

Data Type Definitions: Examples

- $\mathcal{T}_y = \Sigma = \mathcal{P} = \emptyset$
- **data** $\text{Nat} = \text{Zero} : \text{Nat} \mid \text{Succ} : \text{Nat} \rightarrow \text{Nat}$
- processing updates $\mathcal{T}_y = \{\text{Nat}\}$,
 $\Sigma = \{\text{Zero} : \text{Nat}, \text{Succ} : \text{Nat} \rightarrow \text{Nat}\}$
and $\mathcal{P} = \{=_{\text{Nat}} \subseteq \text{Nat} \times \text{Nat}\}$
- **data** $\text{List} = \text{Nil} : \text{List} \mid \text{Cons} : \text{Nat} \times \text{List} \rightarrow \text{List}$
- processing updates $\mathcal{T}_y = \{\text{Nat}, \text{List}\}$,
 $\Sigma = \{\text{Zero} : \text{Nat}, \text{Succ} : \text{Nat} \rightarrow \text{Nat}, \text{Nil} : \text{List}, \text{Cons} : \text{Nat} \times \text{List} \rightarrow \text{List}\}$
and $\mathcal{P} = \{=_{\text{Nat}} \subseteq \text{Nat} \times \text{Nat}, =_{\text{List}} \subseteq \text{List} \times \text{List}\}$
- **data** $\text{BList} = \text{NilB} : \text{BList} \mid \text{ConsB} : \text{Bool} \times \text{BList} \rightarrow \text{BList}$
not allowed, since $\text{Bool} \notin \mathcal{T}_y$
- **data** $\text{LList} = \text{Nil} : \text{LList} \mid \text{Cons} : \text{List} \times \text{LList} \rightarrow \text{LList}$
not allowed, since Nil and Cons are already in Σ
- **data** $\text{Tree} = \text{Node} : \text{Tree} \times \text{Nat} \times \text{Tree} \rightarrow \text{Tree}$
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

$$\text{data } \tau = \begin{array}{l} c_1 : \tau_{1,1} \times \dots \times \tau_{1,m_1} \rightarrow \tau \\ \dots \\ c_n : \tau_{n,1} \times \dots \times \tau_{n,m_n} \rightarrow \tau \end{array}$$

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

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

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

Data Type Definitions: Example and Standard Model

- **data** $\text{Nat} = \text{Zero} : \text{Nat} \mid \text{Succ} : \text{Nat} \rightarrow \text{Nat}$
- processing creates universe \mathcal{A}_{Nat} via the inference rules

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

i.e., $\mathcal{A}_{\text{Nat}} = \{\text{Zero}, \text{Succ}(\text{Zero}), \text{Succ}(\text{Succ}(\text{Zero})), \dots\}$

- $\text{Zero}^{\mathcal{M}} = \text{Zero}$ $\text{Succ}^{\mathcal{M}}(t) = \text{Succ}(t)$
- $=_{\text{Nat}}^{\mathcal{M}} = \{(\text{Zero}, \text{Zero}), (\text{Succ}(\text{Zero}), \text{Succ}(\text{Zero})), \dots\}$
- **data** $\text{List} = \text{Nil} : \text{List} \mid \text{Cons} : \text{Nat} \times \text{List} \rightarrow \text{List}$
- processing creates universe $\mathcal{A}_{\text{List}}$ via the inference rules

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

i.e., $\mathcal{A}_{\text{List}} = \{\text{Nil}, \text{Cons}(\text{Zero}, \text{Nil}), \text{Cons}(\text{Succ}(\text{Zero}), \text{Nil}), \dots\}$

- $=_{\text{List}}^{\mathcal{M}} = \{(\text{Nil}, \text{Nil}), (\text{Cons}(\text{Zero}, \text{Nil}), \text{Cons}(\text{Zero}, \text{Nil})), \dots\}$

