Summer Term 2024

Program Verification

Part 2 – A Logic for Program Specifications

René Thiemann

Department of Computer Science

Recapitulation: Predicate Logic

Inductively Defined Sets

- one can define sets inductively via inference rules of form

$$\frac{premise_1 \quad \dots \quad premise_n}{conclusion}$$

  meaning: if all premises are satisfied, then one can conclude
- example: the set of even numbers

$$\frac{}{0 \in Even} \qquad\qquad \frac{x \in Even}{x + 2 \in Even}$$

- the inference rules describe what is contained in the set
- this can be modeled as formula

$$0 \in Even \wedge (\forall x.\ x \in Even \longrightarrow x + 2 \in Even)$$

- nothing else is in the set (this is not modeled in the formula!)

Inductively Defined Sets, Continued

- the set of even numbers

$$\frac{}{0 \in Even} \qquad\qquad \frac{x \in Even}{x + 2 \in Even}$$

- membership in the set can be proved via inference trees
- example: $4 \in Even$, proved via inference tree

$$\cfrac{\cfrac{\cfrac{}{0 \in Even}}{2 \in Even}}{4 \in Even}$$

- proving that something is not in the set is more difficult:
  show that no inference tree exists
- example: $3 \notin Even$, $-2 \notin Even$

Inductively Defined Sets and Grammars

- inference rules are similar to grammar rules
- example
  - the context-free grammar

$$S \to aSab \mid b \mid TaS \qquad\qquad T \to TT \mid \epsilon$$

  - is modeled via the inference rules

$$\frac{w \in S}{awab \in S} \qquad \frac{}{b \in S} \qquad \frac{w \in T \quad u \in S}{wau \in S}$$

$$\frac{w \in T \quad u \in T}{wu \in T} \qquad \frac{}{\epsilon \in T}$$

- in the same way, inference trees are similar to derivation trees

Inductively Defined Sets: Monotonicity

- inference rules of inductively defined sets must be monotone,
  it is not permitted to negatively refer to the currently defined set
- ill-formed example

$$\frac{}{0 \in Bad} \qquad \frac{0 \in Bad}{0 \notin Bad}$$

- one of the problems: the corresponding formula can be contradictory

$$0 \in Bad \wedge (0 \in Bad \longrightarrow 0 \notin Bad)$$

- allowed example: we define Odd, and negatively refer to previously defined $Even$

$$\frac{x \notin Even}{x \in Odd}$$

Inductively Defined Sets: Structural Induction

- example: the set of even numbers

$$\frac{}{0 \in Even} \qquad \frac{x \in Even}{x + 2 \in Even}$$

- inductively defined sets give rise to a structural induction rule
- induction rule for example, written again as inference rule:

$$\frac{y \in Even \quad P(0) \quad \forall x. P(x) \longrightarrow P(x+2)}{P(y)}$$

  where P is an arbitrary property; alternatively as formula

$$\forall y.\, y \in Even \longrightarrow \underbrace{P(0)}_{base} \longrightarrow \underbrace{(\forall x. P(x) \longrightarrow P(x+2))}_{step} \longrightarrow P(y)$$

Inductively Defined Sets: Structural Induction Continued

- depending on the structure of the inference rules there might be several base- and step-cases
- example: a definition of the set of even integers

$$\frac{}{0 \in EvenZ} \qquad \frac{x \in EvenZ}{x + 2 \in EvenZ} \qquad \frac{x \in EvenZ \quad y \in EvenZ}{x - y \in EvenZ}$$

- structural induction rule in this case contains
  - one base case (without induction hypothesis): $P(0)$
  - one step case with one induction hypothesis: $\forall x. P(x) \longrightarrow P(x+2)$
  - one step case with two induction hypotheses: $\forall x, y.\, P(x) \longrightarrow P(y) \longrightarrow P(x-y)$

Example Proof by Structural Induction

- aim: show that every even number y can be written as $2 \cdot n$
- structural induction rule

$$\frac{y \in Even \quad P(0) \quad \forall x. P(x) \longrightarrow P(x+2)}{P(y)}$$

- property $P(x)$: x can be written as $2 \cdot n$ with $n \in \mathbb{N}$; $P(x) := \exists n. n \in \mathbb{N} \wedge x = 2 \cdot n$
- semi-formal proof: apply structural induction rule to show $P(y)$
  - the subgoal $y \in Even$ is by assumption
  - the base-case $P(0)$ is trivial, since $0 = 2 \cdot 0$ and $0 \in \mathbb{N}$
  - the step-case demands a proof of $\forall x.\ P(x) \longrightarrow P(x+2)$, so let x be arbitrary, assume $P(x)$ and show $P(x+2)$
    - because of $P(x)$ there is some $n \in \mathbb{N}$ such that $x = 2 \cdot n$
    - hence $n + 1 \in \mathbb{N}$ and $x + 2 = 2 \cdot n + 2 = 2 \cdot (n+1)$
    - thus $P(x+2)$ holds by choosing $n+1$ as witness in existential quantifier
- hence, $\forall y.\, y \in Even \longrightarrow \exists n.\, n \in \mathbb{N} \wedge y = 2 \cdot n$

