Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Type equivalence, subtyping, and type transformations in object-oriented databases

C.J.E. Thieme, A.P.J.M. Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9451 1994

Type Equivalence, Subtyping, and Type Transformations in Object-Oriented Databases

Christiaan Thieme and Arno Siebes
{ct,arno}@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

In this report, a number of completeness results are given that are useful for database integration on the schema level and the instance level. Type equivalence and subtyping are proven sound and complete w.r.t. a model-theoretic semantics. Furthermore, a set of type transformations is introduced that is proven sound and complete w.r.t. data capacity. These completeness results imply that if database schemas are integrated using type equivalence, subtyping, and the set of type transformations, then their instances can be integrated as well.

CR Subject Classification (1991): E.1: Data structures, H.2.5: Heterogeneous databases (data translation).
Keywords and Phrases: Type equivalence, subtyping, type transformations, object-oriented databases.
Note: This research is partly funded by The Netherlands Organization for Scientific Research (NWO) through NFI-grant NF74.

Contents

1 Introduction

Database integration has been identified as one of the major challenges in responding to enterprises' information requirements [9]. In this report, we will prove a number of completeness results that are useful for semantic database integration, where databases are integrated using structure and behaviour [10, 11].

As a basis for the completeness results, we need a formalisation of both database schemas and database instances. A database schema in an object-oriented database language (e.g., O_2 [8] or TM [3]) is a class hierarchy: a set of classes related by a subclass relation. A class has a set of attributes, a set of constraints, and a set of methods. A database instance is a set of class extensions, where a class extension is a set of objects.

There are several options to formalise class hierarchies and class extensions. One option is to assign a set of simple instances (e.g., names) to a class and to formalise every attribute as a variable function from the set of instances to the corresponding codomain, every constraint as a fixed function from the powerset of the set of instances to the domain of the booleans, and every method as a fixed function from the set of instances and the domains of the parameters to the corresponding codomain. An instance can be queried and manipulated by applying the formalised attributes, constraints, and methods.

We have chosen another option, similar to the approach of [4], where the attributes of a class and a special identifier attribute are aggregated into a record type (the underlying type of the class). The set of possible instances of a class is the set of instances of its underlying type. Furthermore, every constraint is formalised as a logical formula, the interpretation of which is a function from the powerset of the set of instances to the domain of the booleans, and every method is formalised as a lambda expression, the interpretation of which is a function from the set of instances and the domains of the parameters to the corresponding codomain.

Our formalisation differs from the approach of [4] as follows. In [4], identifier attributes are used both to discriminate between different instances and to cope with recursive class definitions. In our formalisation, identifier attributes are used to discriminate between different instances and recursive types are used to cope with recursive class definitions. In fact, our formalisation resembles the approach of [6], where cyclic graphs are used to model database schemas and database instances, except that we use special identifier attributes. The resemblance stems from the fact that types and instances of types can be interpreted as cyclic graphs or, as will be shown, as infinite trees.

In this report, we focus on types and instances of types, and leave functions for what they are. In Section 2, we define the syntax and semantics of types. In Section 3, we introduce structural type equivalence, similar to type equivalence in Algol68 ([7]), where types are represented by infinite trees. Furthermore, we define extensional type equivalence, reducible type equivalence, and derivable type equivalence, which induces an algorithm, similar to the algorithm in [7]. In fact, our approach resembles the approach of [2], where structural type equivalence resembles $=_T$, extensional type equivalence resembles $=_M$, reducible type equivalence resembles $=_R$, and derivable type equivalence $=_A$. However, there is an important difference. Since we have no functional types, but only basic, set, and recursive record types, the models in our approach (viz., the extensions) are simpler. As a consequence, we have a stronger result: derivable equivalence is sound and complete w.r.t. both structural and extensional type equivalence. In Section 4, we introduce derivable subtyping using a set of subtyping rules, structural subtyping using trees, and extensional subtyping using extensions. These subtyping relations resemble $<_A$, $<_T$, and $<_M$ from [2], respectively. Again, we have a stronger result: derivable subtyping is sound and complete w.r.t. both structural and extensional subtyping. In Section 5, we introduce type transformations: renaming operations and aggregation operations. We show that this set of type transformations is sound and complete w.r.t. data capacity. Finally, in Section 6, we summarise and analyse the results.

2 Types and instances

In this section, we introduce types and μ-complete types, type equality, terms (or instances) of types, an equivalence relation on terms, and a subterm relation on terms.

The set of types consists of type variables, basic types, set types, record types, and recursive record types. The set of μ-complete types is the same as the set of types, except that every record type is preceded by an occurrence of recursion operator μ. In fact, μ-completeness is only a technical restriction, since every general type can be rewritten as a μ-complete type. The restriction to μ-complete types just simplifies the proofs of a number of theorems. The syntax of types is given by the following definition.

Definition 1. First, we postulate a set of type variables $\mathit{TypeVar}$, a set of basic types $\mathit{BTypes} = \{\mathrm{oid}, \mathrm{integer}, \mathrm{rational}, \mathrm{string}\}$, and a set of labels $\mathcal{L}$. The set of types, denoted by Types, is inductively defined by:

1. if $t \in \mathit{TypeVar}$, then $t \in \mathit{Types}$
2. if $B \in \mathit{BTypes}$, then $B \in \mathit{Types}$
3. if $v \in \mathit{Types}$, then $\{v\} \in \mathit{Types}$
4. if $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $\{v_1, \cdots, v_n\} \subseteq \mathit{Types}$, then $< l_1 : v_1, \cdots, l_n : v_n > \in \mathit{Types}$
5. if $t \in \mathit{TypeVar}$ and $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $V = \{v_1, \cdots, v_n\} \subseteq \mathit{Types}$ and $\forall \tau \in V[t \notin \mathit{bvars}(\tau)]$, then $\mu t. < l_1 : v_1, \cdots, l_n : v_n > \in \mathit{Types}$,

where $\mathit{bvars}(\tau)$ is the set of bound type variables in τ:

$\mathit{bvars}(t) = \emptyset$ if $t \in \mathit{TypeVar}$
$\mathit{bvars}(B) = \emptyset$ if $B \in \mathit{BTypes}$
$\mathit{bvars}(\{v\}) = \mathit{bvars}(v)$
$\mathit{bvars}(< l_1 : v_1, \cdots, l_n : v_n >) = \mathit{bvars}(v_1) \cup \cdots \cup \mathit{bvars}(v_n)$
$\mathit{bvars}(\mu t.\alpha) = \mathit{bvars}(\alpha) \cup \{t\}$.

Furthermore, the set of closed types, denoted by CTypes, is defined as follows:

$\mathit{CTypes} = \{\tau \in \mathit{Types} \mid \mathit{fvars}(\tau) = \emptyset\}$,

where $\mathit{fvars}(\tau)$ is the set of free type variables in τ:

$\mathit{fvars}(t) = \{t\}$ if $t \in \mathit{TypeVar}$
$\mathit{fvars}(B) = \emptyset$ if $B \in \mathit{BTypes}$
$\mathit{fvars}(\{v\}) = \mathit{fvars}(v)$
$\mathit{fvars}(< l_1 : v_1, \cdots, l_n : v_n >) = \mathit{fvars}(v_1) \cup \cdots \cup \mathit{fvars}(v_n)$
$\mathit{fvars}(\mu t.\alpha) = \mathit{fvars}(\alpha) - \{t\}$.

The set of μ-complete types, denoted by $\mu Types$, is inductively defined by:

1. if $t \in TypeVar$, then $t \in \mu Types$
2. if $B \in BTypes$, then $B \in \mu Types$
3. if $\upsilon \in \mu Types$, then $\{\upsilon\} \in \mu Types$
4. if $t \in TypeVar$ and $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $V = \{\upsilon_1, \cdots, \upsilon_n\} \subseteq \mu Types$ and $\forall \tau \in V [t \notin bvars(\tau)]$, then $\mu t. < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n > \in \mu Types$.

Finally, the set of closed μ-complete types, denoted by $C\mu Types$, consists of the μ-complete types that have no free type variables:

$$C\mu Types = \{\tau \in \mu Types \mid fvars(\tau) = \emptyset\}.$$

□

Type equality is defined as syntactical equality, modulo addition of dummy type variables and permutation of fields in record types.

Definition 2. Let τ_1 and τ_2 be types. Equality of τ_1 and τ_2, denoted by $\tau_1 = \tau_2$, is inductively defined as follows:

1. $t = t$	if $t \in TypeVar$
2. $B = B$	if $B \in BTypes$
3. $\{\upsilon\} = \{\upsilon'\}$	if $\upsilon = \upsilon'$
4. $< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n > = < l'_1 : \upsilon'_1, \cdots, l'_n : \upsilon'_n >$	if $\forall i \in I \; \exists j \in I \; [l_i = l'_j \wedge \upsilon_i = \upsilon'_j]$
5. $< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n > = \; \mu t. < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >$	if $\forall i \in I \; [t \notin fvars(\upsilon_i)]$
6. $\mu t. < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n > = < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >$	if $\forall i \in I \; [t \notin fvars(\upsilon_i)]$
7. $\mu t.\alpha = \mu t.\alpha'$	if $\alpha = \alpha'$.

Using rule 5 and 6, we can rewrite every type as a μ-complete type. □

The semantics of types can be defined in terms of trees or extensions. The tree of a type represents the structure of the type.

Definition 3. Let τ be a type. The tree representing τ is defined as $struc(\tau, \emptyset)$, where $struc(\tau, \Gamma)$ is defined as follows:

$struc(t, \Gamma) =$

t

if $t \in TypeVar$ and $\eta_\Gamma(t) = \perp$,

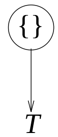
$struc(t, \Gamma) = struc(\mu t.\alpha, \Gamma)$
if $t \in TypeVar$ and $\eta_\Gamma(t) = \mu t.\alpha$,

$struc(B, \Gamma) =$

B

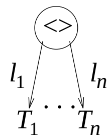
if $B \in BTypes$,

$struc(\{\upsilon\}, \Gamma) =$



where $T = struc(\upsilon, \Gamma)$,

$struc(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >, \Gamma) =$



where $T_i = struc(\upsilon_i, \Gamma)$,

$struc(\mu t.\alpha, \Gamma) = struc(\alpha, \Gamma \cup \{\mu t.\alpha\})$,

where η_Γ is a partial function from *TypeVar* to *Types* induced by Γ; $\eta_\Gamma(t) = \tau$ if τ is a type in Γ and τ starts with μt (there is at most one such type) and $\eta_\Gamma(t) = \perp$ if there is no type in Γ that starts with μt. For convenience, we sometimes write $struc(\tau)$ instead of $struc(\tau, \emptyset)$. $\Box$

The extension of a type is the set of closed terms of which the structure corresponds to the structure of the type.

Definition 4. First, for every basic type B, we postulate a disjoint set of constants $Cons_B$, and, for every type variable t, we postulate a disjoint set of instance variables Var_t. The set of all constants, denoted by *Cons*, is given by:

$$Cons = \{Cons_B \mid B \in BTypes\}.$$

Furthermore, the set of all instance variables, denoted by *Var*, is given by:

$$Var = \{Var_t \mid t \in TypeVar\}.$$

Let τ be a type. The set of terms (or instances) of type τ, denoted by $terms(\tau)$, is defined as follows:

$$\begin{array}{l} terms(t) = Var_t \ \text{ if } t \in TypeVar, \\ terms(B) = Cons_B \ \text{ if } B \in BTypes, \\ terms(\{\upsilon\}) = \wp_{fin}(terms(\upsilon)), \\ terms(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = \\ \quad \{< l_1 = e_1, \cdots, l_n = e_n > \mid e_1 \in terms(\upsilon_1) \wedge \cdots \wedge e_n \in terms(\upsilon_n)\}, \\ terms(\mu t.\alpha) = \\ \quad terms(t) \ \cup \ \{\mu x.(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n]) \mid x \in Var_t \wedge e_0 \in terms(\alpha) \wedge n \in I\!N \wedge \\ \qquad\qquad\qquad \forall i \in \{1, \cdots, n\}[x_i \in Var_t \wedge e_i \in terms(\mu t.\alpha) \wedge x \notin BV(e_i)]\}, \end{array}$$

where $\wp_{fin}(V)$ is the set of all finite subsets of set V and $BV(e)$ is the set of bound variables in term e:

$$\begin{array}{l} BV(y) = \emptyset \ \text{ if } \ y \in Var, \\ BV(b) = \emptyset \ \text{ if } \ b \in Cons, \\ BV(\{e_1, \cdots, e_n\}) = BV(< l_1 = e_1, \cdots, l_n = e_n >) = BV(e_1) \cup \cdots \cup BV(e_n), \\ BV(\mu y.e) = BV(e) \cup \{y\}. \end{array}$$

The set of all terms, denoted by *Terms*, is given by:

$Terms = \{terms(\tau) \mid \tau \in Types\}$.

Finally, the extension of type τ, denoted by $ext(\tau)$, is defined as:

$ext(\tau) = \{e \in terms(\tau) \mid FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\}\}$.

where $FV(e)$ is the set of free variables in term e:

$FV(y) = \{y\}$ if $y \in Var$,
$FV(b) = \emptyset$ if $b \in Cons$,
$FV(\{e_1, \cdots, e_n\}) = FV(< l_1 = e_1, \cdots, l_n = e_n >) = FV(e_1) \cup \cdots \cup FV(e_n)$,
$FV(\mu y.e) = FV(e) - \{y\}$.

□

Example 1. Let τ be type $\mu t. <\mathrm{a} : \mathrm{integer}, \mathrm{b} : t>$. Furthermore, let x and y be elements of Var_t. Then $\mu x. < \mathrm{a} = 1, \mathrm{b} = \mu y. < \mathrm{a} = 2, \mathrm{b} = x >>$ is a term of type τ. □

Similar to types, terms can be represented by trees.