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}_τ well-defined, i.e., $\mathcal{A}_\tau \neq \emptyset$
- recall: each data definition has the following form

$$\text{data } \tau = \begin{array}{l} c_1 : \tau_{1,1} \times \dots \times \tau_{1,m_1} \rightarrow \tau \\ \dots \\ c_n : \tau_{n,1} \times \dots \times \tau_{n,m_n} \rightarrow \tau \end{array}$$

where

- $\tau \notin \mathcal{T}_y$ fresh type name
- $c_1, \dots, c_n \notin \Sigma$ 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)

```
data Nat = Succ : Nat → Nat
```

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

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

- consequence: $\mathcal{A}_{\text{Nat}} = \emptyset$
- 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 $\mathcal{A}_{\tau} \neq \emptyset$.

Proof

Let τ_1, \dots, τ_n be the sequence of types that have been defined. We show

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

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 $\mathcal{A}_{\tau_{n+1}} \neq \emptyset$. By the definition of data types, there must be some $c_i : \tau_{i,1} \times \dots \times \tau_{i,m_i} \rightarrow \tau_{n+1}$ where all $\tau_{i,j} \in \{\tau_1, \dots, \tau_n\}$. By the IH $P(n)$ we know that $\mathcal{A}_{\tau_{i,j}} \neq \emptyset$ for all j between 1 and m_i . Hence, there must be terms $t_1 \in \mathcal{A}_{\tau_{i,1}}, \dots, t_{m_i} \in \mathcal{A}_{\tau_{i,m_i}}$. Consequently, $c_i(t_1, \dots, t_{m_i}) \in \mathcal{A}_{\tau_{n+1}}$, and hence $\mathcal{A}_{\tau_{n+1}} \neq \emptyset$.

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 `Cons(Zero, Cons(Zero, ...))` (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** (start with capital letters in Haskell)
e.g., **Nil**, **Succ**, **Cons**
 - defined functions**, declared via equations (start with 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}_τ is exactly $\mathcal{T}(\mathcal{C})_\tau := \mathcal{T}(\mathcal{C}, \emptyset)_\tau$, which is the set of **constructor ground terms** of type τ

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
 - reverse**(**Cons**(x , **Cons**(y , xs)))
 - reverse**(**Cons**(x , **Cons**(x , xs)))
- the **variables of a term** t are defined as $\mathcal{V}ars(t)$
 - $\mathcal{V}ars(x) = \{x\}$
 - $\mathcal{V}ars(F(t_1, \dots, t_n)) = \mathcal{V}ars(t_1) \cup \dots \cup \mathcal{V}ars(t_n)$



Function Definitions

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

$$f : \tau_1 \times \dots \times \tau_n \rightarrow \tau$$

$$\ell_1 = r_1$$

$$\dots = \dots$$

$$\ell_m = r_m$$

where

- f is a **fresh name** and $\mathcal{D} := \mathcal{D} \cup \{f : \tau_1 \times \dots \times \tau_n \rightarrow \tau\}$ (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** $\mathcal{V}ars(r_i) \subseteq \mathcal{V}ars(\ell_i)$

Function Definitions: Examples

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

add : **Nat** × **Nat** → **Nat**

add(**Zero**, y) = y

add(**Succ**(x), y) = **add**(x , **Succ**(y))

append : **List** × **List** → **List**

append(**Cons**(x , xs), ys) = **Cons**(x , **append**(xs , ys))

append(xs , ys) = ys

head : **List** → **Nat**

head(**Cons**(x , xs)) = x

zeros : **List**

zeros = **Cons**(**Zero**, **zeros**)

