Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

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

Edition 2010/2011

Agenda

Type inference

- Untyped lambda-calculus
- Simply typed lambda-calculus
- System F
- Hindley-Milner typing
- Algorithm W

8. Type inference

8.1 Untyped lambda-calculus

Lambda-calculus

- 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.)

Syntax

x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$t \quad ::= \quad x \mid \lambda x.\, t_1 \mid t_1\ t_2$$

Alpha-equivalence and beta-substitution

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 $[z \mapsto x](\lambda f.\, \lambda y.\, f\ y\ x)$.

Semantics

v	$\in$	$\mathbf{Val}$	values

$$v \quad ::= \quad \lambda x.\, t$$

Big-step operational semantics:

$t \Downarrow v$ evaluation

$$\frac{}{\lambda x.\, t_1 \Downarrow \lambda x.\, t_1}\ [\textit{e-lam}]$$

$$\frac{t_1 \Downarrow \lambda x.\, t_{11} \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v}\ [\textit{e-app}]$$

For example: $(\lambda x.\, \lambda y.\, x)\ (\lambda x.\, x)\ (\lambda x.\, \lambda y.\, y) \Downarrow \lambda x.\, x$.

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:

$$\textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ ni} \quad =_{\text{def}} \quad (\lambda x.\, t_2)\ t_1$$

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.

8.2 Simply typed lambda-calculus

Simple types

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

For example:

$$\lambda x : \mathit{Nat}.\, x$$

$$(\lambda f : \mathit{Bool} \to \mathit{Nat}.\, \lambda x : \mathit{Bool}.\, f\ x)\ (\lambda y : \mathit{Bool}.\, 42)$$

Syntax

x	$\in$	$\mathbf{Var}$	variables
τ	$\in$	$\mathbf{Ty}$	types
t	$\in$	$\mathbf{Tm}$	terms

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

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

Semantics

$v \in$ **Val** values

$v ::= \lambda x : \tau .\, t$

Big-step operational semantics:

$t \Downarrow v$ evaluation

$$\frac{}{\lambda x : \tau .\, t_1 \Downarrow \lambda x : \tau .\, t_1}\ [\textit{e-lam}]$$

$$\frac{t_1 \Downarrow \lambda x : \tau .\, t_{11} \quad t_2 \Downarrow v_2 \quad [x \mapsto v_2]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v}\ [\textit{e-app}]$$

Typing

Type environments map from variables to types:

$\Gamma \in$ **TyEnv** type environments

$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \tau]$

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

Type rules

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

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

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

Erasure

Writing $\lfloor t \rfloor$ 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 $\lfloor t \rfloor \Downarrow \lfloor v \rfloor$.

That is, types play no rôle at run-time.

8.3 System F

Polymorphism

Next, we add polymorphism to our language.

The system obtained is known as System F (Girard, 1972) or the second-order polymorphic lambda-calculus (Reynolds, 1974).

The main innovation with respect to the simply typed lambda-calculus is that, in addition to values, functions can also take types as arguments:

$$\Lambda\alpha.\,\lambda x:\alpha.\,x$$

$$(\Lambda\alpha.\,\Lambda\beta.\,\lambda x:\alpha.\,\lambda y:\beta.\,x)\ [Nat]\ [Bool]\ 2\ \mathtt{false}$$

Syntax

α	$\in$	$\mathbf{TyVar}$	type variables
x	$\in$	$\mathbf{Var}$	term variables
τ	$\in$	$\mathbf{Ty}$	types
t	$\in$	$\mathbf{Tm}$	terms

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau_1 \\ t & ::= & x \mid \lambda x:\tau.\,t_1 \mid t_1\ t_2 \mid \Lambda\alpha.\,t_1 \mid t_1\ [\tau] \end{array}$$

Semantics

$v \in \mathbf{Val}$ values

$$v ::= \lambda x:\tau.\,t \mid \Lambda\alpha.\,t$$

Big-step operational semantics:

$t \Downarrow v$ evaluation

Evaluation rules

$$\frac{}{\lambda x : \tau.\, t_1 \Downarrow \lambda x : \tau.\, t_1}\ [\textit{e-lam}]$$

$$\frac{t_1 \Downarrow \lambda x : \tau.\, t_{11} \quad t_2 \Downarrow v_2 \quad [x \mapsto v_2]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v}\ [\textit{e-app}]$$

$$\frac{}{\Lambda\alpha.\, t_1 \Downarrow \Lambda\alpha.\, t_1}\ [\textit{e-tylam}]$$

$$\frac{t_1 \Downarrow \Lambda\alpha.\, t_{11} \quad [\alpha \mapsto \tau]t_{11} \Downarrow v}{t_1\ [\tau] \Downarrow v}\ [\textit{e-tyapp}]$$

☞ Λ is a binder for type variables.

Type rules

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

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

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

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash \Lambda\alpha.\, t_1 : \forall\alpha.\, \tau_1}\ [\textit{t-tylam}]$$

$$\frac{\Gamma \vdash t_1 : \forall\alpha.\, \tau_1}{\Gamma \vdash t_1\ [\tau_0] : [\alpha \mapsto \tau_0]\tau_1}\ [\textit{t-tyapp}]$$

Erasure

Exercise: investigate erasure for System F.

