

Universiteit Utrecht

[Faculty of Science Information and Computing Sciences]

Compiler Construction

WWW: http://www.cs.uu.nl/wiki/Cco

Edition 2010/2011

Agenda

2

Type inference

Untyped lambda-calculus

Simply typed lambda-calculus

System F

Hindley-Milner typing

Algorithm W

Universiteit Utrecht

[Faculty of Science Information and Computing Sciences]

< □ > < 団 > < Ξ > < Ξ > < Ξ < つへの</p>

8. Type inference

8.1 Untyped lambda-calculus



[Faculty of Science Information and Computing Sciences] < □ ▶ < ⓓ ▶ < 힅 ▶ < 힅 ▶ < 힅 ▶ . 힅 ... ∽ < ↔



Universiteit Utrecht

Lambda-calculus

§**8.1**

Syntax

§**8.1**

- Church, Kleene (1930s).
- Formal system designed to investigate function definition, function application, and recursion.
- Idealised, minimalistic functional programming language.
- Only three language constructs: variables, lambda-abstraction, function application. (Other constructs can be encoded with these.)



5

[Faculty of Science Information and Computing Sciences]

Alpha-equivalence and beta-substitution §8.1

 λ is a *binder*: $\lambda x. t_1$ binds x in t_1 . Unbound variables are called *free*.

Alpha-equivalence: terms that only differ in the names of their bound variables are considered equal. For example: $\lambda x. x$ and $\lambda y. y$ are alpha-equivalent.

Alpha-conversion: consistently renaming bound variables while avoiding free variables from being *captured*. For example: $\lambda f. \lambda x. f x z$ can be alpha-converted into $\lambda f. \lambda y. f y z$, but not into $\lambda f. \lambda z. f z z$.

Beta-substitution: capture-avoiding substitution of free variables, performing alpha-conversion where necessary. For example: $[z \mapsto y](\lambda f.\lambda x.f x z) = \lambda f.\lambda x.f x y$ and $[z \mapsto x](\lambda f.\lambda y.f y x)$.

variables Var x \in \in \mathbf{Tm} terms $::= x \mid \lambda x. t_1 \mid t_1 t_2$ t Faculty of Science Universiteit Utrecht Information and Computing Sciences] 6 **Semantics** §8.1 Val \in values v $::= \lambda x. t$ v**Big-step operational semantics:** $t \Downarrow v$ evaluation - [*e-lam*] $\overline{\lambda x. t_1 \Downarrow \lambda x. t_1}$ $t_1 \Downarrow \lambda x. t_{11} \quad t_2 \Downarrow v_2 \quad [x \mapsto v_2]t_{11} \Downarrow v$ [e-app] $t_1 t_2 \Downarrow v$ For example: $(\lambda x. \lambda y. x) (\lambda x. x) (\lambda x. \lambda y. y) \Downarrow \lambda x. x.$ Faculty of Science

Universiteit Utrecht



8

Derived constructs

Additional language constructs—such as local definitions, natural numbers, boolean constants, conditionals, arithmetic and relational operators, and even recursion—can be introduced as mere syntactic sugar.

For example:

let $x = t_1$ in t_2 ni $=_{\mathsf{def}} (\lambda x. t_2) t_1$

In the sequel, we will just assume that some of these additional constructs are in fact added to the core calculus, so that we can use for example natural numbers in our example.