Definition 5. Let e be a term of an arbitrary type. The tree representing e is defined as $struc(e, \emptyset)$, where $struc(e, V)$ is defined as:

$struc(x, V) =$

$$x$$

if $x \in Var$ and $\eta_V(x) = \perp$,

$struc(x, V) = struc(\mu x.e_x, V)$
if $x \in Var$ and $\eta_V(x) = \mu x.e_x$,

$struc(b, V) =$

$$b$$

if $b \in Cons$,

$struc(\emptyset, V) =$

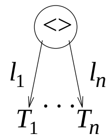
$$\{\},$$

$struc(\{e_1, \cdots, e_n\}, V) =$

$$\begin{tikzcd} & \{\} \arrow[dl, "\in"'] \arrow[dr, "\in"] & \\ T_1 & \cdots & T_n \end{tikzcd}$$

where $T_i = struc(e_i, V)$,

$struc(< l_1 = e_1, \cdots, l_n = e_n >, V) =$



where $T_i = struc(e_i, V)$,

$struc(\mu x.e_x, V) = struc(e_x, V \cup \{\mu x.e_x\})$,

where η_V is a partial function from *Var* to *Terms* induced by V; $\eta_V(x) = e$ if e is a term in V and e starts with μx (there is at most one such term) and $\eta_V(x) = \perp$ if there is no term in V that starts with μx. For convenience, we sometimes write $struc(e)$ instead of $struc(e, \emptyset)$. □

The equivalence relation on terms is defined as an equality relation on the trees of the terms.

Definition 6. Let e_1 and e_2 be terms of arbitrary types. Equivalence of e_1 and e_2, denoted by $e_1 \cong e_2$, is defined as follows:

$$e_1 \cong e_2 \;\Leftrightarrow\; struc(e_1) = struc(e_2).$$

Let E_1 and E_2 be sets of terms. Then E_1 is equivalent to E_2, denoted by $E_1 \cong E_2$, if and only if:

1. $\forall e_1 \in E_1 \exists e_2 \in E_2 [e_1 \cong e_2]$
2. $\forall e_2 \in E_2 \exists e_1 \in E_1 [e_2 \cong e_1]$.

□

For the definition of the subterm relation, we need the following definition.

Definition 7. Let T_1 and T_2 be labeled trees. Furthermore, let φ be a graph homomorphism from T_1 to T_2. Then φ is a tree morphism if and only if φ maps the root to the root. And T_1 is a supertree of T_2, denoted by $T_1 \sqsupseteq T_2$, if and only if there is an injective tree morphism from T_2 to T_1 that preserves labels.

Let S_1 and S_2 be sets of labeled trees. Then S_1 is a set of supertrees of S_2, denoted by $S_1 \sqsupseteq S_2$, if and only if:

$$\forall s_1 \in S_1 \exists s_2 \in S_2 [s_1 \sqsupseteq s_2].$$

□

The subterm relation on terms is defined as a supertree relation on the trees of the terms.

Definition 8. Let e_1 and e_2 be terms of arbitrary types. The subterm relation, where $e_1 \preceq e_2$ denotes that e_1 is a subterm of e_2, is defined by:

$$e_1 \preceq e_2 \;\Leftrightarrow\; struc(e_1) \sqsupseteq struc(e_2).$$

Let E_1 and E_2 be sets of terms. Then E_1 is a set of subterms of E_2, denoted by $E_1 \preceq E_2$, if and only if:

$$\forall e_1 \in E_1 \exists e_2 \in E_2 [e_1 \preceq e_2].$$

□

The following lemma gives the relation between the 'set of subterms' relation and the 'set of supertrees' relation.

Lemma 1. Let E_1 and E_2 be sets of terms. Then:

$$E_1 \preceq E_2 \;\Leftrightarrow\; \{struc(e) \mid e \in E_1\} \sqsupseteq \{struc(e) \mid e \in E_2\}.$$

Proof. The lemma follows from Definition 7 and Definition 8. □

3 Type equivalence

In this section, we define structural, extensional, and derivable type equivalence. We prove that derivable equivalence is sound and complete w.r.t. structural and extensional equivalence, and that structural and extensional equivalence are decidable. Furthermore, we define a normalisation process for types and prove that derivable equivalence is logically equivalent to the equivalence relation induced by the normalisation process.

In the previous section, the semantics of a type was defined in terms of a tree representing the structure of the type and in terms of a set of closed terms of which the structure corresponds to the structure of the type. These two notions of type semantics can be used to define two notions of semantic type equivalence: structural and extensional. Structural type equivalence is defined as an equality relation on the trees of the types.

Definition 9. Let τ_1 and τ_2 be types. Structural equivalence of τ_1 and τ_2, denoted by $\tau_1 \cong_{struc} \tau_2$, is defined as:

$$\tau_1 \cong_{struc} \tau_2 \;\Leftrightarrow\; struc(\tau_1) = struc(\tau_2).$$

□

Extensional type equivalence is defined as an equivalence relation on the extensions of the types.

Definition 10. Let τ_1 and τ_2 be types. Extensional equivalence of τ_1 and τ_2, denoted by $\tau_1 \cong_{ext} \tau_2$, is defined as:

$$\tau_1 \cong_{ext} \tau_2 \;\Leftrightarrow\; ext(\tau_1) \cong ext(\tau_2).$$

□

3.1 Derivation system for type equivalence

In this subsection, we introduce a derivation system for type equivalence and define derivable type equivalence. Informally, a derivation is a tree of formulas, where the children formulas imply the parent formula. A formula is of the form $\Gamma \vdash \tau \cong \sigma$ (where τ and σ are types, $\cong$ is type equivalence, and Γ is a context), saying that $\tau \cong \sigma$ follows from the axioms for basic types and the premises in Γ. Context Γ is a triple $(\Gamma_l, \Gamma_r, \Gamma_p)$, where an element of Γ_l is a type definition of the form $\mu t.\alpha$, saying that every type variable t on the left (i.e., in τ) corresponds to type $\mu t.\alpha$, an element of Γ_r is a type definition of the form $\mu t.\alpha$, saying that every type variable t on the right (i.e., in σ) corresponds to type $\mu t.\alpha$, and an element of Γ_p is a pair of type variables (t, s), where t occurs on the left and s occurs on the right, saying that t and s are equivalent (and the types they corresponds to). For convenience, we write $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\}$ instead of $(\Gamma_l \cup \{\mu t.\alpha\}, \Gamma_r \cup \{\mu s.\beta\}, \Gamma_p \cup \{(t, s)\})$.

Definition 11. The derivation system for type equivalence, denoted by DE, is defined as follows. The axioms of the derivation system are:

1.	$\Gamma \vdash B \cong B$	if $B \in BTypes$
2.	$\Gamma \vdash t \cong s$	if $(t, s) \in \Gamma_p$
3.	$\Gamma \vdash t \cong \mu s.\beta$	if $(t, s) \in \Gamma_p \wedge \mu s.\beta \in \Gamma_r$
4.	$\Gamma \vdash \mu t.\alpha \cong s$	if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$
5.	$\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$	if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$

The rules of the derivation system are:

1. $\dfrac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash t \cong s}$ if $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$

2. $\dfrac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash t \cong \mu s.\beta}$ if $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$

3. $$\frac{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta}{\Gamma \vdash \mu t.\alpha \cong s} \qquad \text{if } (t,s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$$

4. $$\frac{\Gamma \vdash \tau \cong \sigma}{\Gamma \vdash \{\tau\} \cong \{\sigma\}}$$

5. $$\frac{\Gamma \vdash \tau_1 \cong \sigma_1, \cdots, \Gamma \vdash \tau_n \cong \sigma_n}{\Gamma \vdash < l_1 : \tau_1, \cdots, l_n : \tau_n > \cong < l_1 : \sigma_1, \cdots, l_n : \sigma_n >}$$

6. $$\frac{\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t,s)\} \vdash \alpha \cong \beta}{\Gamma \vdash \mu t.\alpha \cong \mu s.\beta} \qquad \text{if } (t,s) \notin \Gamma_p.$$

Rules 1, 2, and 3 introduce folding of types (going from premise to conclusion) and unfolding of types (going from conclusion to premise). □

The set of axioms and rules of DE is the same as the extended set of rules for $=_A$ from [2]. We need the extended set for structural and extensional completeness. Derivable type equivalence for closed μ-complete types is given by the following definition.

Definition 12. Let τ_1 and τ_2 be closed μ-complete types. Equivalence of τ_1 and τ_2 according to derivation system DE, denoted by $\tau_1 \cong_D \tau_2$, is defined as follows:

$$\tau_1 \cong_D \tau_2 \;\Leftrightarrow\; \emptyset \vdash_{DE} \tau_1 \cong \tau_2,$$

where $\Gamma \vdash_{DE} \tau \cong \sigma$ means that there is a derivation in DE with conclusion $\Gamma \vdash \tau \cong \sigma$. □

Derivable type equality for closed μ-complete types, denoted by $=_D$, is obtained in the same way, viz., from the subsystem of DE that consists of axioms 1 and 2, and rules 4, 5, and 6. Since there is no folding or unfolding in the subsystem, derivable equality is just equality modulo renaming of type variables.

The context of a formula in a derivation induces a function from the set of free type variables on the left to the set of types and a function from the set of free type variables on the right to the set of types.

Lemma 2. Let $\Gamma \vdash \tau \cong \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$. If t is a free type variable in τ, then there is exactly one τ' in Γ_l that starts with μt.
Proof. Let $\Gamma \vdash \tau \cong \sigma$ be a step in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, where τ_1 and τ_2 are closed types, and t be a free type variable in τ. Since t is bound by μt in τ_1 and rule 6 is the only rule in which μt is removed, there is at least one type in Γ_l that starts with μt. Furthermore, since every type in Γ_l is a substring of τ_1 (follows from a simple induction) and there is only one substring of τ_1 that starts with μt and is an element of *Types* at the same time, there is at most one type in Γ_l that starts with μt. □

In the same way as Γ induces a partial function from the set of type variables to the set of types in Definition 3, Γ_l induces a total function η_{Γ_l} from the set of free type variables on the right to the set of types. Next, we give an example of a derivation.

Example 2. For convenience, we define a number of abbreviations:

$\alpha = < a_1 : B, a_2 : t, a_3 : t >$
$\beta = < a_1 : B, a_2 : s, a_3 : \mu s'.\beta' >$
$\beta' = < a_1 : B, a_2 : s, a_3 : s' >$
$\Gamma_1 = \{(\mu t.\alpha, \mu s.\beta), (t,s)\}$
$\Gamma_2 = \Gamma_1 \cup \{(\mu t.\alpha, \mu s'.\beta'), (t, s')\}.$

Using derivation system DE, we obtain the following derivation for $\emptyset \vdash \mu t.\alpha \cong \mu s.\beta$:

$$\dfrac{\dfrac{\Gamma_1 \vdash B \cong B,\ \ \Gamma_1 \vdash t \cong s,\ \ \dfrac{\dfrac{\dfrac{\Gamma_2 \vdash B \cong B,\ \Gamma_2 \vdash t \cong s,\ \Gamma_2 \vdash t \cong s'}{\Gamma_2 \vdash \alpha \cong \beta'}\text{(rule 5)}}{\Gamma_1 \vdash \mu t.\alpha \cong \mu s'.\beta'}\text{(rule 6)}}{\Gamma_1 \vdash t \cong \mu s'.\beta'}\text{(rule 2)}}{\Gamma_1 \vdash \alpha \cong \beta}\text{(rule 5)}}{\emptyset \vdash \mu t.\alpha \cong \mu s.\beta}\text{(rule 6)}$$

□

In the sequel, we will prove the following theorems.

Theorem 1. Derivable equivalence is sound and complete w.r.t. structural equivalence. □

Theorem 2. Structural and extensional equivalence are logically equivalent. □

Theorem 3. Structural and extensional equivalence are decidable. □

Finally, using Theorem 1 and 2, we can deduce the following corollary.

Corollary 1. Derivable equivalence is sound and complete w.r.t. extensional equivalence. □

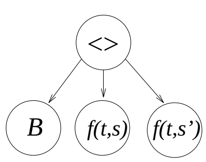
3.2 Soundness w.r.t. structural equivalence

In this subsection, we prove the soundness part of Theorem 1. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

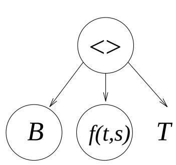
$$\tau_1 \cong_D \tau_2 \ \Rightarrow\ \tau_1 \cong_{struc} \tau_2.$$

First, for every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, a tree is constructed. The tree is constructed in such a way that it is equal to both $struc(\tau, \Gamma_l)$ and $struc(\sigma, \Gamma_r)$, except for the free type variables. Following the derivation, constructing the tree for a formula from the trees for its children formulas, we obtain a tree that is equal to both $struc(\tau_1, \emptyset)$ and $struc(\tau_2, \emptyset)$, (because τ_1 and τ_2 have no free type variables).
Before giving the definition, we give an example of how to construct the trees corresponding to the formulas in a derivation.

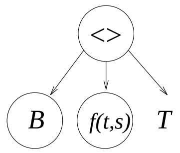
Example 3. Let D be the derivation of Example 1 and f be an injective function from $\{t\} \times \{s, s'\}$ to $TypeVar - \{t, s, s'\}$. The tree for $\Gamma_2 \vdash \alpha \cong \beta'$, denoted by $tree(\alpha, \beta', \Gamma_2)$, is given by:



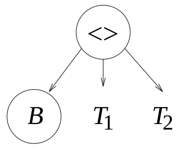
The tree consists of $tree(B, B, \Gamma_2)$, $tree(t, s, \Gamma_2)$, and $tree(t, s', \Gamma_2)$. Furthermore, the tree for $\Gamma_1 \vdash \mu t.\alpha \cong \mu s'.\beta'$, which is the same as the tree for $\Gamma_1 \vdash t \cong \mu s'.\beta'$, is given by:



where $T = tree(\mu t.\alpha, \mu s'.\beta', \Gamma_2)$. The tree is an extension of the tree for $\Gamma_2 \vdash \alpha \cong \beta'$. The tree for $\Gamma_1 \vdash \alpha \cong \beta$ is given by:



where $T = tree(\mu t.\alpha, \mu s'.\beta', \Gamma_2)$. The tree consists of $tree(B, B, \Gamma_1)$, $tree(t, s, \Gamma_1)$, and the tree for $\Gamma_1 \vdash t \cong \mu s'.\beta'$. And the tree for $\Gamma_0 \vdash \mu t.\alpha \cong \mu s.\beta$ is given by:

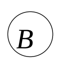


where $T_1 = tree(\mu t.\alpha, \mu s.\beta, \Gamma_0)$ and $T_2 = tree(\mu t.\alpha, \mu s'.\beta', \Gamma_0)$. The tree is an extension of the tree for $\Gamma_1 \vdash \alpha \cong \beta$. $\Box$

Next, we give the definition of trees corresponding to formulas in a derivation.

