decreased

$$\begin{aligned}|(F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n))| &= |F(\ell_1,\dots,\ell_n)| + |F(t_1,\dots,t_n)|\\ &= 1 + \sum_i |\ell_i| + 1 + \sum_i |t_i|\\ &> \sum_i |\ell_i| + \sum_i |t_i|\\ &= \sum_i |(\ell_i,t_i)|\end{aligned}$$

Matching – Type Preservation

- matching algorithm
  - whenever P contains a pair $(F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n))$, replace this pair by the n pairs $(\ell_1,t_1),\dots,(\ell_n,t_n)$ decompose
  - ...
- property: we say that a set of pairs P is type-correct, iff for all pairs $(\ell,t)\in P$ the types of ℓ and t are identical, i.e., $\exists\tau.\,\{\ell,t\}\subseteq\mathcal{T}(\Sigma,\mathcal{V})_\tau$
- theorem: whenever P is type-correct, then P will stay type-correct during the algorithm; consequently, any result $\neq\bot$ will be type-correct
- proof: we prove an invariant, so we only need to prove that the property is maintained when performing a step in the algorithm:
  consider "decompose"
  - we can assume $\{F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n)\}\subseteq\mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - so $F:\tau_1\times\ldots\times\tau_n\to\tau$ for suitable τ_i
  - hence, $\{\ell_i,t_i\}\subseteq\mathcal{T}(\Sigma,\mathcal{V})_{\tau_i}$ for all i

Matching – Structure of Result

- matching algorithm
  - whenever P contains $(F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n))$ ... decompose
  - whenever P contains $(F(...),G(...))$ with $F\neq G$, then return $\bot$ clash
  - whenever P contains $(F(...),x)$ with $x\in\mathcal{V}$, then return $\bot$ fun-var
  - whenever P contains (x,s) and (x,t) with $x\in\mathcal{V}$ and $s\neq t$ then return $\bot$ var-clash
  - when none of the above rules is applicable, return P
- property: result of matching algorithm on well-typed inputs is $\bot$ or set $\{(x_1,s_1),\dots,(x_m,s_m)\}$ with all x_i distinct
- proof
  - assume result is not $\bot$, then it must be some set of pairs $P=\{(u_1,s_1),\dots,(u_m,s_m)\}$ where no rule is applicable
  - if all u_i's are variables, then the result follows: there cannot be two entries (u_i,s_i) and (u_j,s_j) with $u_i=u_j$ and $s_i\neq s_j$ because then "var-clash" would have been applied
  - it remains to consider the case that some $u_i=F(\ell_1,\dots,\ell_n)$
  - $s_i=F(t_1,\dots,t_k)$, as result is not $\bot$, cf. "clash" and "fun-var"
  - then $k=n$ because of type preservation: contradiction to "decompose"

Matching – Preservation of Solutions

- matching algorithm
  - whenever P contains a pair $(F(\ell_1,\dots,\ell_n),F(t_1,\dots,t_n))$, replace this pair by the n pairs $(\ell_1,t_1),\dots,(\ell_n,t_n)$ decompose
  - whenever P contains $(F(...),G(...))$ with $F\neq G$, then return $\bot$ clash
  - whenever P contains $(F(...),x)$ with $x\in\mathcal{V}$, then return $\bot$ fun-var
  - whenever P contains (x,s) and (x,t) with $x\in\mathcal{V}$ and $s\neq t$ then return $\bot$ var-clash
  - when none of the above rules is applicable, return P
- property: algorithm preserves matching substitutions (where $\bot$ has no matching substitution)
- proof via invariant: whenever P is changed to P', then σ is a matcher of P iff σ is matcher of P'
  - clash: both "σ is matcher of $\{(F(...),G(...))\}\cup P$" and "$\sigma$ is matcher of $\bot$" are wrong: $F(t_1,\dots)\sigma=F(t_1\sigma,\dots)\neq G(...)$
  - fun-var and var-clash are similar
  - decompose: $F(\ell_1,\dots,\ell_n)\sigma=F(t_1,\dots,t_n)$
    $\longleftrightarrow F(\ell_1\sigma,\dots,\ell_n\sigma)=F(t_1,\dots,t_n)$
    $\longleftrightarrow \ell_1\sigma=t_1\wedge\ldots\wedge\ell_n\sigma=t_n$

Matching Algorithm – Summary

- algorithm: apply certain steps until no longer possible
- (one) termination proof
- (many) partial correctness proofs
  mainly by showing an invariant that is preserved by each step
  - type preservation
  - preservation of matching substitutions
  - result is $\bot$ or a set which encodes a substitution
- application: compute root steps by testing whether decomposition of term into $\ell\sigma$ for equation $\ell = r$ is possible
- core of functional programming (and term rewriting)
- much better algorithms exists, which avoid to match against all lhss, based on precalculation (term indexing), e.g., group equations by root symbol of lhss

Semantics in the Standard Model

Towards Semantics in Standard Model

- evaluation of terms is now explained: one-step relation $\hookrightarrow$
- algorithm for evaluation is similar to matching algorithm:

  apply $\hookrightarrow$-steps until no longer possible
- questions are similar as in matching algorithm
  - termination: do we always get result?
  - preservation of types?
  - is result a desired value, i.e., a constructor ground term?
  - is result unique?
- questions don't have positive answer in general, cf. slide 20

Type Preservation of $\hookrightarrow$

- aim: show that $\hookrightarrow$ preserves types:

$$t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau \longrightarrow t \hookrightarrow s \longrightarrow s \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$$

- proof will be by induction w.r.t. inductively defined set $\hookrightarrow$ for arbitrary τ
- preliminary: we call a substitution type-correct, if $\sigma(x) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ whenever $x : \tau \in \mathcal{V}$
- easy result: whenever $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ and σ is type-correct, then $t\sigma \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$
  (how would you prove it?)

Type Preservation of $\hookrightarrow$ – Proof

- proof: induction w.r.t. inductively defined set $\hookrightarrow$ for arbitrary τ
- base case: $\ell\sigma \hookrightarrow r\sigma$ for some equation $\ell = r$ of the program where $\ell\sigma \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ and we have to prove $r\sigma \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - since $\ell\sigma \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$, and $\ell, r \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ by the definition of functional programs, we conclude that σ is type-correct, cf. slide 26
  - and since $r \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ and σ is type-correct, then also $r\sigma \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$, cf. previous slide
- step case: $F(s_1,\ldots,s_i,\ldots,s_n) \hookrightarrow F(s_1,\ldots,t_i,\ldots,s_n)$ since $s_i \hookrightarrow t_i$, we know $F(s_1,\ldots,s_i,\ldots,s_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ and have to prove $F(s_1,\ldots,t_i,\ldots,s_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - since $F(s_1,\ldots,s_i,\ldots,s_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$, we know that $F : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma$ and each $s_j \in \mathcal{T}(\Sigma,\mathcal{V})_{\tau_j}$ for $1 \leq j \leq n$
  - by the IH we know $t_i \in \mathcal{T}(\Sigma,\mathcal{V})_{\tau_i}$ – note that here we can take a different type than τ, namely τ_i, because the induction was for arbitrary τ
  - but then we immediately conclude $F(s_1,\ldots,t_i,\ldots,s_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$