[Faculty of Science Information and Computing Sciences]

Simple types

9

§**8.2**

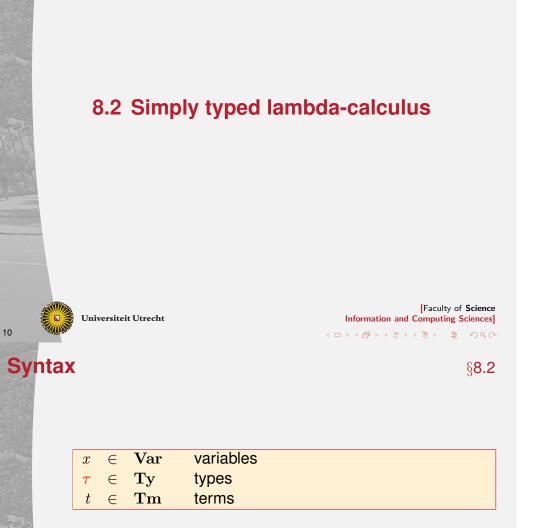
§8.1

To study typing for the lambda-calculus, we extend the syntax of lambda-terms with mandatory type annotations for lambda-abstractions.

For example:

 $\lambda x : Nat. x$

 $(\lambda f : Bool \rightarrow Nat. \lambda x : Bool. f x) (\lambda y : Bool. 42)$



 $\begin{array}{cccc} \tau & ::= & \tau_1 \to \tau_2 \\ t & ::= & x \mid \lambda x : \tau . t_1 \mid t_1 t_2 \end{array}$

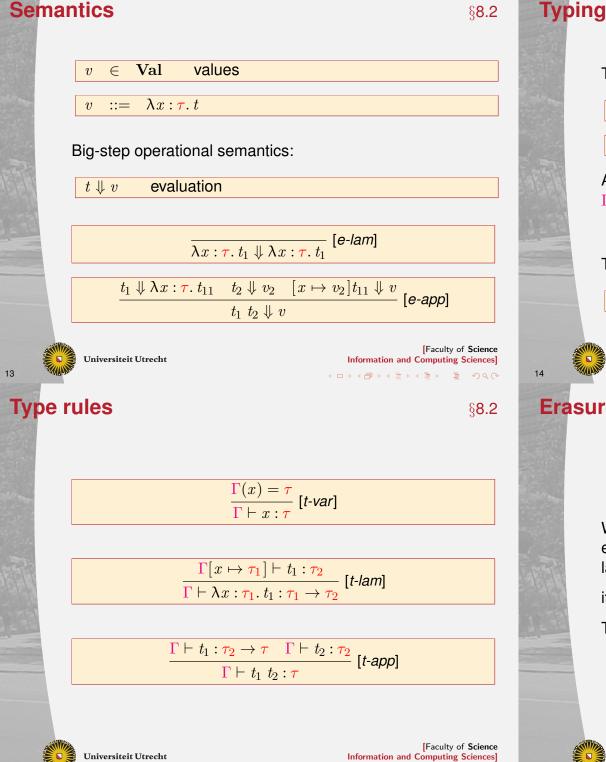
To render the sets of types inhabited, we add type constants such as *Nat* and *Bool*.



[Faculty of Science Information and Computing Sciences]