The Other Direction

- aim: show that $2 \cdot n \in Even$ for every natural number n
- here the structural induction rule for $Even$ is useless, since it has $y \in Even$ as a premise
- this proof is by induction on n and by using the inference rules from the inductively defined set $Even$ (and not the induction rule)

$$\frac{}{0 \in Even} \qquad \frac{x \in Even}{x + 2 \in Even}$$

- base case $n = 0$: $2 \cdot 0 = 0 \in Even$ by the first inference rule of $Even$
- step case from n to $n+1$:
  - the induction hypothesis gives us $2 \cdot n \in Even$
  - hence, $2 \cdot (n+1) = 2 \cdot n + 2 \in Even$ by the second inference rule of $Even$ (instantiate x by $2 \cdot n$)

Further Remark on Inductively Defined Sets

- so far: premises in inference rules speak about set under construction
- in general: there can be additional arbitrary side conditions
- example definition of odd numbers, assuming that $Even$ is already defined:

$$\frac{}{1 \in Odd} \qquad \frac{x \in Even \quad y \in Odd}{x + y \in Odd}$$

- structural induction adds these side conditions as additional premises

$$\frac{z \in Odd \quad P(1) \quad \forall x, y.\, x \in Even \longrightarrow P(y) \longrightarrow P(x+y)}{P(z)}$$

Final Remark on Inductively Defined Sets

- so far: we just considered sets of singleton elements
- in general: sets may contain structured data, e.g. pairs or, more generally, tuples
- example: Fibonacci numbers, $(n, x) \in Fib$ encodes that x is n-th Fibonacci number

$$\frac{}{(0,1) \in Fib} \qquad \frac{}{(1,1) \in Fib} \qquad \frac{(n,x) \in Fib \quad (n+1,y) \in Fib}{(n+2, x+y) \in Fib}$$

- since Fib consists of pairs, property in induction formula becomes a binary predicate

$$\frac{(m,z) \in Fib \quad P(0,1) \quad P(1,1) \quad \forall n, x, y.\, P(n,x) \longrightarrow P(n+1,y) \longrightarrow P(n+2, x+y)}{P(m,z)}$$

Predicate Logic: Terms

- Σ: set of (function) symbols with arity
- $\mathcal{V}$: set of variables, usually infinite
- example: $\Sigma = \{\text{plus}/2, \text{succ}/1, \text{zero}/0\}$, $\mathcal{V} = \{x, y, z, \dots\}$
- $\mathcal{T}(\Sigma, \mathcal{V})$: set of terms, inductively defined by two inference rules

$$\frac{x \in \mathcal{V}}{x \in \mathcal{T}(\Sigma, \mathcal{V})} \qquad \frac{f/n \in \Sigma \quad t_1 \in \mathcal{T}(\Sigma, \mathcal{V}) \quad \dots \quad t_n \in \mathcal{T}(\Sigma, \mathcal{V})}{f(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})}$$

- for symbols with arity 0 we omit the parenthesis in terms in formulas, i.e., we write zero as term and not zero()
- examples
  - $\text{plus}(x, \text{plus}(\text{plus}(\text{zero}, x), \text{succ}(y)))$ ✔
  - x ✔
  - plus ✘
  - $\text{plus}(x, y, z)$ ✘
- remark: we do not use infix-symbols for formal terms

Predicate Logic: Formulas

- Σ: set of function symbols, $\mathcal{V}$: set of variables
- $\mathcal{P}$: set of (predicate) symbols with arity
- $\mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})$: formulas over Σ, $\mathcal{P}$, and $\mathcal{V}$, inductively defined via

$$\frac{}{\text{true} \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})} \qquad \frac{x \in \mathcal{V} \quad \varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\forall x.\ \varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

$$\frac{\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\neg\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})} \qquad \frac{\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V}) \quad \psi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\varphi \wedge \psi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

$$\frac{p/n \in \mathcal{P} \quad t_1 \in \mathcal{T}(\Sigma, \mathcal{V}) \quad \dots \quad t_n \in \mathcal{T}(\Sigma, \mathcal{V})}{p(t_1, \dots, t_n) \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

Predicate Logic: Syntactic Sugar

- we use all Boolean connectives
  - false $= \neg$true
  - $(\varphi \vee \psi) = (\neg(\neg\varphi \wedge \neg\psi))$
  - $(\varphi \longrightarrow \psi) = (\neg\varphi \vee \psi)$
  - $(\varphi \longleftrightarrow \psi) = ((\varphi \longrightarrow \psi) \wedge (\psi \longrightarrow \varphi))$
- we permit existential quantification
  - $(\exists x.\, \varphi) = \neg(\forall x.\ \neg\varphi)$
- however, these are just abbreviations, so when defining properties of formulas, we only need to consider the connectives from the previous slide
- we use binding precedence $\neg \ > \ \wedge \ > \ \vee \ > \ \longrightarrow, \longleftrightarrow \ > \ \exists, \forall$