Function Definitions: Non-Examples

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

`even : Nat → Bool`

`even(Zero) = True`

`even(Succ(x)) = odd(x)` ❌

`odd : Nat → Bool`

`odd(Zero) = False`

`odd(Succ(x)) = even(x)` ❌

`random : Nat`

`random = x` ❌

`minus : Nat × Nat → Nat`

`minus(Succ(x), Succ(y)) = minus(x, y)`

`minus(x, Zero) = x`

`minus(x, x) = Zero` ❌

`minus(add(x, y), x) = y` ❌

Semantics for Function Definitions

- problem: given a function definition

$$f : \tau_1 \times \dots \times \tau_n \rightarrow \tau$$

$$\ell_1 = r_1$$

$$\dots = \dots$$

$$\ell_m = r_m$$

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

$$f^{\mathcal{M}} : \mathcal{A}_{\tau_1} \times \dots \times \mathcal{A}_{\tau_n} \rightarrow \mathcal{A}_{\tau}$$

or equivalently

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

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

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

Semantics for Function Definitions – Continued

- required: $f^{\mathcal{M}} : \mathcal{T}(\mathcal{C})_{\tau_1} \times \dots \times \mathcal{T}(\mathcal{C})_{\tau_n} \rightarrow \mathcal{T}(\mathcal{C})_{\tau}$
- idea: define $f^{\mathcal{M}}(t_1, \dots, t_n)$ as
the result of $f(t_1, \dots, t_n)$ 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

$$\text{add}(\text{Zero}, y) = y \quad (1)$$

$$\text{add}(\text{Succ}(x), y) = \text{add}(x, \text{Succ}(y)) \quad (2)$$

$$\text{append}(\text{Cons}(x, xs), ys) = \text{Cons}(x, \text{append}(xs, ys)) \quad (3)$$

$$\text{append}(xs, ys) = ys \quad (4)$$

$$\text{head}(\text{Cons}(x, xs)) = x \quad (5)$$

$$\text{zeros} = \text{Cons}(\text{Zero}, \text{zeros}) \quad (6)$$

- is result **unique**? no: consider $t = \text{append}(\text{Cons}(\text{Zero}, \text{Nil}), \text{Nil})$
then $t \stackrel{(3)}{=} \text{Cons}(\text{Zero}, \text{append}(\text{Nil}, \text{Nil})) \stackrel{(4)}{=} \text{Cons}(\text{Zero}, \text{Nil})$
and $t \stackrel{(4)}{=} \text{Nil}$
- is result element of $\mathcal{T}(\mathcal{C})_{\tau}$? no: $\text{head}(\text{Nil})$ cannot be evaluated
- does evaluation **terminate**? no: $\text{zeros} = \text{Cons}(\text{Zero}, \text{zeros}) = \dots$
- 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 $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$ to a term, can be defined recursively
 - $x\sigma = \sigma(x)$
 - $F(t_1, \dots, t_n)\sigma = F(t_1\sigma, \dots, t_n\sigma)$
- one-step evaluation relation $\hookrightarrow \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$ defined as inductive set

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

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

- 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 $P = \{(\ell_1, t_1), \dots, (\ell_n, t_n)\}$ and the task is to decide: $\exists\sigma. \ell_1\sigma = t_1 \wedge \dots \wedge \ell_n\sigma = t_n$, i.e.,
 - either return the required substitution σ in the form of a set of pairs $\{(x_1, s_1), \dots, (x_m, s_m)\}$ 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 \perp indicating that no such substitution exists
- matching algorithm**: apply rules \rightsquigarrow as long as possible

$$P \uplus \{(F(\ell_1, \dots, \ell_n), F(t_1, \dots, t_n))\} \rightsquigarrow P \cup \{(\ell_1, t_1), \dots, (\ell_n, t_n)\} \quad \text{(decompose)}$$

$$P \uplus \{(F(\dots), G(\dots))\} \rightsquigarrow \perp \quad \text{if } F \neq G \quad \text{(clash)}$$

$$P \uplus \{(F(\dots), x)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \quad \text{(fun-var)}$$

$$P \uplus \{(x, s), (x, t)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \text{ and } s \neq t \quad \text{(var-clash)}$$