Type Preservation of $\hookrightarrow^*$

- finally, we can show that evaluation (execution of arbitrarily many $\hookrightarrow$-steps, written $\hookrightarrow^*$) preserves types, which is an easy induction proof by the number of steps, using type-preservation of $\hookrightarrow$
- theorem: whenever $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ and $t \hookrightarrow^* s$, then $s \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
- proofs to obtain global result
  1. show that matching preserves types (slide 26)
     proof via invariant, since matching algorithm is imperative (while rules-applicable ...)
  2. show that substitution application preserves types (slide 31)
     proof by induction on terms, following recursive structure of definition of substitution application (slide 22)
  3. show that $\hookrightarrow$ preserves types (slide 33)
     proof by structural induction w.r.t. inductively defined set $\hookrightarrow$;
     uses results 1 and 2
  4. show that $\hookrightarrow^*$ preserves types
     proof on number of steps; uses result 3

Preservation of Groundness of $\hookrightarrow^*$

- a term t is ground if $\mathcal{V}ars(t) = \varnothing$, or equivalently if $t \in \mathcal{T}(\Sigma)$
- recall aim: we want to evaluate ground term like append(Cons(Zero, Nil), Nil) to element of universe, i.e., constructor ground term
- hence, we need to ensure that result of evaluation with $\hookrightarrow$ is ground
- preservation of groundness can be shown with similar proof structure as in the proof of preservation of types

Normal Forms – The Results of an Evaluation

- a term t is a normal form (w.r.t. $\hookrightarrow$) if no further $\hookrightarrow$-steps are possible:
$$\nexists s.\ t \hookrightarrow s$$
- whenever $t \hookrightarrow^* s$ and s is in normal form, then we write
$$t \hookrightarrow^! s$$
  and call s a normal form of t
- normal forms represent the result of an evaluation
- known results at this point: whenever $t \in \mathcal{T}(\Sigma)_\tau$ and $t \hookrightarrow^! s$ then
  - $s \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ (type-preservation)
  - $s \in \mathcal{T}(\Sigma)$ (groundness-preservation)
  - $s \in \mathcal{T}(\Sigma)_\tau$ (combined)
- missing:
  - $s \in \mathcal{T}(\mathcal{C})_\tau$ (constructor-ground term)
  - s is unique
  - s always exists

Pattern Completeness

- a function symbol $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathcal{D}$ is pattern complete iff for all $t_1 \in \mathcal{T}(\mathcal{C})_{\tau_1}$, ..., $t_n \in \mathcal{T}(\mathcal{C})_{\tau_n}$ there is an equation $\ell = r$ in the program, such that ℓ matches $f(t_1, \ldots, t_n)$
- a functional program is pattern complete iff all $f \in \mathcal{D}$ are pattern complete
- example

$$\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys))$$
$$\mathsf{append}(\mathsf{Nil}, ys) = ys$$
$$\mathsf{head}(\mathsf{Cons}(x, xs)) = x$$

  - append is pattern complete
  - head is not pattern complete: for head(Nil) there is no matching lhs

Pattern Completeness and Constructor Ground Terms

- theorem: if a program is pattern complete and $t \in \mathcal{T}(\Sigma)_\tau$ is a normal form, then $t \in \mathcal{T}(\mathcal{C})_\tau$
- proof of $P(t, \tau)$ by structural induction w.r.t. $\mathcal{T}(\Sigma)_\tau$ for

$$P(t, \tau) := t \text{ is normal form} \longrightarrow t \in \mathcal{T}(\mathcal{C})_\tau$$

  - induction yields only one case: $t = F(t_1, \ldots, t_n)$ where $F : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma$
  - IH for each i: if t_i is normal form, then $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$
  - premise: $F(t_1, \ldots, t_n)$ is normal form
  - from premise conclude that t_i is normal form: (if $t_i \hookrightarrow s_i$ then $F(t_1, \ldots, t_n) \hookrightarrow F(t_1, \ldots, s_i, \ldots, t_n)$ shows that $F(t_1, \ldots, t_n)$ is not a normal form)
  - in combination with IH: each $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$
  - consider two cases: $F \in \mathcal{C}$ or $F \in \mathcal{D}$
  - case $F \in \mathcal{C}$: using $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$ immediately yields $F(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{C})_\tau$
  - case $F \in \mathcal{D}$: using pattern completeness and $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$, conclude that $F(t_1, \ldots, t_n)$ must be matched by lhs; this is contradiction to $F(t_1, \ldots, t_n)$ being a normal form

Pattern Disjointness

- a function symbol $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathcal{D}$ is pattern disjoint iff for all $t_1 \in \mathcal{T}(\mathcal{C})_{\tau_1}$, ..., $t_n \in \mathcal{T}(\mathcal{C})_{\tau_n}$ there is at most one equation $\ell = r$ in the program, such that ℓ matches $f(t_1, \ldots, t_n)$
- a functional program is pattern disjoint iff all $f \in \mathcal{D}$ are pattern disjoint
- example

$$\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys))$$
$$\mathsf{append}(xs, ys) = ys$$
$$\mathsf{head}(\mathsf{Cons}(x, xs)) = x$$

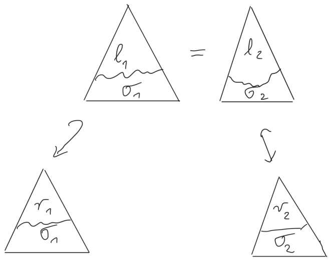
  - head is pattern disjoint
  - append is not pattern disjoint: the term append(Cons(Zero, Nil), Nil) is matched by the lhss of both append-equations

Pattern Disjointness and Unique Normal Forms

- theorem: if a program is pattern disjoint then $\hookrightarrow$ is confluent and each term has at most one normal form
- confluence: whenever $s \hookrightarrow^* t$ and $s \hookrightarrow^* u$ then there exists some v such that $t \hookrightarrow^* v$ and $u \hookrightarrow^* v$
- proof of theorem:
  - pattern disjointness in combination with the other syntactic restrictions on functional programs implies that the defining equations form an orthogonal term rewrite sytem
  - Rosen proved that orthogonal term rewrite sytems are confluent
  - confluence implies that each term has at most one normal form
  - full proof of Rosen given in term rewriting lecture, we only sketch a weaker property on the next slides, namely local confluence: whenever $s \hookrightarrow t$ and $s \hookrightarrow u$ then there exists some v such that $t \hookrightarrow^* v$ and $u \hookrightarrow^* v$
  - local confluence in combination with termination also implies confluence

Proof of Local Confluence: Two Root Steps

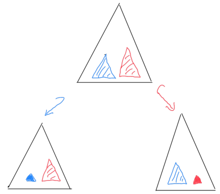
- consider the situation in the diagram where two root steps with equations $\ell_1 = r_1$ and $\ell_2 = r_2$ are applied