Predicate Logic: Semantics

- defined via models, assignments and structural recursion
- a model $\mathcal{M}$ for formulas over Σ, $\mathcal{P}$, and $\mathcal{V}$ consists of
  - a non-empty set $\mathcal{A}$, the universe
  - for each $f/n \in \Sigma$ there is a total function $f^{\mathcal{M}} : \mathcal{A}^n \to \mathcal{A}$
  - for each $p/n \in \mathcal{P}$ there is a relation $p^{\mathcal{M}} \subseteq \mathcal{A}^n$
  - an assignment is a mapping $\alpha : \mathcal{V} \to \mathcal{A}$
- the term evaluation $[\![\cdot]\!]_\alpha : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{A}$ is defined recursively as
  - $[\![x]\!]_\alpha = \alpha(x)$ and $[\![f(t_1, \dots, t_n)]\!]_\alpha = f^{\mathcal{M}}([\![t_1]\!]_\alpha, \dots, [\![t_n]\!]_\alpha)$
- the satisfaction predicate $\mathcal{M} \models_\alpha \cdot$ is defined recursively as
  - $\mathcal{M} \models_\alpha$ true
  - $\mathcal{M} \models_\alpha p(t_1, \dots, t_n)$ iff $([\![t_1]\!]_\alpha, \dots, [\![t_n]\!]_\alpha) \in p^{\mathcal{M}}$
  - $\mathcal{M} \models_\alpha \varphi \wedge \psi$ iff $\mathcal{M} \models_\alpha \varphi$ and $\mathcal{M} \models_\alpha \psi$
  - $\mathcal{M} \models_\alpha \neg\varphi$ iff $\mathcal{M} \not\models_\alpha \varphi$
  - $\mathcal{M} \models_\alpha \forall x.\ \varphi$ iff $\mathcal{M} \models_{\alpha[x:=a]} \varphi$ for all $a \in \mathcal{A}$
    where $\alpha[x := a]$ is defined as $\alpha[x := a](y) = \begin{cases} a, & \text{if } y = x \\ \alpha(y), & \text{otherwise} \end{cases}$
- if φ contains no free variables, we omit α and write $\mathcal{M} \models \varphi$

Examples

- signature: $\Sigma = \{\mathsf{plus}/2, \mathsf{succ}/1, \mathsf{zero}/0\}$, $\mathcal{P} = \{\mathsf{even}/1, =/2\}$
- model 1:
  - $\mathcal{A} = \mathbb{N}$
  - $\mathsf{plus}^{\mathcal{M}}(x,y) = x + y$, $\mathsf{succ}^{\mathcal{M}}(x) = x + 1$, $\mathsf{zero}^{\mathcal{M}} = 0$
  - $\mathsf{even}^{\mathcal{M}} = \{2 \cdot n \mid n \in \mathbb{N}\}$, $=^{\mathcal{M}} = \{(n,n) \mid n \in \mathbb{N}\}$
  - $\mathcal{M} \models \forall x, y.\, \mathsf{plus}(x,y) = \mathsf{plus}(y,x)$
- model 2:
  - $\mathcal{A} = \mathbb{Z}$
  - $\mathsf{plus}^{\mathcal{M}}(x,y) = x - y$, $\mathsf{succ}^{\mathcal{M}}(x) = |x|$, $\mathsf{zero}^{\mathcal{M}} = 42$
  - $\mathsf{even}^{\mathcal{M}} = \{2, -7\}$, $=^{\mathcal{M}} = \{(1000, 2000)\}$
  - $\mathcal{M} \not\models \forall x, y.\, \mathsf{plus}(x,y) = \mathsf{plus}(y,x)$
- model 3:
  - $\mathcal{A} = \{\bullet\}$
  - $\mathsf{plus}^{\mathcal{M}}(x,y) = \bullet$, $\mathsf{succ}^{\mathcal{M}}(x) = \bullet$, $\mathsf{zero}^{\mathcal{M}} = \bullet$
  - $\mathsf{even}^{\mathcal{M}} = \{\bullet\}$, $=^{\mathcal{M}} = \varnothing$
  - $\mathcal{M} \not\models \forall x, y.\, \mathsf{plus}(x,y) = \mathsf{plus}(y,x)$
- not a model:
  - $\mathcal{A} = \mathbb{N}$, $\mathsf{plus}^{\mathcal{M}}(x,y) = x - y$, $\mathsf{even}^{\mathcal{M}} = \{\dots, -4, -2, 0, 2, 4, \dots\}$, ...

Models for Functional Programming

- consider program

$$\mathsf{data\ Nat} = \mathsf{Zero} \mid \mathsf{Succ\ Nat}$$
$$\mathsf{data\ List} = \mathsf{Nil} \mid \mathsf{Cons\ Nat\ List}$$

- datatype definitions clearly correspond to inductively defined sets

$$\frac{}{\mathsf{Zero} \in \mathsf{Nat}} \qquad \frac{n \in \mathsf{Nat}}{\mathsf{Succ}(n) \in \mathsf{Nat}}$$
$$\frac{}{\mathsf{Nil} \in \mathsf{List}} \qquad \frac{n \in \mathsf{Nat} \quad xs \in \mathsf{List}}{\mathsf{Cons}(n, xs) \in \mathsf{List}}$$

- tentative definition of universe $\mathcal{A}$ of model $\mathcal{M}$ for program

$$\mathcal{A} = \mathsf{Nat} \cup \mathsf{List}$$

- obvious definition of meaning of constructors
  - $\mathsf{Zero}^{\mathcal{M}} = \mathsf{Zero}$, $\mathsf{Succ}^{\mathcal{M}}(n) = \mathsf{Succ}(n)$, $\mathsf{Nil}^{\mathcal{M}} = \mathsf{Nil}$, ...