Matching – Example

- we want to test whether there is a root step possible for the term $t = \text{append}(\text{Cons}(y, \text{Nil}), \text{Cons}(y, ys))$ w.r.t. the equation $(\ell = r) = (\text{append}(\text{Cons}(x, xs), ys) = \text{Cons}(x, \text{append}(xs, ys)))$
- setup matching problem $\{(\ell, t)\}$

$$P = \{(\text{append}(\text{Cons}(x, xs), ys), \text{append}(\text{Cons}(y, \text{Nil}), \text{Cons}(y, ys)))\}$$

$$\rightsquigarrow \{(\text{Cons}(x, xs), \text{Cons}(y, \text{Nil})), (ys, \text{Cons}(y, ys))\}$$

$$\rightsquigarrow \{(x, y), (xs, \text{Nil}), (ys, \text{Cons}(y, ys))\}$$

- obtain substitution $\sigma(z) = \begin{cases} y, & \text{if } z = x \\ \text{Nil}, & \text{if } z = xs \\ \text{Cons}(y, ys), & \text{if } z = ys \\ z, & \text{otherwise} \end{cases}$
- so, $t = \ell\sigma \hookrightarrow r\sigma = \text{Cons}(x, \text{append}(xs, ys))\sigma = \text{Cons}(y, \text{append}(\text{Nil}, \text{Cons}(y, ys)))$

Matching – Verification and Termination Proof

- matching algorithm

$$P \uplus \{(F(\ell_1, \dots, \ell_n), F(t_1, \dots, t_n))\} \rightsquigarrow P \cup \{(\ell_1, t_1), \dots, (\ell_n, t_n)\} \quad (\text{decompose})$$

$$P \uplus \dots \rightsquigarrow \perp \quad (\text{other rules})$$

- 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

$$P \uplus \{(F(\ell_1, \dots, \ell_n), F(t_1, \dots, t_n))\} \rightsquigarrow P \cup \{(\ell_1, t_1), \dots, (\ell_n, t_n)\} \quad (\text{decompose})$$

$$P \uplus \dots \rightsquigarrow \perp \quad (\text{other rules})$$

- 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 $\neq \perp$ will be type-correct
- proof: we prove an invariant, so we only need to prove that the property is maintained when performing a single \rightsquigarrow -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 $F : \tau_1 \times \dots \times \tau_n \rightarrow \tau$ 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: apply \rightsquigarrow as long as possible

$$P \uplus \{(F(\ell_1, \dots, \ell_n), F(t_1, \dots, t_n))\} \rightsquigarrow P \cup \{(\ell_1, t_1), \dots, (\ell_n, t_n)\} \quad (\text{decompose})$$

$$P \uplus \{(F(\dots), G(\dots))\} \rightsquigarrow \perp \quad \text{if } F \neq G \quad (\text{clash})$$

$$P \uplus \{(F(\dots), x)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \quad (\text{fun-var})$$

$$P \uplus \{(x, s), (x, t)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \text{ and } s \neq t \quad (\text{var-clash})$$

- property: **result of matching algorithm** on well-typed inputs is \perp or set $\{(x_1, s_1), \dots, (x_m, s_m)\}$ with all x_i distinct
- proof
 - assume result is not \perp , 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 \perp , cf. “clash” and “fun-var”
 - then $k = n$ because of type preservation: contradiction to “decompose”

Matching – Preservation of Solutions

- matching algorithm

$$P \uplus \{(F(\ell_1, \dots, \ell_n), F(t_1, \dots, t_n))\} \rightsquigarrow P \cup \{(\ell_1, t_1), \dots, (\ell_n, t_n)\} \quad (\text{decompose})$$

$$P \uplus \{(F(\dots), G(\dots))\} \rightsquigarrow \perp \quad \text{if } F \neq G \quad (\text{clash})$$

$$P \uplus \{(F(\dots), x)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \quad (\text{fun-var})$$

$$P \uplus \{(x, s), (x, t)\} \rightsquigarrow \perp \quad \text{if } x \in \mathcal{V} \text{ and } s \neq t \quad (\text{var-clash})$$