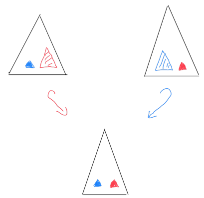
- because of pattern disjointness: $(\ell_1 = r_1) = (\ell_2 = r_2)$
- uniqueness of matching: $\sigma_1(x) = \sigma_2(x)$ for all $x \in \mathcal{V}ars(\ell_{1/2})$
- variable condition of programs: $\sigma_1(x) = \sigma_2(x)$ for all $x \in \mathcal{V}ars(r_{1/2})$
- hence $r_1\sigma_1 = r_2\sigma_2$

Proof of Local Confluence: Independent Steps

- consider the situation in the diagram where two steps at independent positions are applied

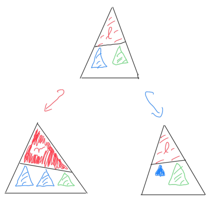


- just do the steps in reverse order

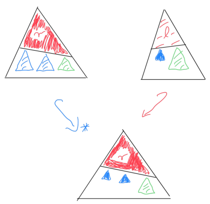


Proof of Local Confluence: Root- and Substitution-Step

- consider the situation in the diagram where a root step overlaps with a step done in the substitution



- just do the steps in reverse order (perhaps multiple times)



Graphical Local Confluence Proof

- the diagrams in the three previous slides describe all situations where one term can be evaluated in two different ways (within one step)
- in all cases the diagrams could be joined
- overall: intuitive graphical proof of local confluence
- often hard task: transform such an intuitive proof into a formal, purely textual proof, using induction, case-analysis, etc.

Semantics for Functional Programs in the Standard Model

- we are now ready to complete the semantics for functional programs
- we call a functional program well-defined, if
  - it is pattern disjoint,
  - it is pattern complete, and
  - $\hookrightarrow$ is terminating
- for well-defined programs, we define for each $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathcal{D}$

$$f^\mathcal{M} : \mathcal{T}(\mathcal{C})_{\tau_1} \times \ldots \times \mathcal{T}(\mathcal{C})_{\tau_n} \to \mathcal{T}(\mathcal{C})_\tau$$
$$f^\mathcal{M}(t_1, \ldots, t_n) = s$$

  where s is the unique normal form of $f(t_1, \ldots, t_n)$, i.e., $f(t_1, \ldots, t_n) \hookrightarrow^! s$
- remarks:
  - a normal form exists, since $\hookrightarrow$ is terminating
  - s is unique because of pattern disjointness
  - $s \in \mathcal{T}(\mathcal{C})_\tau$ because of pattern completeness, and type- and groundness-preservation

Summary: Standard Model

- standard model
  - universes: $\mathcal{T}(\mathcal{C})_\tau$
  - constructors: $c^\mathcal{M}(t_1, \ldots, t_n) = c(t_1, \ldots, t_n)$
  - defined symbols: $f^\mathcal{M}(t_1, \ldots, t_n)$ is normal form of $f(t_1, \ldots, t_n)$ w.r.t. $\hookrightarrow$
- if functional program is well-defined
  - pattern disjoint,
  - pattern complete, and
  - $\hookrightarrow$ is terminating

  then standard model is well-defined
- upcoming
  - what about functional programs that are not well-defined?
  - comparison to real functional programming languages
  - treatment in real proof assistants

Without Pattern Disjointness

- consider Haskell program

```
conj :: Bool -> Bool -> Bool
conj True True = True    -- (1)
conj x    y    = False   -- (2)
```

- obviously not pattern disjoint
- however, Haskell still has unique results, since equations are ordered
  - an equation is only applicable
    if all previous equations are not applicable
  - so, `conj True True` can only be evaluated to `True`
- ordering of equations can be resolved by instantiation equations via complementary patterns
- equivalent equations (in Haskell) which do not rely upon order of equations

```
conj :: Bool -> Bool -> Bool
conj True  True  = True   -- (1)
conj False y     = False  -- (2) with x / False
conj True  False = False  -- (2) with x / True, y / False
```


Without Pattern Disjointness – Continued

- pattern disjointness is sufficient criterion to ensure confluence
- overlaps can be allowed, if they do not cause conflicts
- example:

```
conj :: Bool -> Bool -> Bool
conj True  True  = True
conj False y     = False   -- (1)
conj x     False = False   -- (2)
```

  the only overlap is `conj False False`; it is harmless since the term evaluates to the same result using both (1) and (2)
- translating ordered equations into pattern disjoint equations or equations which only have harmless overlaps can be done automatically
  - usually, there are several possibilities
  - finding the smallest set of equations is hard
  - automatically done in proof-assistants such as Isabelle;
    e.g., overlapping `conj` from previous slide is translated into above one
- consequence: pattern disjointness is no real restriction

Without Pattern Completeness

- pattern completeness is naturally missing in several functions
- examples from Haskell libraries
```
head :: [a] -> a
head (x : xs) = x
```
- resolving pattern incompleteness is possible in the standard model
  - determine missing patterns
  - add for these missing cases equations that assign some element of the universe

$$\mathsf{head}(\mathsf{Cons}(x, xs)) = x \qquad \text{equation as before}$$
$$\mathsf{head}(\mathsf{Nil}) = \text{some element of } \mathcal{T}(\mathcal{C})_{\mathsf{Nat}} \qquad \text{new equation}$$

  - in this way, head becomes pattern complete and $\mathsf{head}^{\mathcal{M}}$ is total
  - "some element" really is an element of $\mathcal{T}(\mathcal{C})_{\mathsf{Nat}}$, and not a special error value like $\bot$
  - the added equation with "some element" is usually not revealed to the user, so he or she cannot infer what number $\mathsf{head}(\mathsf{Nil})$ actually is
- consequence: pattern completeness is no real restriction

Without Termination

- definition of standard model just doesn't work properly in case of non-termination
- one possibility: use Scott's domain theory where among others, explicit $\bot$-elements are added to universe
- examples
  - $\mathcal{A}_{\mathsf{Nat}} = \{\bot, \mathsf{Zero}, \mathsf{Succ}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero})), \dots, \mathsf{Succ}^{\infty}\}$
  - $\mathcal{A}_{\mathsf{List}} = \{\bot, \mathsf{Nil}, \mathsf{Cons}(\mathsf{Zero}, \mathsf{Nil}), \mathsf{Cons}(\bot, \mathsf{Nil}), \mathsf{Cons}(\bot, \bot), \dots\}$
- then semantics can be given to non-terminating computations
  - $\mathsf{inf} = \mathsf{Succ}(\mathsf{inf})$ leads to $\mathsf{inf}^{\mathcal{M}} = \mathsf{Succ}^{\infty}$
  - $\mathsf{undef} = \mathsf{undef}$ leads to $\mathsf{undef}^{\mathcal{M}} = \bot$
- problem: certain equalities don't hold w.r.t. domain theory semantics
  - assume usual definition of program for minus, then $\forall x.\, \mathsf{minus}(x, x) = \mathsf{Zero}$ is not true, consider $x = \mathsf{inf}$ or $x = \mathsf{undef}$
- since reasoning in domain theory is more complex, in this course we restrict to terminating functional programs
- even large proof assistants like Isabelle and Coq usually restrict to terminating functions for that reason