Semantics

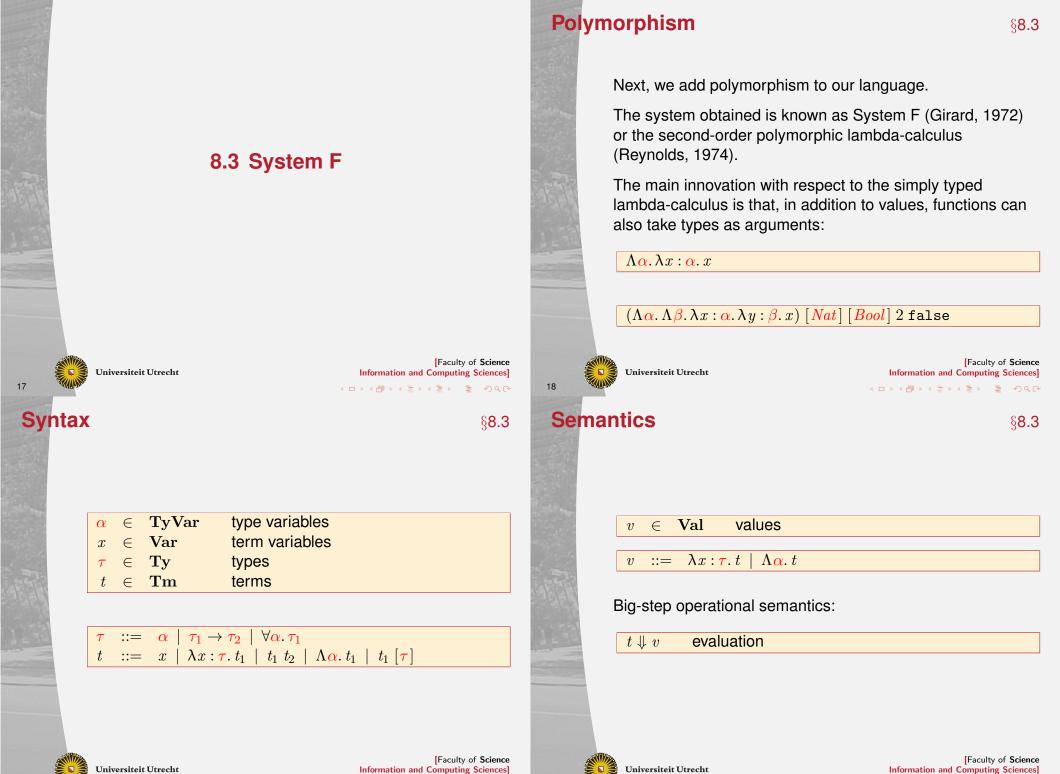


Type environments map from variables to types: \in TyEnv type environments $\Gamma_1[x \mapsto \tau]$::=As always, we write $\Gamma(x) = \tau$ if the rightmost binding for x in Γ maps x to τ . The judgements of the typing relation read $\Gamma \vdash t : \tau$ typing Faculty of Science **Universiteit Utrecht** Information and Computing Sciences] ▲□▶▲□▶▲≡▶▲≡▶ ≡ のへで **Erasure** §8.2 Writing |t| for the untyped lambda-term obtained from erasing all type annotations from the simply typed lambda-term t, we have: if $t \Downarrow v$, then $|t| \Downarrow |v|$. That is, types play no rôle at run-time.

Universiteit Utrecht

16

§8.2



20

◆□▶◆□▶◆三▶◆三▶ 三三 のへで

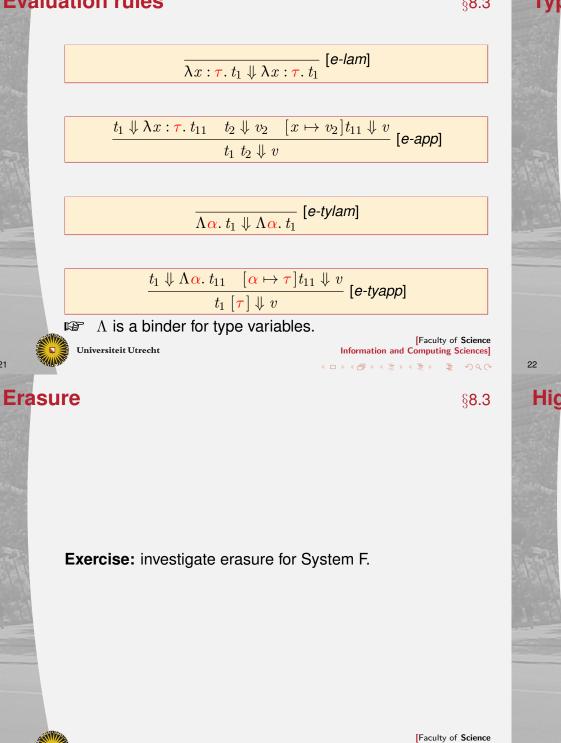
Evaluation rules

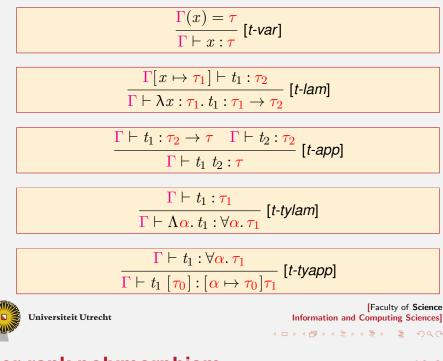
21

23

§8.3

Type rules





Higher-rank polymorphism

§8.3

§8.3

Note: functions can take polymorphic functions as arguments.