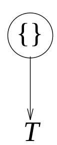
Definition 13. Let τ_1 and τ_2 be closed μ-complete types. Furthermore, let f be an injective function from $bvars(\tau_1) \times bvars(\tau_2)$ to $TypeVar - (bvars(\tau_1) \cup bvars(\tau_2))$ and $\Gamma \vdash \tau \cong \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$. The tree for $\Gamma \vdash \tau \cong \sigma$, denoted by $tree(\tau, \sigma, \Gamma)$, is defined as follows:

$tree(B, B, \Gamma) =$



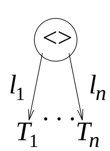
if $B \in BTypes$

$tree(\{\tau'\}, \{\sigma'\}, \Gamma) =$



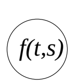
where $T = tree(\tau', \sigma', \Gamma)$

$tree(< l_1 : \tau_1, \cdots, l_n : \tau_n >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma) =$



where $T_i = tree(\tau_i, \sigma_i, \Gamma)$

$tree(t, s, \Gamma) = tree(t, \mu s.\beta, \Gamma) = tree(\mu t.\alpha, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma)$



if $(t, s) \in \Gamma_p$

$$tree(t, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma)$$
$$\text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$$

$$tree(t, \mu s.\beta, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma)$$
$$\text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha$$

$$tree(\mu t.\alpha, s, \Gamma) = tree(\mu t.\alpha, \mu s.\beta, \Gamma)$$
$$\text{if } (t, s) \notin \Gamma_p \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$$

$$tree(\mu t.\alpha, \mu s.\beta, \Gamma) = tree(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\})$$
$$\text{if } (t, s) \notin \Gamma_p.$$

□

Finally, we prove Claim 1:

$$\emptyset \vdash_{DE} \tau_1 \cong \tau_2 \;\Rightarrow\; (tree(\tau_1, \tau_2, \emptyset) = struc(\tau_1, \emptyset) \wedge tree(\tau_1, \tau_2, \emptyset) = struc(\tau_2, \emptyset)).$$

From this claim it follows that derivable equivalence is sound w.r.t. structural equivalence.

$$\emptyset \vdash_{DE} \tau_1 \cong \tau_2 \;\Rightarrow\; struc(\tau_1) = struc(\tau_2).$$

3.2.1 Proof of Claim 1

If $\Gamma \vdash \tau \cong \sigma$ is a formula in the derivation of $\emptyset \vdash \tau_1 \cong \tau_2$, then $tree(\tau, \sigma, \Gamma)$ can contain free type variables, whereas $struc(\tau, \Gamma)$ cannot. Therefore, we prove the following claim; for every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \cong \tau_2$:

$$tree(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu t'.\alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t'.\alpha'] = struc(\tau, \Gamma_l),$$

where $T[t_i \setminus T_i \mid i \in I]$ is the tree obtained from T by replacing every leaf labeled t_i by tree T_i, for every $i \in I$. The proof is an induction argument on the distance of a formula in the derivation tree to its remotest descendant. Base step: the formula is an axiom. For sake of convenience, let $[S_\Gamma]$ denote

$$[f(t', s') \setminus struc(\mu t'.\alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t'.\alpha'].$$

Axiom 1: $\Gamma \vdash B \cong B$. Then, obviously:

$$tree(B, B, \Gamma)[S_\Gamma] = tree(B, B, \Gamma) = struc(B, \Gamma_l).$$

Axiom 2: $\Gamma \vdash t \cong s$. Then (t, s) must be an element of Γ_p and there must be a type α, such that $\eta_{\Gamma_l}(t) = \mu t.\alpha$. Hence:

$$tree(t, s, \Gamma)[S_\Gamma] = struc(\mu t.\alpha, \Gamma_l) = struc(t, \Gamma_l).$$

Axiom 3, 4, or 5: $\Gamma \vdash t \cong \mu s.\beta$, $\Gamma \vdash \mu t.\alpha \cong s$, or $\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$. Similar to the previous case.
Induction step: the formula is the result of applying a rule to a number of formulas which are closer to their remotest descendant.
Rule 1: $\Gamma \vdash t \cong s$. Then there must be types $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$, such that:

$$\Gamma \vdash \mu t.\alpha \cong \mu s.\beta.$$

Using the induction hypothesis, we can conclude: $tree(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = struc(\mu t.\alpha, \Gamma_l)$. Since (t, s) is not an element of Γ_p, it follows that:

$$tree(t, s, \Gamma)[S_\Gamma] = tree(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = struc(\mu t.\alpha, \Gamma_l) = struc(t, \Gamma_l).$$

Rule 2 or 3: $\Gamma \vdash t \cong \mu s.\beta$ or $\Gamma \vdash \mu t.\alpha \cong s$. Similar to the previous case.
Rule 4: $\Gamma \vdash \{\tau\} \cong \{\sigma\}$. From $\Gamma \vdash \tau \cong \sigma$ and the induction hypothesis, it follows that:

$$tree(\tau, \sigma, \Gamma)[S_\Gamma] = struc(\tau, \Gamma_l).$$

Hence:

$$tree(\{\tau\}, \{\sigma\}, \Gamma)[S_\Gamma] = struc(\{\tau\}, \Gamma_l).$$

Rule 5: $\Gamma \vdash < l_1 : \tau_1, \cdots, l_n : \tau_n > \cong < l_1 : \sigma_1, \cdots, l_n : \sigma_n >$. Using the induction hypothesis for $\Gamma \vdash \tau_i \cong \sigma_i$, we can conclude that:

$$tree(\tau_i, \sigma_i, \Gamma)[S_\Gamma] = struc(\tau_i, \Gamma_l).$$

Hence:

$$tree(< l_1 : \tau_1, \cdots, l_n : \tau_n >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma)[S_\Gamma] = struc(< l_1 : \tau_1, \cdots, l_n : \tau_n >, \Gamma_l).$$

Rule 6: $\Gamma \vdash \mu t.\alpha \cong \mu s.\beta$. Let Γ' be $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}$ and Δ be $\Gamma' \cup \{(t, s)\}$. From $\Delta \vdash \alpha \cong \beta$ and the induction hypothesis, it follows that:

a) $tree(\alpha, \beta, \Delta)[S_\Delta] = struc(\alpha, \Delta_l)$.

For every natural number i, define $tree_i(\alpha, \beta, \Delta)$ as follows:

$$tree_1(\alpha, \beta, \Delta) = tree(\alpha, \beta, \Delta),$$
$$tree_{i+1}(\alpha, \beta, \Delta) = tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)].$$

Using an induction argument on i, we can prove that, for every natural number i:

b) $tree_i(\alpha, \beta, \Delta)[S_\Delta] = struc(\alpha, \Delta_l)$.

The base step follows from a) and the induction step is:

$$\begin{aligned}
&tree_{i+1}(\alpha, \beta, \Delta)[S_\Delta] = (tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)])[S_\Delta] = \\
&\quad (tree(\alpha, \beta, \Delta)[f(t, s) \setminus (tree_i(\alpha, \beta, \Delta)[S_\Delta]), S_\Gamma] = \\
&\quad (tree(\alpha, \beta, \Delta)[f(t, s) \setminus struc(\alpha, \Delta_l), S_\Gamma] = \\
&\quad (tree(\alpha, \beta, \Delta)[f(t, s) \setminus struc(\mu t.\alpha, \Delta_l), S_\Gamma] = \\
&\quad tree(\alpha, \beta, \Delta)[S_\Delta] = \\
&\quad struc(\alpha, \Delta_l),
\end{aligned}$$

where the first step follows from the definition of $tree_{i+1}$, the second follows from the definition of substitution and the fact that $\Delta_p = \Gamma_p \cup \{(t, s)\}$, the third from the induction hypothesis, the fourth from the definition of *struc*, the fifth from the definition of S_Δ, and the final from a). Furthermore, from an induction argument on the distance of a formula to its remotest descendant in the derivation tree for $\Delta \vdash \alpha \cong \beta$, it follows that:

c) $tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = tree(\alpha, \beta, \Gamma')$.

The non-trivial case of the induction is :

$$\begin{aligned}
&tree(t, s, \Delta')[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = \\
&\quad tree(\alpha, \beta, \Gamma') = tree(\mu t.\alpha, \mu s.\beta, \Gamma') = tree(t, s, \Gamma'),
\end{aligned}$$

where $\Delta' \supseteq \Delta$. Again, using an induction argument on i, we can prove that, for every natural number i:

$$tree_i(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = tree(\alpha, \beta, \Gamma').$$

The base step follows from c) and the induction step is:

$$\begin{aligned}
&tree_{i+1}(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = \\
&\quad (tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree_i(\alpha, \beta, \Delta)])[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = \\
&\quad tree(\alpha, \beta, \Delta)[f(t, s) \setminus (tree_i(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')])] = \\
&\quad tree(\alpha, \beta, \Delta)[f(t, s) \setminus tree(\alpha, \beta, \Gamma')] = \\
&\quad tree(\alpha, \beta, \Gamma'),
\end{aligned}$$

where the first step follows from the definition of $tree_{i+1}$, the second from the definition of substitution, the third from the induction hypothesis, and the fourth from c). This means that $tree_i(\alpha, \beta, \Delta)$ is equal to $tree(\alpha, \beta, \Gamma')$ from the root to at least depth i. Hence, $tree(\alpha, \beta, \Gamma')[S_\Delta]$ is equal to $tree_i(\alpha, \beta, \Delta)\ [S_\Delta]$ from the root to at least depth i. Furthermore, from the fact that $f(t, s)$ does not appear in $tree(\alpha, \beta, \Gamma')$, it follows that:

$$tree(\alpha, \beta, \Gamma')[S_\Gamma] = tree(\alpha, \beta, \Gamma')[S_\Delta].$$

Using b), we can deduce that $tree(\alpha, \beta, \Gamma')[S_\Gamma]$ is equal to $struc(\alpha, \Delta_l)$. Hence:

$$tree(\mu t.\alpha, \mu s.\beta, \Gamma)[S_\Gamma] = tree(\alpha, \beta, \Gamma')[S_\Gamma] = struc(\alpha, \Delta_l) = struc(\mu t.\alpha, \Gamma_l).$$

In the same way, we can prove the following claim; for every formula $\Gamma \vdash \tau \cong \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \cong \tau_2$:

$$tree(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu s'.\beta', \Gamma_r) \mid (t', s') \in \Gamma \wedge \eta_{\Gamma_r}(s') = \mu s'.\beta'] = struc(\sigma, \Gamma_r).$$

3.3 Completeness w.r.t. structural equivalence

In this subsection, we prove the completeness part of Theorem 1. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

$$\tau_1 \cong_{struc} \tau_2 \ \Rightarrow\ \tau_1 \cong_D \tau_2.$$

First, a structural equivalence tree for τ_1 and τ_2 is constructed (the structural equivalence tree will be proven isomorphic to the derivation tree with conclusion $\emptyset \vdash \tau_1 \cong \tau_2$). The tree is constructed in such a way that every node is labeled by a tuple of the form (τ, σ, Γ), where τ and σ are obtained by using the structure of τ_1 and τ_2 (and, if necessary, by unfolding types), such that $struc(\tau, \Gamma_l) = struc(\sigma, \Gamma_r)$. For example, the root is labeled $(\tau_1, \tau_2, \emptyset)$.
For the definition of structural equivalence trees, we need the following lemma.

Lemma 3. Let τ and σ be μ-complete types and Γ be a context, such that $struc(\tau, \Gamma_l)$ is equal to $struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Then:

1. $\tau = B \ \Rightarrow\ \sigma = B$
2. $\tau = \{\tau_1\} \ \Rightarrow\ (\sigma = \{\sigma_1\} \wedge struc(\tau_1, \Gamma_l) = struc(\sigma_1, \Gamma_r))$
3. $\tau = < l_1 : \tau_1, \cdots, l_n : \tau_n > \Rightarrow$
   $(\sigma = < l_1 : \sigma_1, \cdots, l_n : \sigma_n > \wedge struc(\tau_i, \Gamma_l) = struc(\sigma_i, \Gamma_r))$
4. $\tau = t \ \Rightarrow$
   $(\sigma = s \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta \wedge struc(\alpha, \Gamma_l) = struc(\beta, \Gamma_r)) \vee$
   $(\sigma = \mu s.\beta \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge struc(\alpha, \Gamma_l) = struc(\beta, \Gamma_r \cup \{\sigma\}))$
5. $\tau = \mu t.\alpha \ \Rightarrow$
   $(\sigma = s \wedge \eta_{\Gamma_r}(s) = \mu s.\beta \wedge struc(\alpha, \Gamma_l \cup \{\mu t.\alpha\}) = struc(\beta, \Gamma_r)) \vee$
   $(\sigma = \mu s.\beta \wedge struc(\alpha, \Gamma_l \cup \{\mu t.\alpha\}) = struc(\beta, \Gamma_r \cup \{\sigma\}))$.

Proof. The lemma follows from Definition 3. $\square$

Now, we can define the children of a node in a structural equivalence tree.

Definition 14. Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ-complete types, Γ is a context, $struc(\tau, \Gamma_l)$ is equal to $struc(\sigma, \Gamma_r)$, and $struc(\tau, \Gamma_l)$ contains no type variables. According to Lemma 3, there are 5 cases for x, of which the last two cases each have two subcases. For the definition of the children of x, we divide both subcases into two new subcases (one for $(t, s) \in \Gamma_p$ and one for $(t, s) \notin \Gamma_p$), obtaining 11 cases for x. The set of children of x, denoted by $eqchildren(x)$ is defined as follows:

1. if $x = (B, B, \Gamma)$, then $eqchildren(x) = \emptyset$
2. if $x = (t, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
3. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
4. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
5. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $eqchildren(x) = \emptyset$
6. if $x = (\{\tau\}, \{\sigma\}, \Gamma)$, then $eqchildren(x) = \{(\tau, \sigma, \Gamma)\}$
7. if $x = (< l_1 : \tau_1, \cdots, l_n : \tau_n >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma)$, then $eqchildren(x) = \{(\tau_1, \sigma_1, \Gamma), \cdots, (\tau_n, \sigma_n, \Gamma)\}$
8. if $x = (t, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
9. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
10. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$, then $eqchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
11. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma$, then $eqchildren(x) = \{(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\})\}$.