Higher-rank polymorphism

Note: functions can take polymorphic functions as arguments.

For example:

$$\lambda f : \forall\alpha.\, \alpha \to \mathit{Nat}.\, f\ [\mathit{Nat}]\ 2 + f\ [\mathit{Bool}]\ \texttt{false}$$

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 $(\forall\alpha.\, \alpha \to \mathit{Nat}) \to \mathit{Nat}$. In general, if a function takes a function with a rank-n type as argument, it has itself a rank-$(n+1)$ type.

8.4 Hindley-Milner typing

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).

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.

Syntax

x	$\in$	$\mathbf{Var}$	term variables
t	$\in$	$\mathbf{Tm}$	terms

$$t ::= x \mid \lambda x.\, t_1 \mid t_1\ t_2 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}$$

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

Semantics

v	$\in$	$\mathbf{Val}$	values

$v \;::=\; \lambda x.\, t$

Big-step operational semantics:

$t \Downarrow v$	evaluation

Evaluation rules

$$\frac{}{\lambda x.\, t_1 \Downarrow \lambda x.\, t_1}\ [\textit{e-lam}]$$

$$\frac{t_1 \Downarrow \lambda x.\, t_{11} \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v}\ [\textit{e-app}]$$

$$\frac{t_1 \Downarrow v_1 \qquad [x \mapsto v_1]t_2 \Downarrow v}{\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \Downarrow v}\ [\textit{e-let}]$$

Typing

Implementing the rank-1 restriction, the type language is stratified into two levels: types and type schemes.

α	$\in$	$\mathbf{TyVar}$	type variables
τ	$\in$	$\mathbf{Ty}$	types
σ	$\in$	$\mathbf{TyScheme}$	type schemes

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall \alpha.\, \sigma_1 \end{array}$$

Type environments map from variables to type schemes:

Γ	$\in$	$\mathbf{TyEnv}$	type environments

$\Gamma \;::=\; [\,] \mid \Gamma_1[x \mapsto \sigma]$

Free type variables

$\forall$ is a binder for type variables: α is bound in $\forall \alpha.\, \sigma_1$.

We write *ftv*(σ) for the set of type variables that appear free in σ.

Similarly, we write *ftv*(Γ) for the set of type variables that appear free in the codomain of Γ.

Typing judgements

The judgements of the typing relation take the form

$\Gamma \vdash t : \sigma$	typing

The typing relation is defined by a natural deduction system comprised from six rules: one for each of the four term constructors and two for dealing with polymorphism.

Typing rules

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

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

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

$$\frac{\Gamma \vdash t_1 : \sigma_1 \qquad \Gamma[x \mapsto \sigma_1] \vdash t_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} : \tau}\ [\textit{t-let}]$$

Typing rules (cont'd)

$$\frac{\Gamma \vdash t : \sigma_1 \qquad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash t : \forall \alpha.\, \sigma_1}\ [\textit{t-gen}]$$

$$\frac{\Gamma \vdash t : \forall \alpha.\, \sigma_1}{\Gamma \vdash t : [\alpha \mapsto \tau_0]\sigma_1}\ [\textit{t-inst}]$$

☞ The premise $\alpha \notin \mathit{ftv}\{\Gamma\}$ in [*t-gen*] is needed because we do not have any binders for type variables in our term language.
(Exercise: show that without this premise we can derive $\lambda x.\, x : \forall \alpha.\forall \beta.\, \alpha \to \beta$.)

8.5 Algorithm W

Algorithm W

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.

Challenges

- 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.

Strategy

- 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.

Generalisation and instantiation

The syntax-directed type rules are defined in terms of metaoperations $\mathit{gen}_{\cdot}$ and inst:

$$\begin{array}{l}
\mathit{gen}_{\cdot} : \mathbf{TyEnv} \to \mathbf{Ty} \to \mathbf{TyScheme} \\
\mathit{gen}_{\Gamma}(\tau) = \\
\quad \mathit{let}\ \{\alpha_1, .., \alpha_n\} = \mathit{ftv}(\tau) \backslash \mathit{ftv}(\Gamma) \\
\quad \mathit{in}\ \ \forall \alpha_1. \cdots \forall \alpha_n. \tau
\end{array}$$

$$\begin{array}{l}
\mathit{inst} : \mathbf{TyScheme} \to \mathbf{Ty} \\
\mathit{inst}(\forall \alpha_1. \cdots \forall \alpha_n. \tau_1) \\
\quad = \mathit{let}\ \alpha'_1, \cdots, \alpha'_n\ \mathit{be\ fresh} \\
\qquad \mathit{in}\ \ [\alpha_1 \mapsto \alpha'_1] \cdots [\alpha_n \mapsto \alpha'_n] \tau_1
\end{array}$$

Syntax-directed type rules

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

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

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

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{gen}_\Gamma(\tau_1)] \vdash t_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} : \tau} \;[\textit{t-let}]$$

☞ All judgements have the form $\Gamma \vdash t : \tau$ (rather than $\Gamma \vdash t : \sigma$).