For example:

 $\lambda f: \forall \alpha. \alpha \rightarrow Nat. f [Nat] 2 + f [Bool]$ false

This function takes a "normal" polymorphic function (i.e, of rank 1) as argument and so it has itself a rank-2 type. Its type reads $(\forall \alpha. \alpha \rightarrow Nat) \rightarrow Nat$. In general, if a function takes a function with a rank-*n* type as argument, it has itself a rank-(n + 1) type.

Universiteit Utrecht

Information and Computing Sciences] ◆□▶◆□▶◆三▶◆三▶ 三三 のへで



24

Faculty of Science

8.4 Hindley-Milner typing

Universiteit Utrecht

25

27

[Faculty of Science Information and Computing Sciences]

The Hindley-Milner system

The Hindley-Milner (Hindley, 1969; Milner, 1978) type system is a compromise between the full power of System F and the desire to leave out type annotations.

Hindley-Milner typing comes with two crucial restrictions:

- 1. All types are of at most rank 1, i.e., functions cannot take polymorphic functions as arguments.
- 2. Functions can only have a polymorphic type if they are directly bound in a local definition (let-polymorphism).

The resulting type system is at the heart of languages like Haskell and ML and allows that for each well-typed term a so-called principal (i.e., most polymorphic) type can be inferred.

Type inference

For both the simply typed lambda-calculus and System F, implementing a type checker is straightforward. (Exercise: ...)

Although programming languages based on System F are very powerful, writing type annotation on every function parameter is very tedious—especially if types become more involved due to polymorphism.

So, the question is: can we derive an algorithm that takes an erased System-F term as argument and that *infers* all missing type annotations? That way, we can have the full power for System F, without the burden of having to write possibly complex or otherwise tiresome type annotations.

The anwer is **no**, type inference for System F is undecidable (Wells, 1994).

[Faculty of Science Information and Computing Sciences] 데마아데카에클아에클아이클아 클 수 있으면

Syntax

26

28

§8.4

 $\begin{array}{rcccc} x & \in & \mathbf{Var} & \mbox{term variables} \\ t & \in & \mathbf{Tm} & \mbox{terms} \end{array}$