Inference Rules for the Standard Model

Plan

- from now until the end of these slides consider only well-defined functional programs, so that standard model is well-defined
- aim
  - derive theorems and inference rules which are valid in the standard model
  - these can be used to formally reason about functional programs as on slide 1/18 where associativity of append was proven
- examples
  - reasoning about constructors
    - $\forall x, y.\, \mathsf{Succ}(x) =_{\mathsf{Nat}} \mathsf{Succ}(y) \longleftrightarrow x =_{\mathsf{Nat}} y$
    - $\forall x.\, \neg\, \mathsf{Succ}(x) =_{\mathsf{Nat}} \mathsf{Zero}$
  - getting defining equations of functional programs as theorems
    - $\forall x, xs, ys.\, \mathsf{append}(\mathsf{Cons}(x, xs), ys) =_{\mathsf{List}} \mathsf{Cons}(x, \mathsf{append}(xs, ys))$
  - induction schemes
    - $\dfrac{\varphi(\mathsf{Zero}) \quad \forall x.\, \varphi(x) \longrightarrow \varphi(\mathsf{Succ}(x))}{\forall x.\, \varphi(x)}$

Notation – The Normal Form

- when speaking about $\hookrightarrow$, we always consider some fixed well-defined functional program
- since every term has a unique normal form w.r.t. $\hookrightarrow$, we can define a function $\downarrow : \mathcal{T}(\Sigma,\mathcal{V})_\tau \to \mathcal{T}(\Sigma,\mathcal{V})_\tau$ which returns this normal form and write it in postfix notation:

$$t\downarrow := \text{the unique normal of } t \text{ w.r.t. } \hookrightarrow$$

- using $\downarrow$, the meaning of symbols in the standard model can concisely be written as

$$F^{\mathcal{M}}(t_1,\dots,t_n) = F(t_1,\dots,t_n)\downarrow$$

- proof
  - if $F \in \mathcal{C}$, then $F^{\mathcal{M}}(t_1,\dots,t_n) \stackrel{def}{=} F(t_1,\dots,t_n) = F(t_1,\dots,t_n)\downarrow$
  - if $F \in \mathcal{D}$, then $F^{\mathcal{M}}(t_1,\dots,t_n) \stackrel{def}{=} F(t_1,\dots,t_n)\downarrow$

The Substitution Lemma

- there are two possibilities to plug in objects into variables
  - as environment: $\alpha : \mathcal{V}_\tau \to \mathcal{A}_\tau$
    result of $[\![t]\!]_\alpha$ is an element of $\mathcal{A}_\tau$
  - as substitution: $\sigma : \mathcal{V}_\tau \to \mathcal{T}(\Sigma,\mathcal{V})_\tau$
    result of $t\sigma$ is an element of $\mathcal{T}(\Sigma,\mathcal{V})_\tau$
- substitution lemma: substitutions can be moved into environment:

$$[\![t\sigma]\!]_\alpha = [\![t]\!]_\beta$$

  where $\beta(x) := [\![\sigma(x)]\!]_\alpha$
- proof by structural induction on t
  - $[\![x\sigma]\!]_\alpha = [\![\sigma(x)]\!]_\alpha = \beta(x) = [\![x]\!]_\beta$
  - $$\begin{aligned}[\![F(t_1,\dots,t_n)\sigma]\!]_\alpha &= [\![F(t_1\sigma,\dots,t_n\sigma)]\!]_\alpha \\ &= F^{\mathcal{M}}([\![t_1\sigma]\!]_\alpha,\dots,[\![t_n\sigma]\!]_\alpha) \\ &\stackrel{IH}{=} F^{\mathcal{M}}([\![t_1]\!]_\beta,\dots,[\![t_n]\!]_\beta) \\ &= [\![F(t_1,\dots,t_n)]\!]_\beta\end{aligned}$$

Reverse Substitution Lemma in the Standard Model

- the substitution lemma holds independently of the model
- in case of the standard model, we have the special condition that $\mathcal{A}_\tau = \mathcal{T}(\mathcal{C})_\tau$, so
  - the universes consist of terms
  - hence, each environment $\alpha : \mathcal{V}_\tau \to \mathcal{T}(\mathcal{C})_\tau$ is a special kind of substitution (constructor ground substitution)
- consequence: possibility to encode environment as substitution
- reverse substitution lemma:

$$[\![t]\!]_\alpha = t\alpha\downarrow$$

- proof by structural induction on t
  - $[\![x]\!]_\alpha = \alpha(x) \stackrel{(*)}{=} \alpha(x)\downarrow = x\alpha\downarrow$ where $(*)$ holds, since $\alpha(x) \in \mathcal{T}(\mathcal{C})$
  - $$\begin{aligned}[\![F(t_1,\dots,t_n)]\!]_\alpha &= F^{\mathcal{M}}([\![t_1]\!]_\alpha,\dots,[\![t_n]\!]_\alpha) \\ &\stackrel{IH}{=} F^{\mathcal{M}}(t_1\alpha\downarrow,\dots,t_n\alpha\downarrow) = F(t_1\alpha\downarrow,\dots,t_n\alpha\downarrow)\downarrow \\ &\stackrel{(confl.)}{=} F(t_1\alpha,\dots,t_n\alpha)\downarrow = F(t_1,\dots,t_n)\alpha\downarrow\end{aligned}$$

Defining Equations are Theorems in Standard Model

- notation: $\vec{\forall}\varphi$ means that universal quantification ranges over all free variables that occur in φ
- example: if φ is $\mathsf{append}(\mathsf{Cons}(x,xs),ys) =_{\mathsf{List}} \mathsf{Cons}(x,\mathsf{append}(xs,ys))$ then $\vec{\forall}\varphi$ is

$$\forall x,xs,ys.\, \mathsf{append}(\mathsf{Cons}(x,xs),ys) =_{\mathsf{List}} \mathsf{Cons}(x,\mathsf{append}(xs,ys))$$

- theorem: if $\ell = r$ is defining equation of program (of type τ), then

$$\mathcal{M} \models \vec{\forall}\,\ell =_\tau r$$

- consequence: conversion of well-defined functional programs into equations is now possible, cf. previous problem on slide 1/20
- proof of theorem
  - by definition of $\models$ and $=_\tau^{\mathcal{M}}$ we have to show $[\![\ell]\!]_\alpha = [\![r]\!]_\alpha$ for all α
  - via reverse substitution lemma this is equivalent to $\ell\alpha\downarrow = r\alpha\downarrow$
  - easily follows from confluence, since $\ell\alpha \hookrightarrow r\alpha$

Axiomatic Reasoning

- previous slide already provides us with some theorems that are satisfied in standard model
- axiomatic reasoning:
  take those theorems as axioms to show property φ
- added axioms are theorems of standard model, so they are consistent
- example $AX = \{\vec{\forall}\,\ell =_\tau r \mid \ell = r \text{ is def. eqn.}\}$
- show $AX \models \varphi$ using first-order reasoning in order to prove $\mathcal{M} \models \varphi$
  (and forget standard model $\mathcal{M}$ during the reasoning!)
- question: is it possible to prove every property φ in this way for which $\mathcal{M} \models \varphi$ holds?
- answer for above example is "no"
  - reason: there are models different than the standard model in which all axioms of AX are satisfied, but where φ does not hold!
  - example on next slide