- property: algorithm **preserves matching substitutions** (where \perp has no matching substitution)
- proof by considering invariant of single step: whenever $P \rightsquigarrow P'$, then σ is a matcher of P iff σ is matcher of P'
 - clash: both “ σ is matcher of $\{(F(\dots), G(\dots))\} \cup P$ ” and “ σ is matcher of \perp ” are wrong: $F(t_1, \dots)\sigma = F(t_1\sigma, \dots) \neq G(\dots)$
 - fun-var and var-clash are similar
 - decompose: $F(\ell_1, \dots, \ell_n)\sigma = F(t_1, \dots, t_n)$
 - $\iff F(\ell_1\sigma, \dots, \ell_n\sigma) = F(t_1, \dots, t_n)$
 - $\iff \ell_1\sigma = t_1 \wedge \dots \wedge \ell_n\sigma = t_n$

Matching Algorithm – Summary

- algorithm: apply \curvearrowright as long as possible
- (one) termination proof
- (many) partial correctness proofs
mainly by showing invariants that are preserved by \curvearrowright
 - type preservation
 - preservation of matching substitutions
 - result is \perp 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 exist, 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: $F(s_1, \dots, s_i, \dots, s_n) \hookrightarrow F(s_1, \dots, t_i, \dots, s_n)$ since $s_i \hookrightarrow t_i$, we know $F(s_1, \dots, s_i, \dots, s_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ and have to prove $F(s_1, \dots, t_i, \dots, s_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$
 - since $F(s_1, \dots, s_i, \dots, s_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$, we know that $F : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \Sigma$ 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 $F(s_1, \dots, t_i, \dots, s_n) \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$

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 on the number of steps by 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 $\text{Vars}(t) = \emptyset$, 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 -step is 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 $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \mathcal{D}$ is **pattern complete** iff for all $t_1 \in \mathcal{T}(\mathcal{C})_{\tau_1}, \dots, t_n \in \mathcal{T}(\mathcal{C})_{\tau_n}$ there is an equation $\ell = r$ in the program, such that ℓ matches $f(t_1, \dots, t_n)$
- a functional program is **pattern complete** iff all $f \in \mathcal{D}$ are pattern complete
- example

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

$$\text{append}(\text{Nil}, ys) = ys$$

$$\text{head}(\text{Cons}(x, xs)) = x$$

- **append** is pattern complete
- **head** is not pattern complete: for $\text{head}(\text{Nil})$ 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: $t = F(t_1, \dots, t_n)$ where $F : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \Sigma$
- IH for each i : if t_i is normal form, then $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$
- premise: $F(t_1, \dots, t_n)$ is normal form
- from premise conclude that t_i is normal form:
(if $t_i \hookrightarrow s_i$ then $F(t_1, \dots, t_n) \hookrightarrow F(t_1, \dots, s_i, \dots, t_n)$ shows that $F(t_1, \dots, t_n)$ 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 $F(t_1, \dots, t_n) \in \mathcal{T}(\mathcal{C})_{\tau}$
- case $F \in \mathcal{D}$: using pattern completeness and $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$, conclude that $F(t_1, \dots, t_n)$ must be matched by lhs; this is contradiction to $F(t_1, \dots, t_n)$ being a normal form

Pattern Disjointness

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

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

$$\text{append}(xs, ys) = ys$$

$$\text{head}(\text{Cons}(x, xs)) = x$$

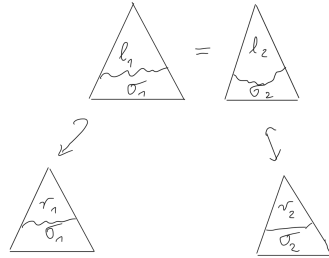
- **head** is pattern disjoint
- **append** is not pattern disjoint: the term $\text{append}(\text{Cons}(\text{Zero}, \text{Nil}), \text{Nil})$ is matched by the lhs of both **append**-equations

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 system**
 - Rosen proved that **orthogonal** term rewrite systems 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