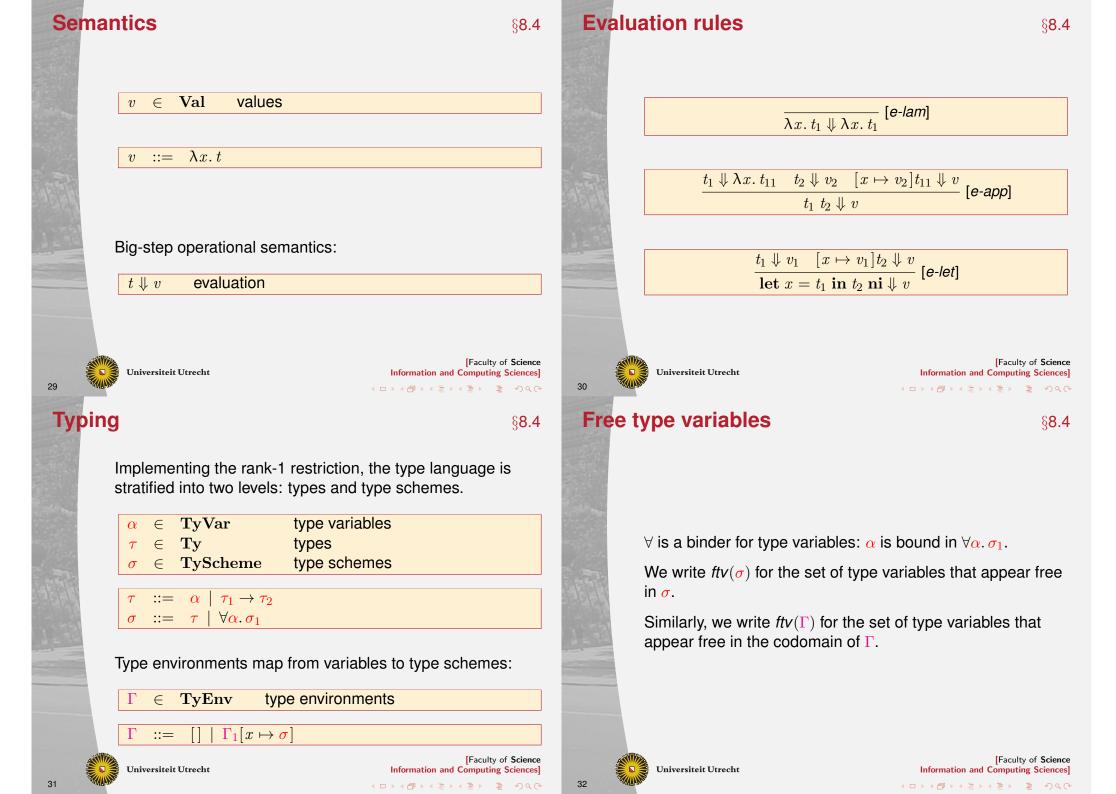
t ::= $x \mid \lambda x. t_1 \mid t_1 t_2 \mid$ let $x = t_1$ in t_2 ni

Local definitions play a crucial rôle in typing now and so, rather than syntactic sugar, they form a true language construct now.

Universiteit Utrecht

§8.4

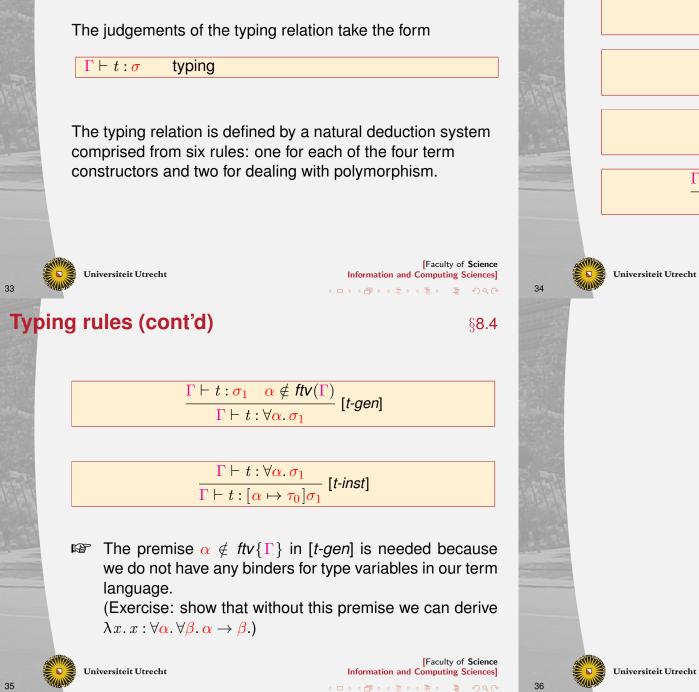
§8.4



Typing judgements

§8.4

Typing rules



 $\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \text{ [t-var]}$ $\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Gamma \vdash \lambda x. t_1 : \tau_1 \to \tau_2} \text{ [t-lam]}$ $\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 \ t_2 : \tau} \ [t\text{-app}]$ $\frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1] \vdash t_2 : \tau}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \text{ ni } : \tau} \text{ [t-let]}$ Faculty of Science Universiteit Utrecht Information and Computing Sciences] 8.5 Algorithm W

Faculty of Science

Algorithm W

§**8.5**

Algorithm W (Damas and Milner, 1982) establishes a procedure for obtaining a principal type, for each well-typed term in the Hindley-Milner system.