Axiomatic Reasoning – Problematic Model

- consider addition program, then example AX consists of two axioms

$$\forall y.\, \mathsf{plus}(\mathsf{Zero}, y) =_{\mathsf{Nat}} y$$
$$\forall x, y.\, \mathsf{plus}(\mathsf{Succ}(x), y) =_{\mathsf{Nat}} \mathsf{Succ}(\mathsf{plus}(x, y))$$

- we want to prove associativity of plus, so let φ be

$$\forall x, y, z.\, \mathsf{plus}(\mathsf{plus}(x, y), z) =_{\mathsf{Nat}} \mathsf{plus}(x, \mathsf{plus}(y, z))$$

- consider the following model $\mathcal{M}'$
  - $\mathcal{A}_{\mathsf{Nat}} = \mathbb{N} \cup \{x + \frac{1}{2} \mid x \in \mathbb{Z}\} = \{\dots, -1\frac{1}{2}, -\frac{1}{2}, 0, \frac{1}{2}, 1, 1\frac{1}{2}, 2, 2\frac{1}{2}, \dots\}$
  - $\mathsf{Zero}^{\mathcal{M}'} = 0$
  - $\mathsf{Succ}^{\mathcal{M}'}(n) = n + 1$
  - $\mathsf{plus}^{\mathcal{M}'}(n, m) = \begin{cases} n + m, & \text{if } n \in \mathbb{N} \text{ or } m \in \mathbb{N} \\ n - m + \frac{1}{2}, & \text{otherwise} \end{cases}$
  - ${=_{\mathsf{Nat}}}^{\mathcal{M}} = \{(n, n) \mid n \in \mathcal{A}_{\mathsf{Nat}}\}$
  - $\mathcal{M}' \models \bigwedge AX$, but $\mathcal{M}' \not\models \varphi$: consider $\alpha(x) = \frac{19}{2}, \alpha(y) = \frac{9}{2}, \alpha(z) = \frac{7}{2}$
  - problem: values in α do not correspond to constructor ground terms

Gödel's Incompleteness Theorem

- taking AX as set of defining equations does not suffice to deduce all valid theorems of standard model
- obvious approach: add more theorems to axioms AX
  (theorems about $=_\tau$, induction rules, ...)
- question: is it then possible to deduce all valid theorems of standard model?
- negative answer by Gödel's First Incompleteness Theorem
- theorem: consider a well-defined functional program that includes addition and multiplication of natural numbers;
  let AX be a decidable set of valid theorems in the standard model;
  then there is a formula φ such that $\mathcal{M} \models \varphi$, but $AX \not\models \varphi$
- note: adding φ to AX does not fix the problem, since then there is another formula φ' so that $AX \cup \{\varphi\} \not\models \varphi'$
- consequence: "proving φ via $AX \models \varphi$" is sound, but never complete
- upcoming: add more axioms than just defining equations,
  so that still several proofs are possible

Axioms about Equality

- we define decomposition theorems and disjointness theorems in the form of logical equivalences
- for each $c : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathcal{C}$ we define its
  decomposition theorem as

$$\vec{\forall}\, c(x_1, \ldots, x_n) =_\tau c(y_1, \ldots, y_n) \longleftrightarrow x_1 =_{\tau_1} y_1 \wedge \ldots \wedge x_n =_{\tau_n} y_n$$

  and for all $d : \tau_1' \times \ldots \times \tau_k' \to \tau \in \mathcal{C}$ with $c \neq d$ we define the disjointness theorem as

$$\vec{\forall}\, c(x_1, \ldots, x_n) =_\tau d(y_1, \ldots, y_k) \longleftrightarrow \mathsf{false}$$

- proof of validity of decomposition theorem:

$$\begin{aligned} & \mathcal{M} \models_\alpha c(x_1, \ldots, x_n) =_\tau c(y_1, \ldots, y_n) \\ \text{iff } & c(\alpha(x_1), \ldots, \alpha(x_n)) = c(\alpha(y_1), \ldots, \alpha(y_n)) \\ \text{iff } & \alpha(x_1) = \alpha(y_1) \text{ and } \ldots \text{ and } \alpha(x_n) = \alpha(y_n) \\ \text{iff } & \mathcal{M} \models_\alpha x_1 =_{\tau_1} y_1 \text{ and } \ldots \text{ and } \mathcal{M} \models_\alpha x_n =_{\tau_n} y_n \\ \text{iff } & \mathcal{M} \models_\alpha x_1 =_{\tau_1} y_1 \wedge \ldots \wedge x_n =_{\tau_n} y_n \end{aligned}$$

Axioms about Equality – Example

- for the datatypes of natural numbers and lists we get the following axioms

$$\begin{aligned} \mathsf{Zero} =_{\mathsf{Nat}} \mathsf{Zero} &\longleftrightarrow \mathsf{true} \\ \forall x, y.\, \mathsf{Succ}(x) =_{\mathsf{Nat}} \mathsf{Succ}(y) &\longleftrightarrow x =_{\mathsf{Nat}} y \\ \mathsf{Nil} =_{\mathsf{List}} \mathsf{Nil} &\longleftrightarrow \mathsf{true} \\ \forall x, xs, y, ys.\, \mathsf{Cons}(x, xs) =_{\mathsf{List}} \mathsf{Cons}(y, ys) &\longleftrightarrow x =_{\mathsf{Nat}} y \wedge xs =_{\mathsf{List}} ys \end{aligned}$$

$$\begin{aligned} \forall y.\, \mathsf{Zero} =_{\mathsf{Nat}} \mathsf{Succ}(y) &\longleftrightarrow \mathsf{false} \\ \forall x.\, \mathsf{Succ}(x) =_{\mathsf{Nat}} \mathsf{Zero} &\longleftrightarrow \mathsf{false} \\ \forall y, ys.\, \mathsf{Nil} =_{\mathsf{List}} \mathsf{Cons}(y, ys) &\longleftrightarrow \mathsf{false} \\ \forall x, xs.\, \mathsf{Cons}(x, xs) =_{\mathsf{List}} \mathsf{Nil} &\longleftrightarrow \mathsf{false} \end{aligned}$$

Induction Theorems

- current axioms are not even strong enough to prove simple theorems, e.g., $\forall x.\ \mathsf{plus}(x, \mathsf{Zero}) =_{\mathsf{Nat}} x$
- problem: proofs by induction are not yet covered in axioms
- since the principle of induction cannot be defined in general in a single first-order formula, we will add infinitely many induction theorems to the set of axioms, one for each property
- not a problem, since set of axioms stays decidable, i.e., one can see whether some tentative formula is an element of the axiom set or not
- example: induction over natural numbers
  - formula below is general, but not first-order as it quantifies over φ

$$\forall \varphi(x : \mathsf{Nat}).\, \varphi(\mathsf{Zero}) \longrightarrow (\forall x.\, \varphi(x) \longrightarrow \varphi(\mathsf{Succ}(x))) \longrightarrow \forall x.\, \varphi(x)$$