□

Before giving the definition, we give an example of how to construct a structural equivalence tree.

Example 4. First, we define the following abbreviations:

$$\alpha = < a_1 : B, a_2 : t, a_3 : t >$$
$$\beta = < a_1 : B, a_2 : s, a_3 : \mu s'.\beta' >$$
$$\beta' = < a_1 : B, a_2 : s, a_3 : s' >$$
$$\Gamma_1 = \{(\mu t.\alpha, \mu s.\beta), (t, s)\}$$
$$\Gamma_2 = \Gamma_1 \cup \{(\mu t.\alpha, \mu s'.\beta'), (t, s')\}.$$

Then $struc(\mu t.\alpha, \emptyset)$ is equal to $struc(\mu s.\beta, \emptyset)$. The descendants of $(\mu t.\alpha, \mu s.\beta, \emptyset)$ are given by:

$$eqchildren((\mu t.\alpha, \mu s.\beta, \emptyset)) = \{(\alpha, \beta, \Gamma_1)\}$$
$$eqchildren((\alpha, \beta, \Gamma_1)) = \{(B, B, \Gamma_1), (t, s, \Gamma_1), (t, \mu s'.\beta', \Gamma_1)\}$$
$$eqchildren((B, B, \Gamma_1)) = \emptyset$$
$$eqchildren((t, s, \Gamma_1)) = \emptyset$$
$$eqchildren((t, \mu s'.\beta', \Gamma_1)) = \{(\mu t.\alpha, \mu s'.\beta', \Gamma_1)\}$$
$$eqchildren((\mu t.\alpha, \mu s'.\beta', \Gamma_1)) = \{(\alpha, \beta', \Gamma_2)\}$$
$$eqchildren((\alpha, \beta', \Gamma_2)) = \{(B, B, \Gamma_2), (t, s, \Gamma_2), (t, s', \Gamma_2)\}$$
$$eqchildren((B, B, \Gamma_2)) = \emptyset$$
$$eqchildren((t, s, \Gamma_2)) = \emptyset$$
$$eqchildren((t, s', \Gamma_2)) = \emptyset.$$

□

The following lemma states that, if a tuple satisfies the precondition of Definition 14, then so do its children.

Lemma 4. Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ-complete types, Γ is a context, $struc(\tau, \Gamma_l)$ is equal to $struc(\sigma, \Gamma_r)$, and $struc(\tau, \Gamma_l)$ contains no type variables. Then every element of $eqchildren(x)$ is a tuple $x' = (\tau', \sigma', \Gamma')$, such that $struc(\tau', \Gamma'_l) = struc(\sigma', \Gamma'_r)$ and $struc(\tau', \Gamma'_l)$ contains no type variables.
proof. The lemma follows from Definition 14 and Lemma 3. □

Using Lemma 4, we can finally give the definition of structural equivalence trees.

Definition 15. Let τ_1 and τ_2 be closed μ-complete types, such that $struc(\tau_1, \emptyset) = struc(\tau_2, \emptyset)$. The structural equivalence tree for τ_1 and τ_2 is defined as $eqtree((\tau_1, \tau_2, \emptyset))$, where $eqtree((\tau, \sigma, \Gamma))$ is defined as follows:

1. if $x = (\tau, \sigma, \Gamma)$ and $eqchildren(x) = \emptyset$,
   then $eqtree(x)$ has only one node, labeled (τ, σ, Γ)
2. if $x = (\tau, \sigma, \Gamma)$ and $eqchildren(x) \neq \emptyset$,
   then $eqtree(x)$ consists of a root, labeled (τ, σ, Γ), and, for every $y \in eqchildren(x)$, a subtree $eqtree(y)$ and an arrow from the root to subtree $eqtree(y)$.

$\square$

Lemma 5. The structural equivalence tree for τ_1 and τ_2 is finite.
Proof. The structural equivalence tree for τ_1 and τ_2 is constructed by starting with root $(\tau_1, \tau_2, \emptyset)$ and applying the definition of *eqchildren* to the leaves, until every leaf has an empty set of children. Case 6, 7, and 11 can only be applied a finite number of times consecutively, because they decrease the complexity of the types. Case 8, 9, and 10 can only be applied to a leaf (t, s, Γ) (resp., $(t, \mu s.\beta, \Gamma)$ and $(\mu t.\alpha, s, \Gamma)$) if (t, s) is not an element of Γ. However, after one of these rules has been applied, case 11 will be applied, adding (t, s) to Γ. This means that neither case 8, 9, or 10 can be applied more than once for the same pair of type variables (t, s) on a path from the root to a leaf. Since there are only finitely many type variables in τ_1 and τ_2, case 8, 9, and 10 can only be applied a finite number of times.

From these observations it follows that every rule can only be applied a finite number of times. Hence, the resulting structural equivalence tree is finite. $\square$

Finally, we prove Claim 2:

for every node labeled (τ, σ, Γ) in $eqtree(\tau_1, \tau_2, \emptyset)$: $\Gamma \vdash_{DE} \tau \cong \sigma$.

From the definition of *eqtree* and Claim 2 it follows that derivable equivalence is complete w.r.t. structural equivalence:

$$struc(\tau_1) = struc(\tau_2) \;\Rightarrow\; \emptyset \vdash_{DE} \tau_1 \cong \tau_2.$$

3.3.1 Proof of Claim 2

The proof is an induction argument on the distance of a node to its remotest descendant. Base step: the node is a leaf.
Case 1: x is labeled (B, B, Γ). Then, using axiom 1 of the derivation system, we have: $\Gamma \vdash_{DE} B \cong B$.
Case 2: x is labeled (t, s, Γ) and $(t, s) \in \Gamma_p$. Then, using axiom 2, we have: $\Gamma \vdash_{DE} t \cong s$.
Case 3: x is labeled $(t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 3, we have: $\Gamma \vdash_{DE} t \cong \mu s.\beta$.
Case 4: x is labeled $(\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 4, we have: $\Gamma \vdash_{DE} \mu t.\alpha \cong s$.
Case 5: x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$. Then, using axiom 5, we have: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$.
Induction step: the node is the parent of a number of nodes which are closer to their remotest descendant.
Case 6: x is labeled $(\{\tau'\}, \{\sigma'\}, \Gamma)$. The only child of x is labeled (τ', σ', Γ). Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \tau \cong \sigma$. Hence, from rule 4 of the derivation system, it follows that: $\Gamma \vdash_{DE} \{\tau\} \cong \{\sigma\}$.
Case 7: x is labeled $(< l_1 : \tau_1, \cdots, l_n : \tau_n >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma)$. The children of x are labeled $(\tau_i, \sigma_i, \Gamma)$. Using the induction hypothesis, for every $i \in \{1, \cdots, n\}$ we can conclude: $\Gamma \vdash_{DE} \tau_i \cong \sigma_i$. Hence, from rule 5 of the derivation system, it follows that: $\Gamma \vdash_{DE} < l_1 : \tau_1, \cdots, l_n : \tau_n > \cong < l_1 : \sigma_1, \cdots, l_n : \sigma_n >$.
Case 8: x is labeled (t, s, Γ) and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$. Hence, from rule 1, it follows that: $\Gamma \vdash_{DE} t \cong s$.
Case 9: x is labeled $(t, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$. Hence, from rule 2, it follows that: $\Gamma \vdash_{DE} t \cong \mu s.\beta$.

Case 10: x is labeled $(\mu t.\alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$. Using the induction hypothesis, we can conclude: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$. Hence, from rule 3, it follows that: $\Gamma \vdash_{DE} \mu t.\alpha \cong s$.
Case 11: x is labeled $(\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p$. The only child of x is labeled $(\alpha, \beta, \Gamma \cup \{\mu t.\alpha, \mu s.\beta, (t, s)\})$. Using the induction hypothesis, we can conclude: $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \vdash_{DE} \alpha \cong \beta$. Hence, from rule 6, it follows that: $\Gamma \vdash_{DE} \mu t.\alpha \cong \mu s.\beta$.

3.4 Equivalence of structural and extensional equivalence

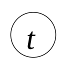
In this subsection, we prove Theorem 2. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

$$\tau_1 \cong_{struc} \tau_2 \;\Leftrightarrow\; \tau_1 \cong_{ext} \tau_2.$$

First, for every type τ, the structural extension is defined (structural extensions will be proven equal for types represented by the same tree). The structural extension is defined in such a way that it directly corresponds to the extension of the associated type.
For the definition of structural instances, we need a number of preliminary definitions. The exact tree of a type is the same as the tree of the type, except that the exact tree contains type variables.

Definition 16. Let τ be a type. The exact tree representing τ is defined as $estruc(\tau, \emptyset)$, where $estruc(\tau', \Gamma)$ is defined as follows:

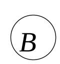
$estruc(t, \Gamma) =$



if $t \in TypeVar$ and $\eta_\Gamma(t) = \perp$,

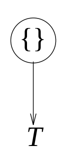
$estruc(t, \Gamma) = estruc(\mu t.\alpha, \Gamma)$
if $t \in TypeVar$ and $\eta_\Gamma(t) = \mu t.\alpha$,

$estruc(B, \Gamma) =$



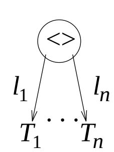
if $B \in BTypes$,

$estruc(\{\tau_1\}, \Gamma) =$



where $T = estruc(\tau_1, \Gamma)$,

$estruc(< l_1 : \tau_1, \cdots, l_n : \tau_n >) =$



where $T_i = estruc(\tau_i, \Gamma)$

$estruc(\mu\, t. < l_1 : \tau_1, \cdots, l_n : \tau_n >, \Gamma) =$

$$\begin{tikzcd} & t \arrow[dl, "l_1"'] \arrow[dr, "l_n"] & \\ T_1 & \cdots & T_n \end{tikzcd}$$

where $T_i = estruc(\tau_i, \Gamma \cup \{\mu\, t. < l_1 : \tau_1, \cdots, l_n : \tau_n >\})$.

□

Lemma 6. Let τ be a type. Then there is a bijective tree homomorphism φ from $estruc(\tau)$ to $struc(\tau)$, such that for every node or arrow q the following holds:

1. if $label(q) \in TypeVar$ and q is not a leaf, then $label(\varphi(q)) = <>$
2. otherwise, $label(\varphi(q)) = label(q)$.

where $label(q)$ denotes the label of node or arrow q.
Proof. The lemma follows from Definition 3 and Definition 16. □

The exact tree of a term is the same as the tree of the term, except that the exact tree contains instance variables.

Definition 17. Let e be a term. The exact tree representing e is defined as $estruc(e, \emptyset)$, where $estruc(e', \Gamma)$ is defined as follows:

$estruc(x, \Gamma) =$

$$x$$

if $x \in Var$ and $\eta_\Gamma(x) = \perp$,

$estruc(x, \Gamma) = estruc(\mu x.e_x, \Gamma)$
if $x \in Var$ and $\eta_\Gamma(x) = \mu x.e_x$,

$estruc(b, \Gamma) =$

$$b$$

if $b \in Cons$,

$estruc(\emptyset, \Gamma) =$

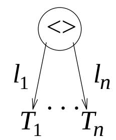
$$\{\}$$

$estruc(\{e_1, \cdots, e_n\}, \Gamma) =$

$$\begin{tikzcd} & \{\} \arrow[dl, "\in"'] \arrow[dr, "\in"] & \\ T_1 & \cdots & T_n \end{tikzcd}$$

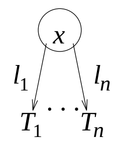
where $T_i = estruc(e_i, \Gamma)$,

$estruc(< l_1 = e_1, \cdots, l_n = e_n >) =$



where $T_i = estruc(e_i, \Gamma)$

$estruc(\mu\, x. < l_1 = e_1, \cdots, l_n = e_n >, \Gamma) =$



where $T_i = estruc(e_i, \Gamma \cup \{\mu\, x. < l_1 = e_1, \cdots, l_n = e_n >\})$.

For convenience, we sometimes write $estruc(e)$ instead of $estruc(e, \emptyset)$. □

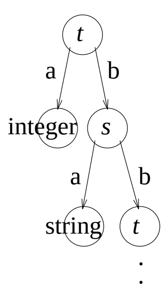
The instance relation between exact trees representing terms and exact trees representing types is defined as follows.

Definition 18. Let τ be a type. Then tree T is an instance of $estruc(\tau)$, denoted by $inst(T, estruc(\tau))$, if and only if there is an injective tree morphism from T to $estruc(\tau)$, such that for every node or arrow q in T the following holds:

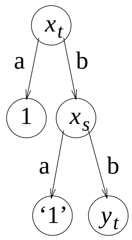
1. if $label(q) \in Cons_B$ for some $B \in BTypes$, then $label(\varphi(q)) = B$
2. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is a leaf, then $label(\varphi(q)) = t$
3. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is not a leaf, then $label(\varphi(q)) = t$ and $\mid children(q) \mid = \mid children(\varphi(q)) \mid$
4. otherwise, $label(\varphi(q)) = label(q)$.

□

Example 5. Let τ be type $\mu t. < \mathrm{a} : \mathrm{integer}, \mathrm{b} : \mu s. < \mathrm{a} : \mathrm{string}, \mathrm{b} : t >>$ and e be term $\mu x_t. < \mathrm{a} = 1, \mathrm{b} = \mu x_s. < \mathrm{a} = \text{‘1’}, \mathrm{b} = y_t >>$, where x_s is an element of Var_s, and x_t and y_t are elements of Var_t. The exact tree representing τ, denoted by $estruc(\tau)$, is given by:



The exact tree representing e, denoted by $estruc(e)$, is given by:



And, obviously, $estruc(e)$ is an instance of $estruc(\tau)$. $\square$

A subterm of a term is obtained by replacing every set in the term by a subset of cardinality 1.