Type substitutions

Algorithm W makes use of type substitutions:

$\theta \in \mathbf{TySubst}$ type substitutions

$\theta ::= \mathit{id} \mid [\alpha \mapsto \tau] \mid \theta_1 \circ \theta_2$

Applying type substitutions

Applying a type substitution to a type scheme:

$$\begin{array}{lll}
\mathit{id}\,\sigma & = \sigma & \\
[\alpha \mapsto \tau_0]\alpha & = \tau_0 & \\
[\alpha \mapsto \tau_0]\alpha_0 & = \alpha_0 & \text{if } \alpha \not\equiv \alpha_0 \\
[\alpha \mapsto \tau_0](\tau_1 \to \tau_2) & = [\alpha \mapsto \tau_0]\tau_1 \to [\alpha \mapsto \tau_0]\tau_2 & \\
[\alpha \mapsto \tau_0](\forall \alpha.\, \sigma_1) & = \forall \alpha.\, \sigma & \\
[\alpha \mapsto \tau_0](\forall \alpha_0.\, \sigma_1) & = \forall \alpha_0.\, [\alpha \mapsto \tau_0]\sigma_1 & \text{if } \alpha \not\equiv \alpha_0 \\
(\theta_1 \circ \theta_2)\sigma & = \theta_1\theta_2\sigma &
\end{array}$$

Applying a type substitution to a type environment:

$$\begin{array}{ll}
\theta[\,] & = [\,] \\
\theta(\Gamma_1[x \mapsto \sigma]) & = \theta\Gamma_1[x \mapsto \theta\sigma]
\end{array}$$

Unification

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

$\mathcal{U} : \mathbf{Ty} \times \mathbf{Ty} \rightharpoonup \mathbf{TySubst}$

$\mathcal{U}$ 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:

$$\begin{array}{l}
\mathcal{U}(\alpha \to \mathit{Bool} \to \alpha\ ,\ \mathit{Nat} \to \beta \to \gamma) \\
\quad = [\gamma \mapsto \mathit{Nat}] \circ [\beta \mapsto \mathit{Bool}] \circ [\alpha \mapsto \mathit{Nat}]
\end{array}$$

Unification (cont'd)

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

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

Algorithm W

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

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 $\theta\Gamma \vdash t : \mathit{gen}_{\theta\Gamma}(\tau)$. Moreover, $\mathit{gen}_{\theta\Gamma}(\tau)$ is then a principal type for t in $\theta\Gamma$.

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

Algorithm W: variables

$$
\begin{array}{l}
\mathcal{W}(\Gamma\ ,\ x) = \\
\quad \mathit{if} \quad\ x \in \mathit{dom}(\Gamma) \\
\quad \mathit{then}\ (\mathit{inst}\ (\Gamma(x))\ ,\ \mathit{id}) \\
\quad \mathit{if} \quad\ \mathit{fail}
\end{array}
$$

Algorithm W: lambda-abstractions

$$
\begin{array}{l}
\mathcal{W}(\Gamma\ ,\ \lambda x.\, t_1) = \\
\quad \mathit{let}\ \alpha_1\ \mathit{be\ fresh} \\
\qquad (\tau_2\ ,\ \theta_1) = \mathcal{W}(\Gamma[x \mapsto \alpha_1]\ ,\ t_1) \\
\quad \mathit{in}\ \ (\theta_1\alpha_1 \to \tau_2, \theta_1)
\end{array}
$$

Algorithm W: applications

$$
\begin{aligned}
&\mathcal{W}(\Gamma\ ,\ t_1\ t_2) = \\
&\quad \textit{let } \alpha \textit{ be fresh} \\
&\qquad (\tau_1\ ,\ \theta_1) = \mathcal{W}(\Gamma\ ,\ t_1) \\
&\qquad (\tau_2\ ,\ \theta_2) = \mathcal{W}(\theta_1\Gamma\ ,\ t_2) \\
&\qquad \theta_3 \qquad\quad\ = \mathcal{U}(\theta_2\ \tau_1\ ,\ \tau_2 \to \alpha) \\
&\quad \textit{in}\ \ (\theta_3\alpha\ ,\ \theta_3 \circ \theta_2 \circ \theta_1)
\end{aligned}
$$

Algorithm W: local definitions

$$
\begin{aligned}
&\mathcal{W}(\Gamma\ ,\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}) = \\
&\quad \textit{let}\ (\tau_1\ ,\ \theta_1) = \mathcal{W}(\Gamma\ ,\ t_1) \\
&\qquad (\tau\ ,\ \theta_2)\ \ = \mathcal{W}(\theta_1\Gamma[x \mapsto \textit{gen}_{\theta_1\Gamma}(\tau_1)]\ ,\ t_2) \\
&\quad \textit{in}\ \ (\tau\ ,\ \theta_2 \circ \theta_1)
\end{aligned}
$$