A Problem in the Model

- inductively defined sets

$$\frac{}{\mathsf{Zero} \in \mathsf{Nat}} \qquad \frac{n \in \mathsf{Nat}}{\mathsf{Succ}(n) \in \mathsf{Nat}}$$
$$\frac{}{\mathsf{Nil} \in \mathsf{List}} \qquad \frac{n \in \mathsf{Nat} \quad xs \in \mathsf{List}}{\mathsf{Cons}(n, xs) \in \mathsf{List}}$$

- construction of model
  - $\mathcal{A} = \mathsf{Nat} \cup \mathsf{List}$
  - $\mathsf{Zero}^{\mathcal{M}} = \mathsf{Zero}$ and $\mathsf{Succ}^{\mathcal{M}}(n) = \mathsf{Succ}(n)$
  - $\mathsf{Nil}^{\mathcal{M}} = \mathsf{Nil}$ and $\mathsf{Cons}^{\mathcal{M}}(n, xs) = \mathsf{Cons}(n, xs)$
- problem: this is not a model
  - $\mathsf{Succ}^{\mathcal{M}}$ must be a total function of type $\mathcal{A} \to \mathcal{A}$
  - but $\mathsf{Succ}^{\mathcal{M}}(\mathsf{Nil}) = \mathsf{Succ}(\mathsf{Nil}) \notin \mathcal{A}$
- similar problem: a formula like
  $\forall xs\ ys\ zs.\ \mathsf{append}(\mathsf{append}(xs, ys), zs) = \mathsf{append}(xs, \mathsf{append}(ys, zs))$ would have to hold even when replacing xs by Zero!

Many-Sorted Logic

Solution to the One-Universe Problem

- consider many-sorted logic
- idea: a separate universe for each sort
- naming issue: sort in logic $\sim$ type in functional programming
- this lecture: we mainly speak about types
- types need to be integrated everywhere
  - typed signature
  - typed terms
  - typed formulas
  - typed assignments
  - typed quantifiers
  - typed universes
  - typed models
- this lecture: simple type system
  - no polymorphism (no generic `List a` type)
  - first-order (no λ, no partial application, ...)

Many-Sorted Predicate Logic: Syntax

- $\mathcal{T}y$: set of types where each $\tau \in \mathcal{T}y$ is just a name
  example: $\mathcal{T}y = \{\mathsf{Nat}, \mathsf{List}, \dots\}$
- Σ: set of function symbols; each $f \in \Sigma$ has type info $\in \mathcal{T}y^+$
  we write $f : \tau_1 \times \ldots \times \tau_n \to \tau_0$ whenever f has type info $\tau_1 \ldots \tau_n \tau_0$
  example: $\Sigma = \{\mathsf{Zero} : \mathsf{Nat}, \mathsf{plus} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}, \mathsf{Cons} : \mathsf{Nat} \times \mathsf{List} \to \mathsf{List}, \dots\}$
- $\mathcal{P}$: set of predicate symbols; each $p \in \mathcal{P}$ has type info $\in \mathcal{T}y^*$
  we write $p \subseteq \tau_1 \times \ldots \times \tau_n$ whenever p has type info $\tau_1 \ldots \tau_n$
  example: $\mathcal{P} = \{< \subseteq \mathsf{Nat} \times \mathsf{Nat}, =_{\mathsf{Nat}} \subseteq \mathsf{Nat} \times \mathsf{Nat}, \mathsf{even} \subseteq \mathsf{Nat},$
  $\mathsf{nonEmpty} \subseteq \mathsf{List}, =_{\mathsf{List}} \subseteq \mathsf{List} \times \mathsf{List}, \mathsf{elem} \subseteq \mathsf{Nat} \times \mathsf{List}, \dots\}$
  note: no polymorphism, so there cannot be a generic equality symbol
- $\mathcal{V}$: set of variables, typed
  example: $\mathcal{V} = \{n : \mathsf{Nat}, xs : \mathsf{List}, \dots\}$
  we write $\mathcal{V}_\tau$ as the set of variables of type τ
- notation
  - function and predicate symbols: blue color, variables: *black* color
  - often $\mathcal{T}y$ and $\mathcal{V}$ are not explicitly specified

Many-Sorted Predicate Logic: Terms

- $\mathcal{T}(\Sigma, \mathcal{V})_\tau$: set of terms of type τ, inductively defined

$$\frac{x : \tau \in \mathcal{V}}{x \in \mathcal{T}(\Sigma, \mathcal{V})_\tau}$$

$$\frac{f : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma \quad t_1 \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_1} \quad \ldots \quad t_n \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_n}}{f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau}$$

- example
  - $\mathcal{V} = \{n : \mathsf{N}, \dots\}$
  - $\Sigma = \{\mathsf{Zero} : \mathsf{N}, \mathsf{Succ} : \mathsf{N} \to \mathsf{N}, \mathsf{Nil} : \mathsf{L}, \mathsf{Cons} : \mathsf{N} \times \mathsf{L} \to \mathsf{L}\}$
  - we omit the "$\in \mathcal{V}$" and "$\in \Sigma$" when applying the inference rules
  - typing terms results in inference trees

$$\frac{\mathsf{Cons} : \mathsf{N} \times \mathsf{L} \to \mathsf{L} \quad \dfrac{\mathsf{Succ} : \mathsf{N} \to \mathsf{N} \quad \dfrac{n : \mathsf{N}}{n \in \mathcal{T}(\Sigma, \mathcal{V})_{\mathsf{N}}}}{\mathsf{Succ}(n) \in \mathcal{T}(\Sigma, \mathcal{V})_{\mathsf{N}}} \quad \dfrac{\mathsf{Nil} : \mathsf{L}}{\mathsf{Nil} \in \mathcal{T}(\Sigma, \mathcal{V})_{\mathsf{L}}}}{\mathsf{Cons}(\mathsf{Succ}(n), \mathsf{Nil}) \in \mathcal{T}(\Sigma, \mathcal{V})_{\mathsf{L}}}$$