  - quantification can be done on meta-level instead: let φ be an arbitrary formula with a free variable of type Nat; then

$$\varphi(\mathsf{Zero}) \longrightarrow (\forall x.\, \varphi(x) \longrightarrow \varphi(\mathsf{Succ}(x))) \longrightarrow \forall x.\, \varphi(x)$$

    is a valid theorem; quantifying over φ results in induction scheme

Induction Theorems – Example Instances

- induction scheme

$$\varphi(\mathsf{Zero}) \longrightarrow (\forall x.\, \varphi(x) \longrightarrow \varphi(\mathsf{Succ}(x))) \longrightarrow \forall x.\, \varphi(x)$$

- example: right-neutral element: $\varphi(x) := \mathsf{plus}(x, \mathsf{Zero}) =_{\mathsf{Nat}} x$

$$\begin{aligned} &\mathsf{plus}(\mathsf{Zero}, \mathsf{Zero}) =_{\mathsf{Nat}} \mathsf{Zero} \\ &\longrightarrow (\forall x.\, \mathsf{plus}(x, \mathsf{Zero}) =_{\mathsf{Nat}} x \longrightarrow \mathsf{plus}(\mathsf{Succ}(x), \mathsf{Zero}) =_{\mathsf{Nat}} \mathsf{Succ}(x)) \\ &\longrightarrow \forall x.\, \mathsf{plus}(x, \mathsf{Zero}) =_{\mathsf{Nat}} x \end{aligned}$$

- example with quantifiers and free variables: $\varphi(x) := \forall y.\, \mathsf{plus}(\mathsf{plus}(x, y), z) =_{\mathsf{Nat}} \mathsf{plus}(x, \mathsf{plus}(y, z))$

$$\begin{aligned} &\forall y.\, \mathsf{plus}(\mathsf{plus}(\mathsf{Zero}, y), z) =_{\mathsf{Nat}} \mathsf{plus}(\mathsf{Zero}, \mathsf{plus}(y, z)) \\ &\longrightarrow (\forall x.\, (\forall y.\, \mathsf{plus}(\mathsf{plus}(x, y), z) =_{\mathsf{Nat}} \mathsf{plus}(x, \mathsf{plus}(y, z))) \\ &\qquad \longrightarrow (\forall y.\, \mathsf{plus}(\mathsf{plus}(\mathsf{Succ}(x), y), z) =_{\mathsf{Nat}} \mathsf{plus}(\mathsf{Succ}(x), \mathsf{plus}(y, z)))) \\ &\longrightarrow \forall x.\, \forall y.\, \mathsf{plus}(\mathsf{plus}(x, y), z) =_{\mathsf{Nat}} \mathsf{plus}(x, \mathsf{plus}(y, z)) \end{aligned}$$

Preparing Induction Theorems – Substitutions in Formulas

- current situation
  - substitutions are functions of type $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$
  - lifted to functions of type $\mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$, cf. slide 22
  - substitution of variables of formulas is not yet defined, but is required for induction formulas, cf. notation $\varphi(x) \longrightarrow \varphi(\mathsf{Succ}(x))$ on previous slide
- formal definition of applying a substitution σ to formulas
  - $\mathsf{true}\,\sigma = \mathsf{true}$
  - $(\neg\varphi)\sigma = \neg(\varphi\sigma)$
  - $(\varphi \wedge \psi)\sigma = \varphi\sigma \wedge \psi\sigma$
  - $P(t_1, \ldots, t_n)\sigma = P(t_1\sigma, \ldots, t_n\sigma)$
  - $(\forall x.\, \varphi)\sigma = \forall x.\, (\varphi\sigma)$ if x does not occur in σ, i.e., $\sigma(x) = x$ and $x \notin \mathcal{V}ars(\sigma(y))$ for all $y \neq x$
  - $(\forall x.\, \varphi)\sigma = (\forall y.\, \varphi[x/y])\sigma$ if x occurs in σ where
    - y is a fresh variable, i.e., $\sigma(y) = y$, $y \notin \mathcal{V}ars(\sigma(z))$ for all $z \neq y$, and y is not a free variable of φ
    - $[x/y]$ is the substitution which just replaces x by y
    - effect is α-renaming: just rename universally quantified variable before substitution to avoid variable capture

Examples

- substitution of formulas
  - $(\forall x. \varphi)\sigma = \forall x. (\varphi\sigma)$ if x does not occur in σ
  - $(\forall x. \varphi)\sigma = (\forall y. \varphi[x/y])\sigma$ if x occurs in σ where y is fresh
- example substitution applications
  - $\varphi := \forall x. \neg\, x =_{\mathsf{Nat}} y$
  - $\varphi[y/\mathsf{Zero}] = \forall x. \neg\, x =_{\mathsf{Nat}} \mathsf{Zero}$ no renaming required
  - $\varphi[y/\mathsf{Succ}(z)] = \forall x. \neg\, x =_{\mathsf{Nat}} \mathsf{Succ}(z)$ no renaming required
  - $\varphi[y/\mathsf{Succ}(x)] = \forall z. \neg\, z =_{\mathsf{Nat}} \mathsf{Succ}(x)$ renaming $[x/z]$ required
    without renaming meaning will change: $\forall x. \neg\, x =_{\mathsf{Nat}} \mathsf{Succ}(x)$
  - $\varphi[x/\mathsf{Succ}(y)] = \forall z. \neg\, z =_{\mathsf{Nat}} y$ renaming $[x/z]$ required
    without renaming meaning will change: $\forall x. \neg \mathsf{Succ}(y) =_{\mathsf{Nat}} y$
- example theorems involving substitutions

$$\varphi[x/\mathsf{Zero}] \longrightarrow (\forall y. \varphi[x/y] \longrightarrow \varphi[x/\mathsf{Succ}(y)]) \longrightarrow \forall x. \varphi$$

Substitution Lemma for Formulas

- example induction formula

$$\varphi[x/\mathsf{Zero}] \longrightarrow (\forall y. \varphi[x/y] \longrightarrow \varphi[x/\mathsf{Succ}(y)]) \longrightarrow \forall x. \varphi$$

- proving validity of this formula (in standard model) requires another substitution lemma about substitutions in formulas
- lemma: $\mathcal{M} \models_\alpha \varphi\sigma$ iff $\mathcal{M} \models_\beta \varphi$ where $\beta(x) := [\![\sigma(x)]\!]_\alpha$
- proof by structural induction on φ for arbitrary α and σ
  - $\mathcal{M} \models_\alpha P(t_1, \dots, t_n)\sigma$
    iff $\mathcal{M} \models_\alpha P(t_1\sigma, \dots, t_n\sigma)$
    iff $([\![t_1\sigma]\!]_\alpha, \dots, [\![t_n\sigma]\!]_\alpha) \in P^{\mathcal{M}}$
    iff $([\![t_1]\!]_\beta, \dots, [\![t_n]\!]_\beta) \in P^{\mathcal{M}}$
    iff $\mathcal{M} \models_\beta P(t_1, \dots, t_n)$
    where we use the substitution lemma of slide 54 to conclude $[\![t_i\sigma]\!]_\alpha = [\![t_i]\!]\beta$
  - $\mathcal{M} \models_\alpha (\neg\varphi)\sigma$ iff $\mathcal{M} \models_\alpha \neg(\varphi\sigma)$ iff $\mathcal{M} \not\models_\alpha \varphi\sigma$
    iff $\mathcal{M} \not\models_\beta \varphi$ (by IH) iff $\mathcal{M} \models_\beta \neg\varphi$
  - cases "true" and conjunction are proved in same way as negation