Intuitively, a principal type is the most polymorphic type that can be assigned to a given term.



[Faculty of Science Information and Computing Sciences]

Strategy

37

Algorithm W proceeds by initially "guessing" a fresh type variable for every parameter type and by incrementally refining theses guesses as more information on the use of parameters becomes available.

Algorithm W uses a syntax-directed variation of the Hindley-Milner type rules in which generalisation only occurs at let-bindings and instantiation only occurs at the use-sites of variables.

Syntax-directed: for every term, at most one rule applies.

- We have to somehow "guess", for every lambda-abstraction, what the type of its formal parameter is.
- The rules [t-gen] and [t-inst] can be applied to terms of any form. We have to decide when to apply them.

Universiteit Utrecht

38

40

§8.5

Challenges

[Faculty of Science Information and Computing Sciences]

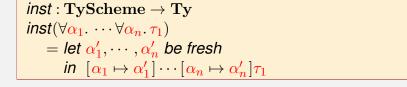
Generalisation and instantiation

§**8.5**

§8.5

The syntax-directed type rules are defined in terms of metaoperations *gen* and *inst*:

 $\begin{array}{l} gen_{\cdot}:\mathbf{TyEnv} \to \mathbf{Ty} \to \mathbf{TyScheme} \\ gen_{\Gamma}(\tau) = \\ let \left\{ \alpha_{1}, \ldots, \alpha_{n} \right\} = \mathit{ftv}(\tau) \backslash \mathit{ftv}(\Gamma) \\ in \ \forall \alpha_{1}, \ldots \forall \alpha_{n}, \tau \end{array}$



Universiteit Utrecht

Syntax-directed type rules

§**8.5**

$\frac{\Gamma(x) = \sigma_0}{\Gamma \vdash x : inst(\sigma_0)}$	[t-var]

 $\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Gamma \vdash \lambda x. \ t_1 : \tau_1 \to \tau_2} \ [t-lam]$

 $\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 \ t_2 : \tau} \ [t\text{-app}]$

 $\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma[x \mapsto gen_{\Gamma}(\tau_1)] \vdash t_2 : \tau}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \text{ ni} : \tau} \text{ [t-let]}$

Image: Second state of the state of th

Applying type substitutions

§**8.5**

42

44

(ロ)

Un

Applying a type substitution to a type scheme:

```
\begin{aligned} \mathbf{id}\sigma &= \sigma \\ [\alpha \mapsto \tau_0]\alpha &= \tau_0 \\ [\alpha \mapsto \tau_0]\alpha_0 &= \alpha_0 & \text{if } \alpha \not\equiv \alpha_0 \\ [\alpha \mapsto \tau_0](\tau_1 \to \tau_2) &= [\alpha \mapsto \tau_0]\tau_1 \to [\alpha \mapsto \tau_0]\tau_2 \\ [\alpha \mapsto \tau_0](\forall \alpha. \sigma_1) &= \forall \alpha. \sigma \\ [\alpha \mapsto \tau_0](\forall \alpha_0. \sigma_1) &= \forall \alpha_0. [\alpha \mapsto \tau_0]\sigma_1 & \text{if } \alpha \not\equiv \alpha_0 \\ (\theta_1 \circ \theta_2)\sigma &= \theta_1 \theta_2 \sigma \end{aligned}
```

Applying a type substitution to a type environment:

 $\begin{array}{l} \theta[] &= [] \\ \theta(\Gamma_1[x \mapsto \sigma]) = \theta\Gamma_1[x \mapsto \theta\sigma] \end{array}$ [Faculty of Sciences] $\begin{array}{l} \text{Universiteit Utrecht} \end{array}$

Type substitutions

Algorithm W makes use of type substitutions:

	$\theta \in$	TySubst	type substitutions
	θ ::=	id $[\alpha \mapsto$	$[\tau] \mid \theta_1 \circ \theta_2$
	Universitei	t Utrecht	[Faculty of Sciences] Information and Computing Sciences
24 A AM			< □ > < 問 > < 言 > < 言 > < 言 > < 言 > くろ < C
ifica	tion		§ 8. 5

Algorithm W makes use of Robinson's unification algorithm (1965).