  - for ill-typed terms such as $\mathsf{Succ}(\mathsf{Nil})$ there is no inference tree

Many-Sorted Predicate Logic: Formulas

- recall: $\mathcal{V}$, Σ and $\mathcal{P}$ are typed sets of variables, function symbols and predicate symbols
- next we define typed formulas $\mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})$ inductively
- the definition is similar as in the untyped setting
  only difference: add types to inference rule for predicates

$$\frac{}{\mathsf{true} \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})} \qquad \frac{x \in \mathcal{V} \quad \varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\forall x.\ \varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

$$\frac{\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\neg\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})} \qquad \frac{\varphi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V}) \quad \psi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}{\varphi \wedge \psi \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

$$\frac{(p \subseteq \tau_1 \times \ldots \times \tau_n) \in \mathcal{P} \quad t_1 \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_1} \quad \ldots \quad t_n \in \mathcal{T}(\Sigma, \mathcal{V})_{\tau_n}}{p(t_1, \ldots, t_n) \in \mathcal{F}(\Sigma, \mathcal{P}, \mathcal{V})}$$

Many-Sorted Predicate Logic: Semantics

- defined via typed models and assignments
- a model $\mathcal{M}$ for formulas over $\mathcal{T}y$, Σ, $\mathcal{P}$, and $\mathcal{V}$ consists of
  - a collection of non-empty universes $\mathcal{A}_\tau$, one for each $\tau \in \mathcal{T}y$
  - for each $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma$ there is a function $f^\mathcal{M} : \mathcal{A}_{\tau_1} \times \ldots \times \mathcal{A}_{\tau_n} \to \mathcal{A}_\tau$
  - for each $(p \subseteq \tau_1 \times \ldots \times \tau_n) \in \mathcal{P}$ there is a relation $p^\mathcal{M} \subseteq \mathcal{A}_{\tau_1} \times \ldots \times \mathcal{A}_{\tau_n}$
  - an assignment is a type-preserving mapping $\alpha : \mathcal{V} \to \bigcup_{\tau \in \mathcal{T}y} \mathcal{A}_\tau$, i.e., whenever $x : \tau \in \mathcal{V}$ then $\alpha(x) \in \mathcal{A}_\tau$
- the term evaluation $[\![\cdot]\!]_\alpha : \mathcal{T}(\Sigma, \mathcal{V})_\tau \to \mathcal{A}_\tau$ is defined recursively as
  - $[\![x]\!]_\alpha = \alpha(x)$
  - $[\![f(t_1, \ldots, t_n)]\!]_\alpha = f^\mathcal{M}([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha)$

  note that $[\![\cdot]\!]_\alpha$ is overloaded in the sense that it works for each type τ
- the satisfaction predicate $\mathcal{M} \models_\alpha \cdot$ is defined recursively as
  - $\mathcal{M} \models_\alpha \forall x.\ \varphi$ iff $\mathcal{M} \models_{\alpha[x:=a]} \varphi$ for all $a \in \mathcal{A}_\tau$, where τ is the type of x
  - $\mathcal{M} \models_\alpha p(t_1, \ldots, t_n)$ iff $([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha) \in p^\mathcal{M}$
  - ... remainder as in untyped setting

Example

- $\mathcal{T}y = \{\mathsf{Nat}, \mathsf{List}\}$
- $\Sigma = \{\mathsf{Zero} : \mathsf{Nat}, \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{Nil} : \mathsf{List}, \mathsf{app} : \mathsf{List} \times \mathsf{List} \to \mathsf{List}\}$
  $\mathcal{P} = \{= \subseteq \mathsf{List} \times \mathsf{List}\}$
- $\mathcal{A}_\mathsf{Nat} = \mathbb{N}$
- $\mathcal{A}_\mathsf{List} = \{[x_1, \ldots, x_n] \mid n \in \mathbb{N}, \forall 1 \leq i \leq n.\, x_i \in \mathbb{N}\}$
- $\mathsf{Zero}^\mathcal{M} = 0$
- $\mathsf{Succ}^\mathcal{M}(n) = n + 1$
  definition is okay: n can be no list, since $n \in \mathcal{A}_\mathsf{Nat} = \mathbb{N}$
- $\mathsf{Nil}^\mathcal{M} = [\,]$
- $\mathsf{app}^\mathcal{M}([x_1, \ldots, x_n], [y_1, \ldots, y_m]) = [x_1, \ldots, x_n, y_1, \ldots, y_m]$
  again, this is sufficiently defined, since the arguments of $\mathsf{app}^\mathcal{M}$ are two lists
- $=^\mathcal{M} = \{(xs, xs) \mid xs \in \mathcal{A}_\mathsf{List}\}$
- $\mathcal{M} \models \forall xs, ys, zs.\, \mathsf{app}(xs, \mathsf{app}(ys, zs)) = \mathsf{app}(\mathsf{app}(xs, ys), zs)$
- $\mathcal{M} \not\models \forall xs.\, \mathsf{app}(xs, xs) = xs \qquad \mathcal{M} \models \exists xs.\, \mathsf{app}(xs, xs) = xs$