Substitution Lemma for Formulas – Proof Continued

- lemma: $\mathcal{M} \models_\alpha \varphi\sigma$ iff $\mathcal{M} \models_\beta \varphi$ where $\beta(x) := [\![\sigma(x)]\!]_\alpha$
- proof by structural induction on φ for arbitrary α and σ
  - for quantification we here only consider the more complex case where renaming is required
  - $\mathcal{M} \models_\alpha (\forall x. \varphi)\sigma$
    iff $\mathcal{M} \models_\alpha (\forall y. \varphi[x/y])\sigma$ for fresh y
    iff $\mathcal{M} \models_\alpha \forall y. (\varphi[x/y]\sigma)$
    iff $\mathcal{M} \models_{\alpha[y:=a]} \varphi[x/y]\sigma$ for all $a \in \mathcal{A}$
    iff $\mathcal{M} \models_{\beta'} \varphi$ for all $a \in \mathcal{A}$ where $\beta'(z) := [\![([x/y]\sigma)(z)]\!]_{\alpha[y:=a]}$ (by IH)
    iff $\mathcal{M} \models_{\beta[x:=a]} \varphi$ for all $a \in \mathcal{A}$ only non-automatic step
    iff $\mathcal{M} \models_\beta \forall x. \varphi$
  - equivalence of β' and $\beta[x := a]$ on variables of φ
    - $\beta'(x) = [\![([x/y]\sigma)(x)]\!]_{\alpha[y:=a]} = [\![\sigma(y)]\!]_{\alpha[y:=a]} = [\![y]\!]_{\alpha[y:=a]} = a$ and $\beta[x := a](x) = a$
    - z is variable of φ, $z \neq x$:
      by freshness condition conclude $z \neq y$ and $y \notin \mathcal{V}ars(\sigma(z))$; hence
      $\beta'(z) = [\![([x/y]\sigma)(z)]\!]_{\alpha[y:=a]} = [\![\sigma(z)]\!]_{\alpha[y:=a]} = [\![\sigma(z)]\!]_\alpha$ and
      $\beta[x := a](z) = \beta(z) = [\![\sigma(z)]\!]_\alpha$

Substitution Lemma in Standard Model

- substitution lemma: $\mathcal{M} \models_\alpha \varphi\sigma$ iff $\mathcal{M} \models_\beta \varphi$ where $\beta(x) := [\![\sigma(x)]\!]_\alpha$
- lemma is valid for all models
- in standard model, substitution lemma permits to characterize universal quantification by substitutions, similar to reverse substitution lemma on slide 55
- lemma: let $x : \tau \in \mathcal{V}$, let $\mathcal{M}$ be the standard model
  1. $\mathcal{M} \models_{\alpha[x:=t]} \varphi$ iff $\mathcal{M} \models_\alpha \varphi[x/t]$
  2. $\mathcal{M} \models_\alpha \forall x. \varphi$ iff $\mathcal{M} \models_\alpha \varphi[x/t]$ for all $t \in \mathcal{T}(\mathcal{C})_\tau$
- proof
  1. first note that the usage of $\alpha[x := t]$ implies $t \in \mathcal{A}_\tau = \mathcal{T}(\mathcal{C})_\tau$;
     by the substitution lemma obtain
     $\mathcal{M} \models_\alpha \varphi[x/t]$
     iff $\mathcal{M} \models_\beta \varphi$ for $\beta(z) = [\![[x/t](z)]\!]_\alpha = \alpha[x := [\![t]\!]_\alpha](z)$
     iff $\mathcal{M} \models_{\alpha[x:=t]} \varphi$ ($[\![t]\!]_\alpha = t$, since $t \in \mathcal{T}(\mathcal{C})$)
  2. immediate by part 1 of lemma

Substitution Lemma and Induction Formulas

- substitution lemma (SL) is crucial result to lift structural induction rule of universe $\mathcal{T}(\mathcal{C})_\tau$ to a structural induction formula
- example: structural induction formula ψ for lists with fresh x, xs
$$\psi := \underbrace{\varphi[ys/\mathsf{Nil}]}_{1} \longrightarrow (\underbrace{\forall x, xs.\, \varphi[ys/xs] \longrightarrow \varphi[ys/\mathsf{Cons}(x, xs)]}_{2}) \longrightarrow \forall ys.\, \varphi$$
- proof of $\mathcal{M} \models_\alpha \psi$:
assume premises 1 ($\mathcal{M} \models_\alpha \varphi[ys/\mathsf{Nil}]$) and 2 and show $\mathcal{M} \models_\alpha \forall ys.\, \varphi$:
by SL the latter is equivalent to "$\mathcal{M} \models_\alpha \varphi[ys/\ell]$ for all $\ell \in \mathcal{T}(\mathcal{C})_{\mathsf{List}}$";
prove this statement by structural induction on lists
  - Nil: showing $\mathcal{M} \models_\alpha \varphi[ys/\mathsf{Nil}]$ is easy: it is exactly premise 1
  - $\mathsf{Cons}(n, \ell)$: use SL on premise 2 to conclude
$$\mathcal{M} \models_\alpha (\varphi[ys/xs] \longrightarrow \varphi[ys/\mathsf{Cons}(x, xs)])[x/n, xs/\ell]$$
hence
$$\mathcal{M} \models_\alpha \varphi[ys/\ell] \longrightarrow \varphi[ys/\mathsf{Cons}(n, \ell)]$$
and with IH $\mathcal{M} \models_\alpha \varphi[ys/\ell]$ conclude $\mathcal{M} \models_\alpha \varphi[ys/\mathsf{Cons}(n, \ell)]$

Freshness of Variables

- example: structural induction formula for lists with fresh x, xs
$$\varphi[ys/\mathsf{Nil}] \longrightarrow (\forall x, xs.\, \varphi[ys/xs] \longrightarrow \varphi[ys/\mathsf{Cons}(x, xs)]) \longrightarrow \forall ys.\, \varphi$$
- why freshness required? isn't name of quantified variables irrelevant?
- problem: substitution is applied below quantifier!
- example: let us drop freshness condition and "prove" non-theorem
$$\mathcal{M} \models \forall x, xs, ys.\, ys =_{\mathsf{List}} \mathsf{Nil} \vee ys =_{\mathsf{List}} \mathsf{Cons}(x, xs)$$
  - by semantics of $\forall x, xs. \ldots$ it suffices to prove
$$\mathcal{M} \models_\alpha \forall ys.\, \underbrace{ys =_{\mathsf{List}} \mathsf{Nil} \vee ys =_{\mathsf{List}} \mathsf{Cons}(x, xs)}_{\varphi}$$
  - apply above induction formula and obtain two subgoals $\mathcal{M} \models_\alpha \ldots$ for
    - $\varphi[ys/\mathsf{Nil}]$ which is $\mathsf{Nil} =_{\mathsf{List}} \mathsf{Nil} \vee \mathsf{Nil} =_{\mathsf{List}} \mathsf{Cons}(x, xs)$
    - $\forall x, xs.\, \varphi[ys/xs] \longrightarrow \varphi[ys/\mathsf{Cons}(x, xs)]$ which is $\forall x, xs.\, \ldots \longrightarrow \mathsf{Cons}(x, xs) =_{\mathsf{List}} \mathsf{Nil} \vee \mathsf{Cons}(x, xs) =_{\mathsf{List}} \mathsf{Cons}(x, xs)$
- solution: rename variables in induction formula whenever required