Definition 19. Let e be a term. The set of subterms of e, denoted by $subterms(e)$, is defined as follows:

$$
\begin{array}{l}
subterms(x) = \{x\} \text{ if } x \in Var, \\
subterms(b) = \{b\} \text{ if } b \in Cons, \\
subterms(\emptyset) = \emptyset, \\
subterms(\{e_1, \cdots, e_n\}) = \\
\quad \{\{e'\} \mid e' \in subterms(e_1)\} \cup \cdots \cup \{\{e'\} \mid e' \in subterms(e_n)\}, \\
subterms(< l_1 = e_1, \cdots, l_n = e_n >) = \\
\quad \{< l_1 = e'_1, \cdots, l_n = e'_n > \mid \forall i \in \{1, \cdots, n\}\ [e'_i \in subterms(e_i)]\}, \\
subterms(\mu x.e) = \{\mu x.e' \mid e' \in subterms(e)\}.
\end{array}
$$

$\square$

Example 6. Let e be term $\mu x. < \mathrm{a} = \{1, 2\}, \mathrm{b} = \mu y. < \mathrm{a} = \{2, 5\}, \mathrm{b} = x >>$. The set of subterms of e is given by:

$$
\begin{array}{l}
\{\mu x. < \mathrm{a} = \{1\}, \mathrm{b} = \mu y. < \mathrm{a} = \{2\}, \mathrm{b} = x >> \\
\quad \mu x. < \mathrm{a} = \{1\}, \mathrm{b} = \mu y. < \mathrm{a} = \{5\}, \mathrm{b} = x >> \\
\quad \mu x. < \mathrm{a} = \{2\}, \mathrm{b} = \mu y. < \mathrm{a} = \{2\}, \mathrm{b} = x >> \\
\quad \mu x. < \mathrm{a} = \{2\}, \mathrm{b} = \mu y. < \mathrm{a} = \{5\}, \mathrm{b} = x >>\}.
\end{array}
$$

$\square$

Now we can define structural extensions.

Definition 20. Let τ be a type. The structural extension of τ, denoted by $struc\_ext(\tau)$, is defined as:

$$
\begin{array}{rl}
struc\_ext(\tau) = \{struc(e) \mid e \in Terms \wedge FV(e) \subseteq \{y \in Var_s \mid s \in fvars(\tau)\} \wedge \\
\forall e' \in subterms(e)[inst(estruc(e), estruc(\tau))]\}.
\end{array}
$$

$\square$

Note that the elements of the structural extension of a type contain sets of arbitrary cardinality. Finally, we prove Claim 3:

$$struc(\tau_1) = struc(\tau_2) \Leftrightarrow struc\_ext(\tau_1) = struc\_ext(\tau_2),$$

and Claim 4:

$$struc\_ext(\tau) = \{struc(e) \mid e \in ext(\tau)\}.$$

From these claims it follows that structural and extensional equivalence are logically equivalent:

$$
\begin{array}{l}
struc(\tau_1) = struc(\tau_2) \Leftrightarrow \\
\quad struc\_ext(\tau_1) = struc\_ext(\tau_2) \Leftrightarrow \\
\quad \{struc(e) \mid e \in ext(\tau_1)\} = \{struc(e) \mid e \in ext(\tau_2)\} \Leftrightarrow \\
\quad ext(\tau_1) \cong ext(\tau_2).
\end{array}
$$

3.4.1 Proof of Claim 3

We prove the following claim:

$$struc(\tau_1) = struc(\tau_2) \;\Leftrightarrow\; struc\_ext(\tau_1) = struc\_ext(\tau_2),$$

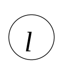
First, we define the projection of a term on (the tree of) a type. A term is projected on a type by unfolding the term and renaming instance variables until the resulting term matches the type exactly.

Definition 21. Let τ_1 and τ_2 be types, such that $struc(\tau_1) = struc(\tau_2)$. Furthermore, let g be an injective function from $bvars(\tau_1) \times TypeVar$ to $Var - fvars(\tau_1)$ and e be a term, such that $\forall e' \in subterms(e)[inst(estruc(e'), estruc(\tau_1))]$. The projection of e on $estruc(\tau_2)$ is defined as $proj(e, estruc(\tau_2), \emptyset)$, where $proj(e', U, V)$ is defined as follows:

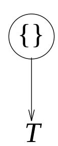
$$\begin{array}{l}
proj(x, node(t), V) = x \;\text{ if } x \in Var\\
proj(b, node(B), V) = b \;\text{ if } b \in Cons\\
proj(\emptyset, tree(\{T\}), V) = \emptyset\\
proj(\{e_1, \cdots, e_n\}, tree(\{T\}), V) = \{proj(e_1, T, V), \cdots, proj(e_n, T, V)\}\\
proj(x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = g(x, t)\\
\quad \text{if } g(x,t) \in V\\
proj(x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = proj(\mu x.e_x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V)\\
\quad \text{if } g(x,t) \notin V \text{ and } \eta_V(x) = \mu x.e_x\\
proj(\mu x. < l_1 = e_1, \cdots, l_n = e_n >, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = g(x, t)\\
\quad \text{if } g(x,t) \in V\\
proj(\mu x. < l_1 = e_1, \cdots, l_n = e_n >, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) =\\
\quad \mu\, g(x,t). < l_1 : proj(e_1, T_1, V \cup \{g(x,t), \mu x. < l_1 = e_1, \cdots, l_n = e_n >\}), \cdots,\\
\qquad\qquad\qquad l_n : proj(e_n, T_n, V \cup \{g(x,t), \mu x. < l_1 = e_1, \cdots, l_n = e_n >\}) >\\
\quad \text{if } g(x,t) \notin V,
\end{array}$$

where

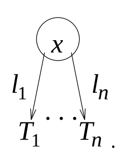
$node(l) =$



$tree(\{T\}) =$



$tree(x. < l_1 : T_1, \cdots, l_n : T_n >) =$



□

Example 7. Let τ be type $\mu t. < \mathrm{a} : \mathrm{integer}, \mathrm{c} : \mu s. < \mathrm{a} : \mathrm{integer}, \mathrm{b} : \mathrm{string}, \mathrm{c} : t >>$ and e be term $\mu x. < \mathrm{a} = 1, \mathrm{b} = \text{‘1’}, \mathrm{c} = x >$. The projection of e on $estruc(\tau)$ is given by:

$$\mu\, g(x,t). < \mathrm{a} = 1, \mathrm{c} = \mu\, g(x,s). < \mathrm{a} = 1, \mathrm{b} = \text{‘1’}, \mathrm{c} = g(x,t) >>.$$

□

Proof of $\Rightarrow$. Suppose $struc(\tau_1) = struc(\tau_2)$. Using Lemma 6, we can conclude that there is a bijective tree morphism from $estruc(\tau_1)$ to $estruc(\tau_2)$, such that for every node or arrow q in $estruc(\tau_1)$ the following holds:

1. if $label(q) = t$ for some $t \in TypeVar$ and q is a leaf, then $label(\varphi(q)) = t$,
2. if $label(q) = t$ for some $t \in TypeVar$ and q is not a leaf, then $label(\varphi(q)) = s$ for some $s \in TypeVar$,
3. otherwise, $label(\varphi(q)) = label(q)$.

Now, let $struc(e)$ be an element of $struc\_ext(\tau_1)$ and e' be an element of $subterms(e)$. Furthermore, let $P(e')$ be $proj(e', estruc(\tau_2), \emptyset)$. Then $inst(estruc(e'), estruc(\tau_1))$, because e' is an element of $subterms(e)$ and $struc(e)$ is an element of $struc\_ext(\tau_1)$. From the definition of *proj* it follows that there is an injective tree morphism from $estruc(P(e'))$ to $estruc(\tau_2)$, such that for every node or arrow q in $estruc(P(e'))$ the following holds:

1. if $label(q) \in Cons_B$ for some $B \in BTypes$, then $label(\varphi(q)) = B$
2. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is a leaf, then $label(\varphi(q)) = t$
3. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is not a leaf, then $label(\varphi(q)) = t$ and $\mid children(q) \mid = \mid children(\varphi(q)) \mid$
4. otherwise, $label(\varphi(q)) = label(q)$.

That is, $inst(estruc(P(e')), estruc(\tau_2))$. Hence, for every $e' \in subterms(e)$, we have:

$$inst(estruc(P(e')), estruc(\tau_2))].$$

Let $P(e)$ be $proj(e, estruc(\tau_2), \emptyset)$. Using the definition of *proj*, we can conclude that $struc(P(e)) = struc(e)$ and:

$$\forall e'' \in subterms(P(e))[inst(estruc(e''), estruc(\tau_2))],$$

because $e'' \in subterms(P(e)) \Leftrightarrow (e'' = P(e') \wedge e' \in subterms(e))$. Since $FV(e) = FV(P(e))$, we have $struc(e) = struc(P(e)) \in struc\_ext(\tau_2)$.
Proof of $\Leftarrow$. Suppose $struc\_ext(\tau_1) = struc\_ext(\tau_2)$. Then there are terms e_1 and e_2, such that $inst(estruc(e_1), estruc(\tau_1))$, $inst(estruc(e_2), estruc(\tau_2))$, and $struc(e_1) = struc(e_2)$. From Definition 18 and Lemma 6, it follows that $struc(\tau_1) = struc(\tau_2)$.

3.4.2 Proof of Claim 4

It suffices to prove the following claim:

$$e \in terms(\tau) \;\Leftrightarrow\; (e \in Terms \wedge \forall e' \in subterms(e)\;[inst(estruc(e'), estruc(\tau))]).$$

The proof is an induction argument on the structure of τ. Let τ be basic type B. Since $inst(estruc(e), estruc(B)) \Leftrightarrow e \in Cons_B$, and, for $b \in Cons_B$, $subterms(b) = \{b\}$, we have:

$$b \in terms(B) \;\Leftrightarrow\; (b \in Terms \wedge \forall b' \in subterms(b)\;[inst(estruc(b'), estruc(B))]).$$

Let τ be type variable t. Since $inst(estruc(e), estruc(t)) \Leftrightarrow e \in Var_t$, and, for $x \in Var_t$, $subterms(x) = \{x\}$, we have:

$$x \in terms(t) \;\Leftrightarrow\; (x \in Terms \wedge \forall x' \in subterms(x)\;[inst(estruc(x'), estruc(t))]).$$

Let τ be set type $\{\tau_1\}$. Apply the induction hypothesis to τ_1 and use $subterms(\{e_1, \cdots, e_n\}) = subterms(\{e_1\}) \cup \cdots \cup subterms(\{e_n\})$.
Let τ be record type $< l_1 : \tau_1, \cdots, l_n : \tau_n >$. Apply the induction hypothesis to τ_i, for $i \in \{1, \cdots, n\}$.
Let τ be recursive type $\mu t.\alpha$. The proof of $\Rightarrow$ is an induction argument on the structure of term $e \in terms(\tau)$.
Base step: $e = x \in Var_t$. Then:

$$\forall x' \in subterms(x)\ [inst(estruc(x'), estruc(\mu t.\alpha))].$$

Induction step: $e = \mu x.(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n])$, such that $e_0 \in terms(\alpha)$ and $e_i \in terms(\mu t.\alpha)$, for $i \in \{1, \cdots, n\}$. Applying the first induction hypothesis to e_0 and the second induction hypothesis to e_1 through e_n, gives us:

$$\forall e' \in subterms(e_0)\ [inst(estruc(e'), estruc(\alpha))]$$
$$\forall i \in \{1, \cdots, n\}\ \forall e' \in subterms(e_i)\ [inst(estruc(e'), estruc(\mu t.\alpha))].$$

Let R_t be the transformation on trees that replaces the label of the root by t. Since

$$subterms(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n]) = \\ \{e'_0[x_1 \setminus e'_1, \cdots, x_n \setminus e'_n] \mid e'_0 \in subterms(e_0) \wedge \forall i \in \{1, \cdots, n\}\ [e'_i \in subterms(e_i)]\},$$

and $R_t(estruc(\alpha))[t \setminus estruc(\mu t.\alpha)] = estruc(\mu t.\alpha)$, we have:

$$\forall e' \in subterms(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n])\ [inst(estruc(e'), estruc(\alpha)[t \setminus estruc(\mu t.\alpha)])]$$

and:

$$\forall e' \in subterms(\mu x.(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n]))\ [inst(estruc(e'), estruc(\mu t.\alpha))]).$$

The proof of $\Leftarrow$ is an induction argument on the number of bound instance variables from Var_t in $p\_struc(e)$.
Base step: $estruc(e)$ has no bound instance variables from Var_t. Then $e = x \in Var_t$ and, hence, $e \in terms(\mu t.\alpha)$.
Induction step: $estruc(e)$ has $j + 1$ bound instance variables from Var_t. Then $e = \mu x.e_x$ and:

$$\forall \mu x.e' \in subterms(\mu x.e_x)\ [inst(estruc(\mu x.e'), estruc(\mu t.\alpha))].$$

Let R_x be the transformation on trees that replaces the label of the root by x. Since $estruc(\mu x.e') = R_x(estruc(e'))[x \setminus estruc(\mu x.e')]$ and $estruc(\mu t.\alpha) = R_t(estruc(\alpha))[t \setminus estruc(\mu t.\alpha)]$, we have:

$$\forall e' \in subterms(e_x)\ [inst(estruc(e'), estruc(\alpha)[t \setminus estruc(\mu t.\alpha)]).$$

In fact, we have: $e_x = e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n]$, where $\{x_1, \cdots, x_n\} \subset Var_t$, $\{e_0, \cdots, e_n\} \subset Terms$, and:

$$\forall e' \in subterms(e_0)\ [inst(estruc(e'), estruc(\alpha))]$$
$$\forall i \in \{1, \cdots, n\} \forall e' \in subterms(e_i)\ [inst(estruc(e'), estruc(\mu t.\alpha))].$$

Applying the first induction hypothesis to e_0 and the second induction hypothesis to e_1 through e_n (every $estruc(e_i)$ has at most j bound instance variables from Var_t), gives us:

$$e_0 \in terms(\alpha)\ \wedge\ \forall i \in \{1, \cdots, n\}[e_i \in terms(\mu t.\alpha)].$$

Hence, $e = \mu x.e_x = \mu x.(e_0[x_1 \setminus e_1, \cdots, x_n \setminus e_n]) \in terms(\mu t.\alpha)$.