Many-Sorted Predicate Logic: Well-Definedness

- consider the term evaluation
  - $[\![x]\!]_\alpha = \alpha(x)$
  - $[\![f(t_1, \ldots, t_n)]\!]_\alpha = f^\mathcal{M}([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha)$
- it was just stated that this a function of type $[\![\cdot]\!]_\alpha : \mathcal{T}(\Sigma, \mathcal{V})_\tau \to \mathcal{A}_\tau$
- similarly, the definition
  - $\mathcal{M} \models_\alpha p(t_1, \ldots, t_n)$ iff $([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha) \in p^\mathcal{M}$

  has to be taken with care: we need to ensure that $([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha)$ and $p^\mathcal{M}$ fit together, such that the membership test is type-correct
- in general, such type-preservation statements need to be proven!
- however, often this is not even mentioned

Type-Checking

Type-Checking

- inference trees are proofs that certain terms have a certain type
- inference trees cannot be used to show that a term is not typable
- want: executable algorithm that given Σ, $\mathcal{V}$, and a candidate term, computes the type or detects failure
- in Haskell: function definition with type
  `typeCheck :: Sig -> Vars -> Term -> Maybe Type`
- preparation: error handling in Haskell with monads

Explicit Error-Handling with Maybe

- recall Haskell's builtin type
  `data Maybe a = Just a | Nothing`
- useful to distinguish successful from non-successful computations
  - `Just x` represents successful computation with result value `x`
  - `Nothing` represents that some error occurred
- example for explicit error handling: evaluating an arithmetic expression

```
data Expr = Var String | Plus Expr Expr | Div Expr Expr

eval :: (String -> Integer) -> Expr -> Maybe Integer
eval alpha (Var x)      = Just (alpha x)
eval alpha (Plus e1 e2) = case (eval alpha e1, eval alpha e2) of
  (Just x1, Just x2) -> Just (x1 + x2)
  _  -> Nothing
eval alpha (Div e1 e2)  = case (eval alpha e1, eval alpha e2) of
  (Just x1, Just x2) ->
     if x2 /= 0 then Just (x1 `div` x2) else Nothing
  _  -> Nothing
```


Error-Handling with Monads

- recall Haskell's I/O-monad
  - `IO a` internally stores a state (the world) and returns result of type `a`
  - with `do`-blocks, we can sequentially perform IO-actions, and receive intermediate values; core function for sequential composition: `(>>=) :: IO a -> (a -> IO b) -> IO b`
  - example

```
greeting = do
  x <- getLine    -- IO String, action: read user input
  putStr "hello " -- IO (), action: print something
  putStr x        -- IO (), action: print something
  return (x ++ x) -- IO String, no action, return result
```

- also `Maybe` can be viewed as monad
  - `Maybe a` internally stores a state (successful or error) and returns result of type `a`
  - core functions for Maybe-monad
    - `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b`
      ```
      Nothing >>= _ = Nothing -- errors propagate
      Just x  >>= f = f x
      ```
    - `return :: a -> Maybe a`
      ```
      return x = Just x
      ```


Monads in Haskell

- Haskell's I/O-monad
  - `(>>=) :: IO a -> (a -> IO b) -> IO b`
  - `return :: a -> IO a`
- the error monad of type `Maybe a`
  - `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b`
  - `return :: a -> Maybe a`
- generalization: arbitrary monads via type-class

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

  - `IO` and `Maybe` are instances of `Monad`
  - do-notation is available for all monads
  - monad-instances should satisfy the three monad laws

```
(return x) >>= f   =  f x
m >>= return       =  m
(m >>= f) >>= g    =  m >>= (\ x -> f x >>= g)
```

Example: Expression-Evaluation in Monadic Style

```
data Expr = Var String | Plus Expr Expr | Div Expr Expr

eval :: (String -> Integer) -> Expr -> Maybe Integer
eval alpha (Var x)       = return (alpha x)
eval alpha (Plus e1 e2) = do
  x1 <- eval alpha e1
  x2 <- eval alpha e2
  return (x1 + x2)
eval alpha (Div e1 e2)  = do
  x1 <- eval alpha e1
  x2 <- eval alpha e2
  if x2 /= 0 then return (x1 `div` x2) else Nothing
```

- advantages
  - no pattern-matching on `Maybe`-type required any more, more readable code; hence monadic style simplifies reasoning about these programs
  - easy to switch to other monads, e.g. for errors with messages
  - Prelude already contains several functions for monads

Example Library Function for Monads

- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
  - similar to `map :: (a -> b) -> [a] -> [b]`, just in monadic setting
  - applies a monadic function sequentially to all list elements
  - possible implementation

```
mapM f [] = return []
mapM f (x : xs) = do
  y <- f x
  ys <- mapM f xs
  return (y : ys)
```

  - consequence for `Maybe`-monad:

    `mapM f [x_1, ..., x_n] = return ys`

    is satisfied iff
    - `f x_i = return y_i` for all $1 \leq i \leq n$, and
    - `ys = [y_1, ..., y_n]`