 $\mathcal{U}: \mathbf{Ty} imes \mathbf{Ty} o \mathbf{TySubst}$

 \mathcal{U} provides a partial function that, for any two types τ_1 and τ_2 , constructs a *most general unifier*, i.e., a type substitution θ , such that $\theta \tau_1 = \theta \tau_2$ and, for all θ' with $\theta' \tau_1 = \theta' \tau_2$ there is a θ'' with $\theta' \approx \theta'' \circ \theta$. (Where $\theta_1 \approx \theta_2$ iff, $\theta_1 \sigma = \theta_2 \sigma$ for all σ .)

If τ_1 and τ_2 are not unifiable, \mathcal{U} fails.

For example:

```
\mathcal{U}(\alpha \to Bool \to \alpha , Nat \to \beta \to \gamma) \\ = [\gamma \mapsto Nat] \circ [\beta \mapsto Bool] \circ [\alpha \mapsto Nat]
```

§8.5

Unification (cont'd)

 $\mathcal{U}(\boldsymbol{\alpha},\boldsymbol{\alpha})$ = id $\begin{array}{ll} \mathcal{U}(\alpha_1, \tau_2) &= [\alpha_1 \mapsto \tau_2] & \text{ if } \alpha_1 \notin \textit{ftv}(\tau_2) \\ \mathcal{U}(\tau_1, \alpha_2) &= [\alpha_2 \mapsto \tau_1] & \text{ if } \alpha_2 \notin \textit{ftv}(\tau_1) \end{array}$ $\mathcal{U}(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}) = let \ \theta_1 = \mathcal{U}(\tau_{11}, \tau_{21})$ $\theta_2 = \mathcal{U}(\theta_1 \tau_{12}, \theta_1 \tau_{22})$ in $\theta_2 \circ \theta_1$ $\mathcal{U}(\tau_1, \tau_2)$ = fail in all other cases

 \mathfrak{W} The side conditions $\alpha_1 \notin ftv(\tau_2)$ and $\alpha_2 \notin ftv(\tau_2)$ are known as the "occurs check" and prevent the construction of infinite types.

AWYER	Universiteit Utrecht			
-------	----------------------	--	--	--

Algorithm W: variables

 $\mathcal{W}(\Gamma, x) =$ if $x \in dom(\Gamma)$ then (inst $(\Gamma(x))$, id) fail if

§8.5

Faculty of Science

§8.5

Information and Computing Sciences]

▲□▶▲圖▶▲≣▶▲≣▶ ≣ の�?

 $\mathcal{W}: \mathbf{TyEnv} \times \mathbf{Tm} \rightarrow \mathbf{Ty} \times \mathbf{TySubst}$

Algorithm W takes a type environment Γ and a term t, and produces, if t is well-typed in Γ , a type τ and a type substitution θ , such that $\theta \Gamma \vdash t : gen_{\theta \Gamma}(\tau)$. Moreover, $gen_{\theta\Gamma}(\tau)$ is then a principal type for t in $\theta\Gamma$.

If t is ill-typed in Γ , Algorithm W fails.



 $\mathcal{W}(\Gamma, \lambda x. t_1) =$ let α_1 be fresh $(\tau_2, \theta_1) = \mathcal{W}(\Gamma[x \mapsto \alpha_1], t_1)$ in $(\theta_1 \alpha_1 \rightarrow \tau_2, \theta_1)$



Faculty of Science Information and Computing Sciences]



48

Faculty of Science Information and Computing Sciences]

45

§8.5