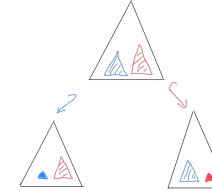
- consider the situation in the diagram where two root steps with equations $l_1 = r_1$ and $l_2 = r_2$ are applied



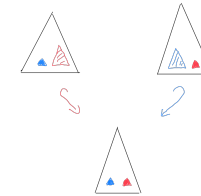
- because of pattern disjointness: $(l_1 = r_1) = (l_2 = r_2)$
- uniqueness of matching: $\sigma_1(x) = \sigma_2(x)$ for all $x \in \text{Vars}(l_{1/2})$
- variable condition of programs: $\sigma_1(x) = \sigma_2(x)$ for all $x \in \text{Vars}(r_{1/2})$
- hence $r_1\sigma_1 = r_2\sigma_2$

Proof of Local Confluence: Independent Steps

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

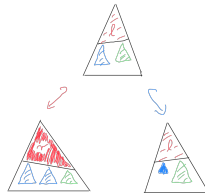


- just do the steps in reverse order

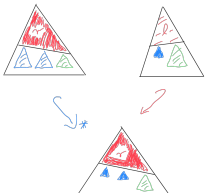


Proof of Local Confluence: Root- and Substitution-Step

- 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 $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \mathcal{D}$

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

$$f^{\mathcal{M}}(t_1, \dots, t_n) = s$$

where s is the **unique normal form** of $f(t_1, \dots, t_n)$, i.e., $f(t_1, \dots, t_n) \hookrightarrow^! s$

- 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: $c^{\mathcal{M}}(t_1, \dots, t_n) = c(t_1, \dots, t_n)$
 - defined symbols: $f^{\mathcal{M}}(t_1, \dots, t_n)$ is normal form of $f(t_1, \dots, t_n)$ 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 instantiating 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 computationally 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

$\text{head}(\text{Cons}(x, xs)) = x$	equation as before
$\text{head}(\text{Nil}) = \text{some element of } \mathcal{T}(\mathcal{C})_{\text{Nat}}$	new equation
- in this way, **head** becomes pattern complete and $\text{head}^{\mathcal{M}}$ is total
- “some element” really is an element of $\mathcal{T}(\mathcal{C})_{\text{Nat}}$, and **not a special error value** like \perp
- the added equation with “some element” is usually not revealed to the user, so the user cannot infer what number $\text{head}(\text{Nil})$ 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 **\perp -elements** are added to universe
- examples
 - $\mathcal{A}_{\text{Nat}} = \{\perp, \text{Zero}, \text{Succ}(\text{Zero}), \text{Succ}(\perp), \text{Succ}(\text{Succ}(\text{Zero})), \dots, \text{Succ}^{\infty}\}$
 - $\mathcal{A}_{\text{List}} = \{\perp, \text{Nil}, \text{Cons}(\text{Zero}, \text{Nil}), \text{Cons}(\perp, \text{Nil}), \text{Cons}(\perp, \perp), \dots\}$
- then semantics can be given to non-terminating computations
 - $\text{inf} = \text{Succ}(\text{inf})$ leads to $\text{inf}^{\mathcal{M}} = \text{Succ}^{\infty}$
 - $\text{undef} = \text{undef}$ leads to $\text{undef}^{\mathcal{M}} = \perp$
- problem: certain equalities don't hold w.r.t. domain theory semantics
 - assume usual definition of program for **minus**, then $\forall x. \text{minus}(x, x) = \text{Zero}$ is not true, consider $x = \text{inf}$ or $x = \text{undef}$
- since reasoning in domain theory is more complex, in this course we **restrict to terminating functional programs**
- even large proof assistants like Isabelle and Roqc usually restrict to terminating functions for that reason

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 \leftrightarrow
- definition of standard model
- upcoming
 - part 4: detect well-definedness, in particular termination
 - part 5: inference rules for standard model, equational reasoning engine