Type-Checking Algorithm

- back to type-checking
- the algorithm can now be defined concisely as

```
type Type = String
type Var  = String
type FSym = String
type Vars = Var -> Maybe Type
type FSymInfo = ([Type], Type)
type Sig = FSym -> Maybe FSymInfo
data Term = Var Var | Fun FSym [Term]

typeCheck :: Sig -> Vars -> Term -> Maybe Type
typeCheck sigma vars (Var x) = vars x
typeCheck sigma vars (Fun f ts) = do
  (tysIn,tyOut) <- sigma f
  tysTs <- mapM (typeCheck sigma vars) ts
  if tysTs == tysIn then return tyOut else Nothing
```


Correctness of Type-Checking

- aim: prove correctness of type-checking algorithm
- (informal) proof is performed in two steps
  - if $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ then `typeCheck sigma vars t = return tau`
  - if `typeCheck sigma vars t = return tau` then $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$
- before these two steps are done, some alignment of the representation is performed
  - in the theory $\mathcal{V}$ is set of type-annotated variables
  - in the program `vars` is a partial function from variables to types
  - obviously, these two representations can be aligned:

    $x : \tau \in \mathcal{V}$ is the same as `vars x = return tau`
  - similarly for function symbols we demand that

    $$f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \Sigma$$

    is the same as

    `sigma f = return ([tau_1,...,tau_n], tau_0)`
  - moreover the term representations can be aligned, e.g.

    $f(t_1, \ldots, t_n)$ is the same as `Fun f [t_1, ..., t_n]`

  from now on we mainly use mathematical notation assuming the obvious alignments, even when executing Haskell programs

Completeness of Type-Checking Algorithm

if $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ then $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau$

- proof is by structural induction according to the definition of $\mathcal{T}(\Sigma,\mathcal{V})_\tau$
- note that in the definition of the inductively defined set $\mathcal{T}(\Sigma,\mathcal{V})_\tau$ the τ changes; therefore, the induction rule uses a binary property:

$$\frac{t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau \quad \forall x, \tau_0.\, x : \tau_0 \in \mathcal{V} \longrightarrow P(x,\tau_0) \quad (*)}{P(t,\tau)}$$

$$\forall f, \tau_0, \ldots, \tau_n, t_1, \ldots, t_n.\, f : \tau_1 \times \ldots \times \tau_n \to \tau_0 \in \Sigma \longrightarrow \qquad (*)$$
$$P(t_1,\tau_1) \longrightarrow \ldots \longrightarrow P(t_n,\tau_n) \longrightarrow P(f(t_1,\ldots,t_n),\tau_0)$$

- in our case $P(t,\tau)$ is $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau$
- base case:
  - let $x : \tau_0 \in \mathcal{V}$, aim is to prove $P(x,\tau_0)$
  - via the alignment we know $\mathcal{V}\ x = \mathit{return}\ \tau_0$ (where $\mathcal{V}$ refers to the partial function within the algorithm)
  - hence by the definition of the algorithm: $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ x = \mathcal{V}\ x = \mathit{return}\ \tau_0$

Completeness of Type-Checking Algorithm

recall: $P(t,\tau)$ is $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau$

- it remains to prove $(*)$, so let $f : \tau_1 \times \ldots \times \tau_n \to \tau_0 \in \Sigma$
- we have to prove $P(f(t_1,\ldots,t_n),\tau_0)$ using the induction hypothesis $P(t_i,\tau_i)$ for all $1 \le i \le n$
- via the alignment we know $\Sigma\, f = \mathit{return}\ ([\tau_1,\ldots,\tau_n],\tau_0)$
- from the induction hypothesis we know that $\mathit{map}\ (\mathit{typeCheck}\ \Sigma\ \mathcal{V})\ [t_1,\ldots,t_n] = [\mathit{return}\ \tau_1,\ldots,\mathit{return}\ \tau_n]$
- hence, by the definition of mapM, $\mathit{mapM}\ (\mathit{typeCheck}\ \Sigma\ \mathcal{V})\ [t_1,\ldots,t_n] = \mathit{return}\ [\tau_1,\ldots,\tau_n]$
- hence by evaluating the Haskell-code we obtain
  $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ f(t_1,\ldots,t_n)$
  $= \mathit{if}\ [\tau_1,\ldots,\tau_n] = [\tau_1,\ldots,\tau_n]\ \mathit{then}\ \mathit{return}\ \tau_0\ \mathit{else}\ \mathit{Nothing}$
  $= \mathit{return}\ \tau_0$
  so $P(f(t_1,\ldots,t_n),\tau_0)$ is satisfied

Soundness of Type-Checking Algorithm

if $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau$ then $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$

- we perform structural induction on t (w.r.t. untyped terms as defined by the Haskell datatype definition)
- the induction rule only mentions a unary property

$$\frac{\forall x.\, P(\mathit{Var}\ x) \quad (*)}{P(t : \mathit{Term})}$$

$$\forall f, t_1, \ldots, t_n.\, P(t_1) \longrightarrow \ldots \longrightarrow P(t_n) \longrightarrow P(f(t_1,\ldots,t_n)) \qquad (*)$$

- first attempt: define $P(t)$ as

$$\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$$

- then the induction hypothesis in the case $f(t_1,\ldots,t_n)$ for each t_i is

$$P(t_i) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t_i = \mathit{return}\ \tau \longrightarrow t_i \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$

- the IH is unusable as t_i will have type τ_i which in general differs from τ

Induction Proofs with Arbitrary Variables

- previous slide: using

$$P(t) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$

  as property in induction rule is too restrictive, leads to IH

$$P(t_i) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t_i = \mathit{return}\ \tau \longrightarrow t_i \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$