3.5 Decidability of structural and extensional equivalence

In this subsection, we prove Theorem 3. In fact, we prove that there is a decision procedure for derivable equivalence. Since $\tau_1 \cong_D \tau_2 \Leftrightarrow \tau_1 \cong_{struc} \tau_2 \Leftrightarrow \tau_1 \cong_{ext} \tau_2$, there is a decision procedure for structural and extensional equivalence.

Suppose τ_1 and τ_2 are closed μ-complete types, and we want to know whether $\tau_1 \cong_D \tau_2$ or $\tau_1 \not\cong_D \tau_2$. Then we try to derive $\emptyset \vdash \tau_1 \cong \tau_2$ bottom up, by starting from $\emptyset \vdash \tau_1 \cong \tau_2$ and by applying the rules of derivation system DE until no rules can be applied any more. For every formula of the form $\Gamma \vdash \tau \cong \sigma$, at most one rule can be applied. Hence, the derivation process is deterministic: there is at most one derivation.

Rule 4, 5, and 6 can only be applied a finite number of times consecutively, because they decrease the complexity of the types. Rule 1, 2, and 3 can only be applied to a formula $\Gamma \vdash t \cong s$ (resp., $\Gamma \vdash t \cong \mu s.\beta$ and $\Gamma \vdash \mu t.\alpha \cong s$) if (t, s) is not an element of Γ. However, after one of these rules has been applied, rule 6 will be applied, adding (t, s) to Γ. This means that neither rule 1, 2, or 3 can be applied more than once for the same pair of type variables (t, s) on a path from the root to a leaf. Since there are only finitely many type variables in τ_1 and τ_2, rule 1, 2, and 3 can only be applied a finite number of times.

From these observations it follows that every rule can only be applied a finite number of times, resulting in a partial, but finite, derivation tree. If all leaves of the partial derivation tree are axioms, then there is a derivation of $\emptyset \vdash \tau_1 \cong \tau_2$ and, hence, we know: $\tau_1 \cong_D \tau_2$. If not all leaves of the partial derivation tree are axioms, then there is no derivation of $\emptyset \vdash \tau_1 \cong \tau_2$ and, hence, we know: $\tau_1 \not\cong_D \tau_2$.

3.6 Normalisation

In this subsection, we define a normalisation process for types and prove that derivable equivalence is logically equivalent to the equivalence relation induced by the normalisation process.
First, we define a reduction process for types using folding of types.

Definition 22. Let τ be a μ-complete type. The reduced form of τ is defined as $rd(\tau, \emptyset)$, where $rd(\tau', \Gamma)$ is defined as follows:

$$\begin{array}{ll}
rd(t, \Gamma) = t & \text{if } t \in \mathit{TypeVar} \\
rd(B, \Gamma) = B & \text{if } B \in \mathit{BTypes} \\
rd(\{\upsilon\}, \Gamma) = \{rd(\upsilon)\} & \\
rd(< l_1 : \upsilon_1, \cdots, l_n : \tau_n >, \Gamma) = & \\
\quad < l_1 : rd(\upsilon_1, \Gamma), \cdots, l_n : rd(\upsilon_n, \Gamma) > & \\
rd(\mu t.\alpha, \Gamma) = s & \text{if } \exists \mu s.\beta \in \Gamma \ [struc(\mu t.\alpha, \Gamma) = struc(\mu s.\beta, \Gamma)] \\
rd(\mu t.\alpha, \Gamma) = \mu t.(rd(\alpha, \Gamma \cup \{\mu t.\alpha\})) & \text{if } \forall \mu s.\beta \in \Gamma \ [struc(\mu t.\alpha, \Gamma) \neq struc(\mu s.\beta, \Gamma)].
\end{array}$$

For convenience, we sometimes write $rd(\tau)$ instead of $rd(\tau, \emptyset)$. □

Since every type has exactly one reduced form, the reduction process is a normalisation process. Furthermore, the reduction process induces an equivalence relation on types.

Definition 23. Let τ_1 and τ_2 be closed μ-complete types. Reducible equivalence of τ_1 and τ_2, denoted by $\tau_1 \cong_R \tau_2$, is defined as follows:

$$\tau_1 \cong_R \tau_2 \Leftrightarrow rd(\tau_1) =_D rd(\tau_2).$$

□

Finally, we prove that derivable equivalence is logically equivalent to the equivalence relation induced by the reduction process.

Theorem 4. Let τ_1 and τ_2 be closed μ-complete types. Then:

$$\tau_1 \cong_D \tau_2 \Leftrightarrow \tau_1 \cong_R \tau_2.$$

Proof of $\Rightarrow$. Suppose $\tau_1 \cong_D \tau_2$. Then $struc(\tau_1) = struc(\tau_2)$. Hence, there is a bijective tree morphism from $estruc(\tau_1)$ to $estruc(\tau_2)$, such that for every node or arrow q in $estruc(\tau_1)$ the following holds:

1. if $label(q) = t$ for some $t \in TypeVar$, then $label(\varphi(q)) = s$ for some $s \in TypeVar$,
2. otherwise, $label(\varphi(q)) = label(q)$.

From the definition of rd, it follows that if τ is a type, φ is the bijective tree homomorphism from $estruc(rd(\tau))$ to $struc(rd(\tau))$ as given by Lemma 6, n_1 and n_2 are nodes on the same path starting at the root of $estruc(rd(\tau))$, and the tree starting at $\varphi(n_1)$ is equal to the tree starting at $\varphi(n_2)$, then the label of n_1 is equal to the label of n_2. Hence, there is a bijective tree morphism from $estruc(rd(\tau_1))$ to $estruc(rd(\tau_2))$ and a bijective function f from $bvars(rd(\tau_1))$ to $bvars(rd(\tau_2))$, such that for every node or arrow q in $estruc(\tau_1)$ the following holds:

1. if $label(q) = t$ for some $t \in TypeVar$, then $label(\varphi(q)) = f(t)$
2. otherwise, $label(\varphi(q)) = label(q)$.

That is, $rd(\tau_1)$ and $rd(\tau_2)$ are equal, modulo renaming of type variables. Hence, $rd(\tau_1) =_D rd(\tau_2)$.
Proof of $\Leftarrow$. Suppose $rd(\tau_1) =_D rd(\tau_2)$. That is, $rd(\tau_1)$ and $rd(\tau_2)$ are equal, modulo renaming of type variables. Then:

$$struc(\tau_1) = struc(rd(\tau_1)) = struc(rd(\tau_2)) = struc(\tau_2)$$

Hence, $\tau_1 \cong_D \tau_2$. □

4 Subtyping

In this section, we define structural, extensional, and derivable subtyping. We prove that derivable subtyping is sound and complete w.r.t. structural and extensional subtyping and that structural and extensional subtyping are decidable.

Similar to type equivalence, trees and extensions can be used to define two notions of semantic subtyping. Structural subtyping is defined as a supertree relation on the trees of the types.

Definition 24. Let τ_1 and τ_2 be types. Structural subtyping, where $\tau_1 \preceq_{struc} \tau_2$ denotes that τ_1 is a structural subtype of τ_2, is defined by:

$$\tau_1 \preceq_{struc} \tau_2 \;\Leftrightarrow\; struc(\tau_1) \sqsupseteq struc(\tau_2).$$

□

Extensional subtyping is defined as a 'set of subterms' relation on the extensions of the types.

Definition 25. Let τ_1 and τ_2 be types. Extensional subtyping, where $\tau_1 \preceq_{ext} \tau_2$ denotes that τ_1 is an extensional subtype of τ_2, is defined by:

$$\tau_1 \preceq_{ext} \tau_2 \;\Leftrightarrow\; ext(\tau_1) \preceq ext(\tau_2).$$

□

4.1 Derivation system for subtyping

In this subsection, we introduce a derivation system for subtyping. Again, a derivation is a tree of formulas, where the children formulas imply the parent formula. A formula is of the form $\Gamma \vdash \tau \preceq \sigma$ (where τ and σ are types, $\preceq$ is the subtype relation, and Γ is a context), saying that $\tau \preceq \sigma$ follows from the axioms for basic types and the premises in Γ. A context Γ is a triple $(\Gamma_l, \Gamma_r, \Gamma_p)$, similar to a context in the derivation system for type equivalence.

Definition 26. The derivation system for subtyping, denoted by *DS*, is defined as follows. The axioms of the derivation system are:

1.	$\Gamma \vdash B \preceq B$	if $B \in BTypes$
2.	$\Gamma \vdash t \preceq s$	if $(t, s) \in \Gamma_p$
3.	$\Gamma \vdash t \preceq \mu s.\beta$	if $(t, s) \in \Gamma_p \wedge \mu s.\beta \in \Gamma_r$
4.	$\Gamma \vdash \mu t.\alpha \preceq s$	if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$
5.	$\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$	if $(t, s) \in \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$.

The rules of the derivation system are:

1. $\dfrac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash t \preceq s}$ if $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$

2. $\dfrac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash t \preceq \mu s.\beta}$ if $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$

3. $\dfrac{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}{\Gamma \vdash \mu t.\alpha \preceq s}$ if $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$

4. $\dfrac{\Gamma \vdash \tau \preceq \sigma}{\Gamma \vdash \{\tau\} \preceq \{\sigma\}}$

5. $\dfrac{\Gamma \vdash \tau_1 \preceq \sigma_1, \cdots, \Gamma \vdash \tau_n \preceq \sigma_n}{\Gamma \vdash < l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots, l_{n+m} : \tau_{n+m} > \preceq < l_1 : \sigma_1, \cdots, l_n : \sigma_n >}$

6. $\dfrac{\Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\} \vdash \alpha \preceq \beta}{\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta}$ if $(t, s) \notin \Gamma_p$.

□

The set of axioms and rules of *DS* is the same as the extended set of rules for $<_A$ from [2] and an extension of the well-known subtype rules from [5] with rules for recursive types. Derivable subtyping of μ-complete types is given by the following definition.

Definition 27. Let τ_1 and τ_2 be closed μ-complete types. Derivable subtyping, where $\tau_1 \preceq_D \tau_2$ denotes that τ_1 is a subtype of τ_2 according to derivation system *DS*, is defined by:

$$\tau_1 \preceq_D \tau_2 \;\Leftrightarrow\; \emptyset \vdash_{DS} \tau_1 \preceq \tau_2,$$

where $\Gamma \vdash_{DS} \tau \cong \sigma$ means that there is a derivation in *DS* with conclusion $\Gamma \vdash \tau \preceq \sigma$. □

Next, we give an example of a derivation.

Example 8. For convenience, we define a number of abbreviations:

$\alpha = < a_1 : B, a_2 : t, a_3 : t, a_4 : t >$
$\beta = < a_1 : B, a_2 : s, a_3 : \mu s'.\beta' >$
$\beta' = < a_1 : B, a_2 : s' >$
$\Gamma_1 = \{(\mu t.\alpha, \mu s.\beta), (t, s)\}$
$\Gamma_2 = \Gamma_1 \cup \{(\mu t.\alpha, \mu s'.\beta'), (t, s')\}$.

Using derivation system *DS*, we obtain the following derivation for $\emptyset \vdash \mu t.\alpha \preceq \mu s.\beta$:

$$\cfrac{\cfrac{\cfrac{\Gamma_2 \vdash B \preceq B,\ \Gamma_2 \vdash t \preceq s'}{\Gamma_2 \vdash \alpha \preceq \beta'}\ \text{(rule 5)}}{\Gamma_1 \vdash \mu t.\alpha \preceq \mu s'.\beta'}\ \text{(rule 6)}}{\qquad}\ \text{(rule 2)}$$

$$\cfrac{\cfrac{\Gamma_1 \vdash B \preceq B,\ \Gamma_1 \vdash t \preceq s,\quad \Gamma_1 \vdash t \preceq \mu s'.\beta'}{\Gamma_1 \vdash \alpha \preceq \beta}\ \text{(rule 5)}}{\emptyset \vdash \mu t.\alpha \preceq \mu s.\beta}\ \text{(rule 6)}$$

□

In the sequel, we will prove the following theorems.

Theorem 5. Derivable subtyping is sound and complete w.r.t. structural subtyping. □

Theorem 6. Structural and extensional subtyping are logically equivalent. □

Theorem 7. Structural and extensional subtyping are decidable. □

Using Theorem 5 and 6, we can deduce the following corollary.

Corollary 2. Derivable subtyping is sound and complete w.r.t. extensional subtyping. □

Finally, using Theorem 1 and 5, we can deduce that derivable equivalence implies derivable subtyping and that derivable subtyping is antisymmetric w.r.t. derivable equivalence.

Lemma 7. Let τ_1 and τ_2 be closed μ-complete types. Then:

$$\tau_1 \cong_D \tau_2 \;\Leftrightarrow\; (\tau_1 \preceq_D \tau_2 \wedge \tau_2 \preceq_D \tau_1).$$

Proof.

$$\begin{aligned}
&\tau_1 \cong_D \tau_2 \Leftrightarrow \\
&\quad \tau_1 \cong_{struc} \tau_2 \Leftrightarrow \\
&\quad struc(\tau_1) = struc(\tau_2) \Leftrightarrow \\
&\quad (struc(\tau_1) \sqsupseteq struc(\tau_2) \wedge struc(\tau_2) \sqsupseteq struc(\tau_1)) \Leftrightarrow \\
&\quad (\tau_1 \preceq_{struc} \tau_2 \wedge \tau_2 \preceq_{struc} \tau_1) \Leftrightarrow \\
&\quad \tau_1 \preceq_D \tau_2 \wedge \tau_2 \preceq_D \tau_1.
\end{aligned}$$

□

4.2 Soundness w.r.t structural subtyping

In this subsection, we prove the soundness part of Theorem 5. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

$$\tau_1 \preceq_D \tau_2 \;\Rightarrow\; \tau_1 \preceq_{struc} \tau_2.$$

The proof is similar to the proof of the soundness part of Theorem 1. First, for every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation of $\emptyset \vdash \tau_1 \preceq \tau_2$, an l-tree and an r-tree are constructed (the l-tree will be proven to be a supertree of the r-tree). The l-tree is constructed in such a way that it is equal to $struc(\tau, \Gamma_l)$ and the r-tree is constructed in such a way that it is equal to $struc(\sigma, \Gamma_r)$, except for the free type variables. Following the derivation, constructing the tree for a formula from the trees for its children formulas, we obtain an l-tree that is equal to $struc(\tau_1, \emptyset)$ and an r-tree that is equal to $struc(\tau_2, \emptyset)$ (because τ_1 and τ_2 have no free type variables).