Structural Induction Formula

- finally definition of induction formula for data structures is possible
- consider
$$\begin{aligned} \mathsf{data}\ \tau = {} & c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau \\ \mid {} & \ldots \\ \mid {} & c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau \end{aligned}$$
- let $x \in \mathcal{V}_\tau$, let φ be a formula, let variables $x_1, x_2, \ldots$ be fresh w.r.t. φ
- for each c_i define
$$\varphi_i := \forall x_1, \ldots, x_{m_i}.\, \underbrace{\left(\bigwedge_{j, \tau_{i,j} = \tau} \varphi[x/x_j] \right)}_{\text{IH for recursive arguments}} \longrightarrow \varphi[x/c_i(x_1, \ldots, x_{m_i})]$$
- the induction formula is $\vec{\forall}\, (\varphi_1 \longrightarrow \ldots \longrightarrow \varphi_n \longrightarrow \forall x.\, \varphi)$
- theorem: $\mathcal{M} \models \vec{\forall}\, (\varphi_1 \longrightarrow \ldots \longrightarrow \varphi_n \longrightarrow \forall x.\, \varphi)$

Proof of Structural Induction Formula

- to prove: $\mathcal{M} \models \vec{\forall}\, (\varphi_1 \longrightarrow \ldots \longrightarrow \varphi_n \longrightarrow \forall x.\, \varphi)$
- $\forall$-intro: $\mathcal{M} \models_\alpha (\varphi_1 \longrightarrow \ldots \longrightarrow \varphi_n \longrightarrow \forall x.\, \varphi)$ for arbitrary α
- $\longrightarrow$-intro: assume $\mathcal{M} \models_\alpha \varphi_i$ for all i and show $\mathcal{M} \models_\alpha \forall x.\, \varphi$
- $\forall$-intro via SL: show $\mathcal{M} \models_\alpha \varphi[x/t]$ for all $t \in \mathcal{T}(\mathcal{C})_\tau$
- prove this by structural induction on t w.r.t. induction rule of $\mathcal{T}(\mathcal{C})_\tau$ (for precisely this α, not for arbitrary α)
- induction step for each constructor $c_i : \tau_{i,1} \times \ldots \times \tau_{i,m_i} \to \tau$
  - aim: $\mathcal{M} \models_\alpha \varphi[x/c_i(t_1, \ldots, t_{m_i})]$ IH: $\mathcal{M} \models_\alpha \varphi[x/t_j]$ for all j such that $\tau_{i,j} = \tau$
  - use assumption $\mathcal{M} \models_\alpha \varphi_i$, i.e., (here important: same α)
$$\mathcal{M} \models_\alpha \forall x_1, \ldots, x_{m_i}.\, (\bigwedge_{j, \tau_{i,j} = \tau} \varphi[x/x_j]) \longrightarrow \varphi[x/c_i(x_1, \ldots, x_{m_i})]$$
  - use SL as $\forall$-elimination with substitution $[x_1/t_1, \ldots, x_{m_i}/t_{m_i}]$, obtain
$$\mathcal{M} \models_\alpha (\bigwedge_{j, \tau_{i,j} = \tau} \varphi[x/t_j]) \longrightarrow \varphi[x/c_i(t_1, \ldots, t_{m_i})]$$
  - combination with IH yields desired $\mathcal{M} \models_\alpha \varphi[x/c_i(t_1, \ldots, t_{m_i})]$

Summary: Axiomatic Proofs of Functional Programs

- given a well-defined functional program, define a set of axioms AX consisting of
  - equations of defined symbols (slide 56)
  - axioms about equality of constructors (slide 60)
  - structural induction formulas (slide 71)
- instead of proving $\mathcal{M} \models \varphi$ deduce $AX \models \varphi$
- fact: standard model is ignored in previous step
- question: why all these efforts and not just state AX?
- reason:

  having proven $\mathcal{M} \models \psi$ for all $\psi \in AX$
  implies that AX is consistent!
- recall: already just converting functional program equations naively into theorems led to proof of $0 = 1$ on slide 1/20, i.e., inconsistent axioms,
  and AX now contains more complex axioms than just equalities

Example: Attempt to Prove Associativity of Append via AX

- task: prove associativity of append via natural deduction and AX
- define $\varphi := \mathsf{append}(\mathsf{append}(xs, ys), zs) =_{\mathsf{List}} \mathsf{append}(xs, \mathsf{append}(ys, zs))$
  1. show $\forall xs, ys, zs.\, \varphi$
  2. $\forall$-intro: show φ where now xs, ys, zs are fresh variables
  3. to this end prove intermediate goal: $\forall xs.\, \varphi$
  4. applying induction axiom $\varphi[xs/\mathsf{Nil}] \longrightarrow (\forall u, us.\, \varphi[xs/us] \longrightarrow \varphi[xs/\mathsf{Cons}(u, us)]) \longrightarrow \forall xs.\, \varphi$ in combination with modus ponens yields two subgoals, one of them is $\varphi[xs/\mathsf{Nil}]$, i.e., $\mathsf{append}(\mathsf{append}(\mathsf{Nil}, ys), zs) =_{\mathsf{List}} \mathsf{append}(\mathsf{Nil}, \mathsf{append}(ys, zs))$
  5. use axiom $\forall ys.\, \mathsf{append}(\mathsf{Nil}, ys) =_{\mathsf{List}} ys$
  6. $\forall$-elim: $\mathsf{append}(\mathsf{Nil}, \mathsf{append}(ys, zs)) =_{\mathsf{List}} \mathsf{append}(ys, zs)$
  7. at this point we would like to simplify the rhs in the goal to obtain obligation $\mathsf{append}(\mathsf{append}(\mathsf{Nil}, ys), zs) =_{\mathsf{List}} \mathsf{append}(ys, zs)$
  8. this is not possible at this point: there are missing axioms
     - $=_{\mathsf{List}}$ is an equivalence relation
     - $=_{\mathsf{List}}$ is a congruence; required to simplify the lhs $\mathsf{append}(\cdot, zs)$ at $\cdot$
     - ...
- reconsider the reasoning engine and the available axioms in part 5

Summary of Part 3

- definition of well-defined functional programs
  - datatypes and function definitions (first order)
  - type-preserving equations within simple type system
  - well-defined: terminating, pattern complete and pattern disjoint
- definition of operational semantics $\hookrightarrow$
- definition of standard model
- definition of several axioms (inference rules)
  - all axioms are satisfied in standard model, so they are consistent
- upcoming
  - part 4: detect well-definedness, in particular termination
  - part 5: equational reasoning engine to prove properties of programs