- aim: ability to use arbitrary τ_i in IH instead of τ
- formal solution via universal quantification:
  define P and Q as follows and use P in induction

$$Q(t,\tau) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$
$$P(t) = (\forall \tau.\ Q(t,\tau))$$

- effect: induction hypothesis for t_i will be $P(t_i) = (\forall \tau.\ Q(t_i,\tau))$ which in particular implies the desired $Q(t_i,\tau_i)$

Induction Proofs with Arbitrary Variables

- previous slide:

$$Q(t,\tau) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$
$$P(t) = (\forall \tau.\ Q(t,\tau))$$

- we now prove $P(t)$ by induction on t, this time being quite formal
- base case: $t = \mathit{Var}\ x$
  - we have to show $P(t) = P(\mathit{Var}\ x) = (\forall\tau.\ Q(\mathit{Var}\ x,\tau))$
  - ◦ $\forall$-intro: pick an arbitrary τ and show $Q(\mathit{Var}\ x,\tau)$, i.e., $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ (\mathit{Var}\ x) = \mathit{return}\ \tau \longrightarrow x \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - $\longrightarrow$-intro: assume $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ (\mathit{Var}\ x) = \mathit{return}\ \tau$, and then show $x \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - simplify assumption $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ (\mathit{Var}\ x) = \mathit{return}\ \tau$ to $\mathcal{V}\ x = \mathit{return}\ \tau$
  - by alignment this is identical to $x : \tau \in \mathcal{V}$
  - use introduction rule of $\mathcal{T}(\Sigma,\mathcal{V})_\tau$ to finally show $x \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$

  note that step ◦ is the only additional (but obvious) step that was required to deal with the auxiliary universal quantifier

Induction Proofs with Arbitrary Variables: Step Case

$$Q(t,\tau) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$
$$P(t) = (\forall \tau.\ Q(t,\tau))$$

- step case: $t = f(t_1,\ldots,t_n)$
  - we have to show $P(f(t_1,\ldots,t_n)) = (\forall\tau.\ Q(f(t_1,\ldots,t_n),\tau))$
  - ◦ $\forall$-intro: pick an arbitrary τ and show $Q(f(t_1,\ldots,t_n),\tau)$, i.e., $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ f(t_1,\ldots,t_n) = \mathit{return}\ \tau \longrightarrow f(t_1,\ldots,t_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - $\longrightarrow$-intro: assume $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ f(t_1,\ldots,t_n) = \mathit{return}\ \tau$, and show $f(t_1,\ldots,t_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
  - by the assumption $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ f(t_1,\ldots,t_n) = \mathit{return}\ \tau$ and by definition of $\mathit{typeCheck}$, we know that there must be types $\tau_1,\ldots,\tau_n$ such that $\mathit{mapM}\ (\mathit{typeCheck}\ \Sigma\ \mathcal{V})\ [t_1,\ldots,t_n] = \mathit{return}\ [\tau_1,\ldots,\tau_n]$, and hence $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t_i = \mathit{return}\ \tau_i$ for all $1 \le i \le n$
  - again using the assumption and the algorithm definition we conclude that $\Sigma\ f = \mathit{return}\ ([\tau_1,\ldots,\tau_n],\tau)$ and thus, $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma$
  - ◦ by the IH we conclude $P(t_i)$ and hence $Q(t_i,\tau_i)$ using $\forall$-elimination
  - in combination with $\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t_i = \mathit{return}\ \tau_i$ we arrive at $t_i \in \mathcal{T}(\Sigma,\mathcal{V})_{\tau_i}$ and can finally apply the introduction rules for typed terms to conclude $f(t_1,\ldots,t_n) \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$

Induction Proofs with Arbitrary Variables: Remarks

$$Q(t,\tau) = (\mathit{typeCheck}\ \Sigma\ \mathcal{V}\ t = \mathit{return}\ \tau \longrightarrow t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau)$$
$$P(t) = (\forall \tau.\ Q(t,\tau))$$

- the method to make a variable arbitrary within an induction proof is always the same, via universal quantification
- the required steps within the formal reasoning (marked with ◦ in the previous proof) are also automatic
- therefore, in the following we will just write statements like

  "we perform induction on x for arbitrary y and z"

  or

  "we prove $P(x,y,z)$ by induction on x for arbitrary y and z"

  without doing the universal quantification explicitly
- the effect of introducing arbitrary variables is a generalization: instead of proving $P(x,y,z)$ for a fixed y and z, we show it for all y and z

Summary of Type-Checking

- definition of typed terms via inference rules
- equivalent definition via type-checking algorithm
- both representations have their advantages
  - inference rules come with convenient induction principle
  - type-checking can also detect typing errors, i.e., it can show that something is not member of an inductively defined set
- note: we have verified a first non-trivial program!
  - given the precise semantics of typed terms
  - via an intuitive meaning of what inductively defined sets are
  - with an intuitive meaning of how Haskell evaluates
  - with intuitively created alignments

Summary of Chapter

- inductively defined sets give rise to structural induction rule
- inductively defined sets can be used to model datatypes of
  (first-order non-polymorphic) functional programs
- many sorted/typed terms and predicate logic allows adequate modeling of datatypes
- verified type-checking algorithm
- induction proofs with “arbitrary” variables