The l-tree and the r-tree are exactly the same as the tree in the proof of soundness w.r.t. structural equivalence, except the l-tree and the r-tree for record types.

Definition 28. Let τ_1 and τ_2 be closed μ-complete types. Furthermore, let f be an injective function from $bvars(\tau_1) \times bvars(\tau_2)$ to $TypeVar - (bvars(\tau_1) \cup bvars(\tau_2))$ and $\Gamma \vdash \tau \preceq \sigma$ be a formula in the derivation of $\emptyset \vdash \tau_1 \preceq \tau_2$. The l-tree for $\Gamma \vdash \tau \preceq \sigma$, denoted by $tree_l(\tau, \sigma, \Gamma)$ and the r-tree $\Gamma \vdash \tau \preceq \sigma$, denoted by $tree_r(\tau, \sigma, \Gamma)$, are defined as follows:

$tree_j(B, B, \Gamma) =$

$$B$$

if $B \in BTypes$

$tree_j(\{\tau_1\}, \{\sigma_1\}, \Gamma) =$

$$\begin{tikzcd} \{\} \arrow[d] \\ T \end{tikzcd}$$

where $T = tree_j(\tau_1, \sigma_1, \Gamma)$

$tree_l(< l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots, l_{n+m} : \tau_{n+m} >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma) =$

$$\begin{tikzcd} & <> \arrow[dl, "l_1"'] \arrow[dr, "l_{n+m}"] & \\ T_1 & \cdots & T_{n+m} \end{tikzcd}$$

where $T_i = tree_l(\tau_i, \sigma_i, \Gamma)$ for $i \in \{1, \cdots, n\}$
and $T_i = struc(\tau_i, \Gamma_l)$ for $i \in \{n+1, \cdots, n+m\}$

$tree_r(< l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots, l_{n+m} : \tau_{n+m} >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma) =$

$$\begin{tikzcd} & <> \arrow[dl, "l_1"'] \arrow[dr, "l_n"] & \\ T_1 & \cdots & T_n \end{tikzcd}$$

where $T_i = tree_r(\tau_i, \sigma_i, \Gamma)$ for $i \in \{1, \cdots, n\}$
$tree_j(t, s, \Gamma) = tree_j(t, \mu s.\beta, \Gamma) = tree_j(\mu t.\alpha, s, \Gamma) = tree_j(\mu t.\alpha, \mu s.\beta, \Gamma) =$

$$f(t,s)$$

if $(t, s) \in \Gamma_p$

$tree_j(t, s, \Gamma) = tree_j(\mu t.\alpha, \mu s.\beta, \Gamma)$
if $(t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$

$tree_j(t, \mu s.\beta, \Gamma) = tree_j(\mu t.\alpha, \mu s.\beta, \Gamma)$
if $(t, s) \notin \Gamma_p \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha$

$tree_j(\mu t.\alpha, s, \Gamma) = tree_j(\mu t.\alpha, \mu s.\beta, \Gamma)$
if $(t, s) \notin \Gamma_p \wedge \eta_{\Gamma_r}(s) = \mu s.\beta$

$tree_j(\mu t.\alpha, \mu s.\beta, \Gamma) = tree_j(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\})$
if $(t, s) \notin \Gamma_p$,

where $j \in \{l, r\}$. $\Box$

Finally, we prove soundness in two steps. First, we prove Claim 5:

$$\emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \;\Rightarrow\; tree_l(\tau_1, \tau_2, \emptyset) \sqsupseteq tree_r(\tau_2, \tau_1, \emptyset),$$

and second, we prove Claim 6:

$$\emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \;\Rightarrow\; (tree_l(\tau_1, \tau_2, \emptyset) = struc(\tau_1, \emptyset) \wedge tree_r(\tau_1, \tau_2, \emptyset) = struc(\tau_2, \emptyset)).$$

From these claims it follows that derivable subtyping is sound w.r.t. structural subtyping:

$$\emptyset \vdash_{DS} \tau_1 \preceq \tau_2 \;\Rightarrow\; struc(\tau_1) \sqsupseteq struc(\tau_2).$$

4.2.1 Proof of Claim 5

It suffices to prove the following claim; for every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \preceq \tau_2$:

$$tree_l(\tau, \sigma, \Gamma) \sqsupseteq tree_r(\sigma, \tau, \Gamma).$$

The proof is an induction argument on the distance of a formula in the derivation tree to its remotest descendant. Base step: the formula is an axiom.
Axiom 1: $\Gamma \vdash B \preceq B$. Then, obviously, $tree_l(B, B, \Gamma) \sqsupseteq tree_r(B, B, \Gamma)$.
Axiom 2: $\Gamma \vdash t \preceq s$. Then (t, s) must be an element of Γ. Hence, $tree_l(t, s, \Gamma) \sqsupseteq tree_r(t, s, \Gamma)$.
Axiom 3, 4, or 5: $\Gamma \vdash t \preceq \mu s.\beta$, $\Gamma \vdash \mu t.\alpha \preceq s$, or $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$. Similar to the previous case.
Induction step: the formula is the result of applying a rule to a number of formulas which are closer to their remotest descendant.
Rule 1: $\Gamma \vdash t \preceq s$. Then there must be types $\eta_{\Gamma_l}(t) = \mu t.\alpha$ and $\eta_{\Gamma_r}(s) = \mu s.\beta$, such that:

$$\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta.$$

Using the induction hypothesis, we can conclude: $tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) \sqsupseteq tree_r(\mu t.\alpha, \mu s.\beta, \Gamma)$. Since (t, s) is not an element of Γ, it follows that:

$$tree_l(t, s, \Gamma) = tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) \sqsupseteq tree_r(\mu t.\alpha, \mu s.\beta, \Gamma) = tree_r(t, s, \Gamma).$$

Rule 2 and 3: $\Gamma \vdash t \preceq \mu s.\beta$ or $\Gamma \vdash \mu t.\alpha \preceq s$. Similar to the previous case.
Rule 4: $\Gamma \vdash \{\tau\} \preceq \{\sigma\}$. Apply the induction hypothesis to $\Gamma \vdash \tau \preceq \sigma$.
Rule 5: $\Gamma \vdash < l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots, l_{n+m} : \tau_{n+m} > \preceq < l_1 : \sigma_1, \cdots, l_n : \sigma_n >$. Apply the induction hypothesis to $\vdash \tau_i \preceq \sigma_i$.
Rule 6: $\Gamma \vdash \mu t.\alpha \preceq \mu s.\beta$. Let Γ' be $\Gamma \cup \{(\mu t.\alpha, \mu s.\beta)\}$ and Δ be $\Gamma' \cup \{(t, s)\}$. From $\Delta \vdash \alpha \preceq \beta$ and the induction hypothesis, it follows that:

a) $tree_l(\alpha, \beta, \Delta) \sqsupseteq tree_r(\alpha, \beta, \Delta)$.

For every natural number i and every $j \in \{l, r\}$, define $tree_{j,i}(\alpha, \beta, \Delta)$ as follows:

$$tree_{j,1}(\alpha, \beta, \Delta) = tree_j(\alpha, \beta, \Delta),$$
$$tree_{j,i+1}(\alpha, \beta, \Delta) = tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{j,i}(\alpha, \beta, \Delta)].$$

Using an induction argument on i, where the base step follows from a) and the induction step is:

$$\begin{aligned} tree_{l,i+1}(\alpha, \beta, \Delta) &= tree_l(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{l,i}(\alpha, \beta, \Delta)] \sqsupseteq \\ &\quad tree_r(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{l,i}(\alpha, \beta, \Delta)] \sqsupseteq \\ &\quad tree_r(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{r,i}(\alpha, \beta, \Delta)] = \\ &\quad tree_{r,i+1}(\alpha, \beta, \Delta), \end{aligned}$$

we can conclude that, for every natural number i:

b) $tree_{l,i}(\alpha, \beta, \Delta) \sqsupseteq tree_{r,i}(\alpha, \beta, \Delta)$.

Furthermore, from an induction argument on the distance of a formula to its remotest descendant in the derivation tree for $\Delta \vdash \alpha \preceq \beta$, it follows that:

c) $tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] = tree_j(\alpha, \beta, \Gamma')$.

Again, using an induction argument on i, where the base step follows from c) and the induction step is:

$$\begin{aligned}
&tree_{j,i+1}(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] = \\
&\quad (tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus tree_{j,i}(\alpha, \beta, \Delta)])[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] = \\
&\quad tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus (tree_{j,i}(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')])] = \\
&\quad tree_j(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] = \\
&\quad tree_j(\alpha, \beta, \Gamma'),
\end{aligned}$$

we can conclude that, for every natural number i:

$$tree_{j,i}(\alpha, \beta, \Delta)[f(t, s) \setminus tree_j(\alpha, \beta, \Gamma')] = tree_j(\alpha, \beta, \Gamma').$$

This means that $tree_{j,i}(\alpha, \beta, \Delta)$ is equal to $tree_j(\alpha, \beta, \Gamma')$ from the root to at least depth i. From b) it follows that $tree_l(\alpha, \beta, \Gamma')$ is a supertree of $tree_r(\alpha, \beta, \Gamma')$. Hence:

$$tree_l(\mu t.\alpha, \mu s.\beta, \Gamma) = tree_l(\alpha, \beta, \Gamma') \sqsupseteq tree_r(\alpha, \beta, \Gamma') = tree_r(\mu t.\alpha, \mu s.\beta, \Gamma).$$

4.2.2 Proof of Claim 6

In the same way as Claim 1 was proven, we can prove the following claim; for every formula $\Gamma \vdash \tau \preceq \sigma$ in the derivation tree of $\emptyset \vdash \tau_1 \preceq \tau_2$:

a) $tree_l(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu t'.\alpha', \Gamma_l) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_l}(t') = \mu t'.\alpha'] = struc(\tau, \Gamma_l)$
b) $tree_r(\tau, \sigma, \Gamma)[f(t', s') \setminus struc(\mu s'.\beta', \Gamma_r) \mid (t', s') \in \Gamma_p \wedge \eta_{\Gamma_r}(s') = \mu s'.\beta'] = struc(\sigma, \Gamma_r)$.

4.3 Completeness w.r.t. structural subtyping

In this subsection, we prove the completeness part of Theorem 5. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

$$\tau_1 \preceq_{struc} \tau_2 \;\Rightarrow\; \tau_1 \preceq_D \tau_2.$$

The proof is similar to the proof of the completeness part of Theorem 1. First, a structural subtyping tree for τ_1 and τ_2 is constructed (the structural subtyping tree will be proven isomorphic to the derivation tree with conclusion $\emptyset \vdash \tau_1 \preceq \tau_2$). The tree is constructed in such a way that every node is labeled by a tuple of the form (τ, σ, Γ), where τ and σ are obtained by using the structure of τ_1 and τ_2 (and, if necessary, by unfolding of types), such that $struc(\tau, \Gamma_l) \sqsupseteq struc(\sigma, \Gamma_r)$. For example, the root is labeled $(\tau_1, \tau_2, \emptyset)$.
For the definition of structural subtyping trees, we need the following Lemma.

Lemma 8. Let τ and σ be μ-complete types, and Γ be a context, such that $struc(\tau, \Gamma_l) \sqsupseteq struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Then:

1. $\tau = B \;\Rightarrow\; \sigma = B$
2. $\tau = \{\tau_1\} \;\Rightarrow\; (\sigma = \{\sigma_1\} \wedge struc(\tau_1, \Gamma_l) \sqsupseteq struc(\sigma_1, \Gamma_r))$
3. $\tau = < l_1 : \tau_1, \cdots, l_n : \tau_n > \Rightarrow$
   $(\sigma = < l_1 : \sigma_1, \cdots, l_n : \sigma_n > \wedge\, n \geq m \wedge struc(\tau_i, \Gamma_l) \sqsupseteq struc(\sigma_i, \Gamma_r))$
4. $\tau = t \;\Rightarrow$
   $(\sigma = s \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge \eta_{\Gamma_r}(s) = \mu s.\beta \wedge struc(\alpha, \Gamma_l) \sqsupseteq struc(\beta, \Gamma_r)) \vee$
   $(\sigma = \mu s.\beta \wedge \eta_{\Gamma_l}(t) = \mu t.\alpha \wedge struc(\alpha, \Gamma_l) \sqsupseteq struc(\beta, \Gamma_r \cup \{\sigma\}))$
5. $\tau = \mu t.\alpha \;\Rightarrow$
   $(\sigma = s \wedge \eta_{\Gamma_r}(s) = \mu s.\beta \wedge struc(\alpha, \Gamma_l \cup \{\mu t.\alpha\}) \sqsupseteq struc(\beta, \Gamma_r)) \vee$
   $(\sigma = \mu s.\beta \wedge struc(\alpha, \Gamma_l \cup \{\mu t.\alpha\}) \sqsupseteq struc(\beta, \Gamma_r \cup \{\sigma\}))$.

Proof. The lemma follows from Definition 3. □

Now, we can define the children of a node in a structural subtyping tree.

Definition 29. Let τ and σ be μ-complete types, and Γ be a context, such that $struc(\tau, \Gamma_l) \sqsupseteq struc(\sigma, \Gamma_r)$ and $struc(\tau, \Gamma_l)$ contains no type variables. Furthermore, let x be (τ, σ, Γ). According to Lemma 8, there are 5 cases for x, of which the last two cases each have two subcases. For the definition of the children of x, we divide each subcase into two new subcases (one for $(t, s) \in \Gamma_p$ and one for $(t, s) \notin \Gamma_p$), obtaining 11 cases for x. The set of children of x, denoted by $stchildren(x)$ is defined as follows:

1. if $x = (B, B, \Gamma)$, then $stchildren(x) = \emptyset$
2. if $x = (t, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
3. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
4. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
5. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \in \Gamma_p$, then $stchildren(x) = \emptyset$
6. if $x = (\{\tau\}, \{\sigma\}, \Gamma)$, then $stchildren(x) = \{(\tau, \sigma, \Gamma)\}$
7. if $x = (< l_1 : \tau_1, \cdots, l_{n+m} : \tau_{n+m} >, < l_1 : \sigma_1, \cdots, l_n : \sigma_n >, \Gamma)$,
   then $stchildren(x) = \{(\tau_1, \sigma_1, \Gamma), \cdots, (\tau_n, \sigma_n, \Gamma)\}$
8. if $x = (t, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l \wedge \mu s.\beta \in \Gamma_r$,
   then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
9. if $x = (t, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu t.\alpha \in \Gamma_l$,
   then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
10. if $x = (\mu t.\alpha, s, \Gamma)$ and $(t, s) \notin \Gamma_p \wedge \mu s.\beta \in \Gamma_r$,
    then $stchildren(x) = \{(\mu t.\alpha, \mu s.\beta, \Gamma)\}$
11. if $x = (\mu t.\alpha, \mu s.\beta, \Gamma)$ and $(t, s) \notin \Gamma\_p$,
    then $stchildren(x) = \{(\alpha, \beta, \Gamma \cup \{(\mu t.\alpha, \mu s.\beta), (t, s)\})\}$.

□

The following lemma states that, if a tuple satisfies the precondition of Definition 29, then so do its children.

Lemma 9. Let $x = (\tau, \sigma, \Gamma)$ be a tuple, such that τ and σ are μ-complete types, Γ is a context, $struc(\tau, \Gamma_l) \sqsupseteq struc(\sigma, \Gamma_r)$, and $struc(\tau, \Gamma_l)$ contains no type variables. Then every element of $stchildren(x)$ is a tuple $x' = (\tau', \sigma', \Gamma')$, such that $struc(\tau', \Gamma'_l)$ is a supertree of $struc(\sigma', \Gamma'_r)$ and $struc(\tau', \Gamma'_l)$ contains no type variables.
proof. The lemma follows from Definition 29 and Lemma 8. □

Using Lemma 9, we can finally define structural subtyping trees.

Definition 30. Let τ_1 and τ_2 be closed μ-complete types, such that $struc(\tau_1, \emptyset) \sqsupseteq struc(\tau_2, \emptyset)$. The structural subtyping tree for τ_1 and τ_2 is defined as $sttree((\tau_1, \tau_2, \emptyset))$, where $sttree((\tau, \sigma, \Gamma))$ is defined as follows:

if $x = (\tau, \sigma, \Gamma)$ and $stchildren(x) = \emptyset$,
  then $sttree(x)$ has only one node, labeled (τ, σ, Γ)
if $x = (\tau, \sigma, \Gamma)$ and $stchildren(x) \neq \emptyset$,
  then $sttree(x)$ consists of a root, labeled (τ, σ, Γ), and, for every $y \in stchildren(x)$, a subtree $sttree(y)$ and an arrow from the root to subtree $sttree(y)$.

□

Lemma 10. The structural subtyping tree for τ_1 and τ_2 is finite.
Proof. The proof is the same as the proof of Lemma 5. □

Finally, in the same way as Claim 2 was proven, we can prove the following claim:

for every node labeled (τ, σ, Γ) in $sttree(\tau_1, \tau_2, \emptyset)$: $\Gamma \vdash_{DS} \tau \preceq \sigma$.

From the definition of *sttree* and the claim it follows that derivable subtyping is complete w.r.t. structural subtyping:

$$struc(\tau_1) \sqsupseteq struc(\tau_2) \;\Rightarrow\; \emptyset \vdash_{DS} \tau_1 \preceq \tau_2.$$

4.4 Equivalence of structural and extensional subtyping

In this subsection, we prove Theorem 6. More precisely, for closed μ-complete types τ_1 and τ_2, we prove:

$$\tau_1 \preceq_{struc} \tau_2 \;\Leftrightarrow\; \tau_1 \preceq_{ext} \tau_2.$$

Some of the work has already been done in the proof of Theorem 2. It suffices to prove Claim 7:

$$struc(\tau_1) \sqsupseteq struc(\tau_2) \;\Leftrightarrow\; struc\_ext(\tau_1) \sqsupseteq struc\_ext(\tau_2).$$

Using Claim 7 and Claim 4, we can conclude that structural and extensional subtyping are logically equivalent:

$$\begin{array}{l} struc(\tau_1) \sqsupseteq struc(\tau_2) \;\Leftrightarrow \\ \quad struc\_ext(\tau_1) \sqsupseteq struc\_ext(\tau_2) \;\Leftrightarrow \\ \quad \{struc(e) \mid e \in ext(\tau_1)\} \sqsupseteq \{struc(e) \mid e \in ext(\tau_2)\} \;\Leftrightarrow \\ \quad ext(\tau_1) \preceq ext(\tau_2). \end{array}$$

4.4.1 Proof of Claim 7

We prove the following claim:

$$struc(\tau_1) \sqsupseteq struc(\tau_2) \;\Leftrightarrow\; struc\_ext(\tau_1) \sqsupseteq struc\_ext(\tau_2).$$

The proof is similar to the proof of Claim 3. First, we define the projection of a term on (the tree of) a type. The projection is exactly the same as the projection in the proof of Claim 3, except for records.

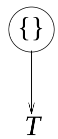
Definition 31. Let τ_1 and τ_2 be types, such that $struc(\tau_1) \sqsupseteq struc(\tau_2)$. Furthermore, let g be an injective function from $bvars(\tau_1) \times TypeVar$ to $Var - fvars(\tau_1)$ and e be a term, such that $\forall e' \in subterms(e)[inst(estruc(e'), estruc(\tau_1))]$. The projection of e on $estruc(\tau_2)$ is defined as $proj(e, estruc(\tau_2), \emptyset)$, where $proj(e', U, V)$ is defined as follows:

$$\begin{array}{l}
proj(x, node(t), V) = x \;\text{ if } x \in Var \\
proj(b, node(B), V) = b \;\text{ if } b \in Cons \\
proj(\emptyset, tree(\{T\}), V) = \emptyset \\
proj(\{e_1, \cdots, e_n\}, tree(\{T\}), V) = \{proj(e_1, T, V), \cdots, proj(e_n, T, V)\} \\
proj(x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = g(x,t) \\
\quad \text{if } g(x,t) \in V \\
proj(x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = proj(\mu x.e_x, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) \\
\quad \text{if } g(x,t) \notin V \text{ and } \eta_V(x) = \mu x.e_x \\
proj(\mu x. < l_1 = e_1, \cdots, l_{n+m} = e_{n+m} >, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = g(x,t) \\
\quad \text{if } g(x,t) \in V \\
proj(\mu x. < l_1 = e_1, \cdots, l_{n+m} = e_{n+m} >, tree(t. < l_1 : T_1, \cdots, l_n : T_n >), V) = \\
\quad \mu\, g(x,t). < l_1 : proj(e_1, T_1, V \cup \{g(x,t), \mu x. < l_1 = e_1, \cdots, l_{n+m} = e_{n+m} >\}), \cdots, \\
\qquad\qquad\quad l_n : proj(e_n, T_n, V \cup \{g(x,t), \mu x. < l_1 = e_1, \cdots, l_{n+m} = e_{n+m} >\}) > \\
\quad \text{if } g(x,t) \notin V,
\end{array}$$

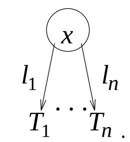
where

$node(l) =$

$tree(\{T\}) =$



$tree(x. < l_1 : T_1, \cdots, l_n : T_n >) =$



□

Proof of $\Rightarrow$. Suppose $struc(\tau_1) \sqsupseteq struc(\tau_2)$. Using Lemma 6, we can deduce that there is an injective tree morphism from $estruc(\tau_2)$ to $estruc(\tau_1)$, such that for every node or arrow q in $estruc(\tau_2)$ the following holds:

1. if $label(q) = t$ for some $t \in TypeVar$ and q is a leaf,
   then $label(\varphi(q)) = t$,
2. if $label(q) = t$ for some $t \in TypeVar$ and q is not a leaf,
   then $label(\varphi(q)) = s$ for some $s \in TypeVar$,
3. otherwise, $label(\varphi(q)) = label(q)$.

Now, let $struc(e)$ be an element of $struc\_ext(\tau_1)$ and e' be an element of $subterms(e)$. Furthermore, let $P(e')$ be $proj(e', estruc(\tau_2), \emptyset)$. Then $inst(estruc(e'), estruc(\tau_1))$, because e' is an element of $subterms(e)$ and $struc(e)$ is an element of $struc\_ext(\tau_1)$. From the definition of *proj* it follows that there is an injective tree morphism from $estruc(P(e'))$ to $estruc(\tau_2)$, such that for every node or arrow q in $estruc(P(e'))$ the following holds:

1. if $label(q) \in Cons_B$ for some $B \in BTypes$, then $label(\varphi(q)) = B$
2. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is a leaf, then $label(\varphi(q)) = t$
3. if $label(q) \in Var_t$ for some $t \in TypeVar$ and q is not a leaf,
   then $label(\varphi(q)) = t$ and $\mid children(q) \mid = \mid children(\varphi(q)) \mid$
4. otherwise, $label(\varphi(q)) = label(q)$.

That is, $inst(estruc(P(e')), estruc(\tau_2))$. Hence, for every $e' \in subterms(e)$, we have:

$inst(estruc(P(e')), estruc(\tau_2))]$.

Let $P(e)$ be $proj(e, estruc(\tau_2), \emptyset)$. Using the definition of *proj*, we can conclude that $struc(e) \sqsupseteq struc(P(e))$ and:

$\forall e'' \in subterms(P(e))[inst(estruc(e''), estruc(\tau_2))]$,

because $e'' \in subterms(P(e)) \Leftrightarrow (e'' = P(e') \wedge e' \in subterms(e))$. Since $FV(e) = FV(P(e))$, we have $struc(e) \sqsupseteq struc(P(e)) \in struc\_ext(\tau_2)$.

Proof of $\Leftarrow$. Suppose $struc\_ext(\tau_1) \sqsupseteq struc\_ext(\tau_2)$. Then there are terms e_1 and e_2, such that $inst(estruc(e_1), estruc(\tau_1))$, $inst(estruc(e_2), estruc(\tau_2))$, and $struc(e_1) \sqsupseteq struc(e_2)$. From Definition 18 and Lemma 6, it follows that $struc(\tau_1) \sqsupseteq struc(\tau_2)$.

4.5 Decidability of structural and extensional subtyping

The proof of Theorem 7 is the same as the proof of Theorem 3. In fact, in the same way as for derivable equivalence, we can prove that there is a decision procedure for derivable subtyping. Since $\tau_1 \preceq_{struc} \tau_2 \Leftrightarrow \tau_1 \preceq_D \tau_2$ and $\tau_1 \preceq_{ext} \tau_2 \Leftrightarrow \tau_1 \preceq_D \tau_2$, it follows that there is a decision procedure for structural and extensional subtyping.

5 Type transformations

In this section, we introduce a set of type transformations and prove that this set is sound and complete w.r.t. data capacity.

The set of basic type transformations consists of renaming and aggregation operations (cf. [1]).

Definition 32. Renaming is defined as a function of type $\mathcal{L} \rightarrow \mathcal{L} \rightarrow \mathit{Types} \rightarrow \mathit{Types}$:

$$
\begin{array}{l}
rename(l_i)(l)(t) = t \;\text{ if } t \in \mathit{TypeVar} \\
rename(l_i)(l)(B) = B \;\text{ if } B \in \mathit{BTypes} \\
rename(l_i)(l)(\{\upsilon\}) = \{\upsilon\} \\
rename(l_i)(l)(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n > \\
\quad \text{if } l_i \notin \{l_1, \cdots, l_n\} \text{ or } l \in \{l_1, \cdots, l_n\}, \\
rename(l_i)(l)(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = < l_1 : \upsilon_1, \cdots, l : \upsilon_i, l_n : \upsilon_n > \\
\quad \text{if } l_i \in \{l_1, \cdots, l_n\} \text{ and } l \notin \{l_1, \cdots, l_n\}, \\
rename(l_i)(l)(\mu t.\alpha) = \mu t.(rename(l_i)(l)(\alpha)).
\end{array}
$$

The set of basic renaming operations is given by:

$$\mathcal{BT}_{ren} = \{rename(l)(l') \mid l \in \mathcal{L} \wedge l' \in \mathcal{L}\}.$$

We distinguish between two kinds of aggregation: tupling and aggregation within a record type. Tupling is defined as a function of type $\mathcal{L} \rightarrow \mathit{Types} \rightarrow \mathit{Types}$:

$$tuple(l)(\tau) = < l : \tau >.$$

The inverse of tupling is de-tupling, defined as a function of type $\mathit{Types} \rightarrow \mathit{Types}$:

$$de\_tuple(< l : \tau >) = \tau.$$

For all other cases, we have: $de\_tuple(\upsilon) = \upsilon$. Aggregation within a record type is defined as a function of type $\wp_{fin}(\mathcal{L}) \rightarrow \mathcal{L} \rightarrow \mathit{Types} \rightarrow \mathit{Types}$:

$$
\begin{array}{l}
aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = \\
\quad < l_1 : \upsilon_1, \cdots, l :< l_i : \upsilon_i, \cdots, l_j : \upsilon_j >, \cdots, l_n : \upsilon_n > \\
\quad \text{if } \{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\} \text{ and } l \notin (\{l_1, \cdots, l_n\} - \{l_i, l_{i+1}, \cdots, l_j\}) \\
aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t.\alpha) = \\
\quad \mu t.(aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\alpha)) \\
\quad \text{if id} \notin \{l_i, l_{i+1}, \cdots, l_j\}, \\
aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \\
\quad \mu s. < l_1 : \tau_1[t \setminus s], \cdots, l : \mu t. < l_i : \tau_i[t \setminus s], \cdots, l_j : \tau_j[t \setminus s] >, \cdots, l_n : \tau_n[t \setminus s] > \\
\quad \text{if id} \in \{l_i, l_{i+1}, \cdots, l_j\} \text{ and } \{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}.
\end{array}
$$

For all other cases, we have: $aggregate(L)(l)(\upsilon) = \upsilon$. The inverse of aggregation is de-aggregation, defined as a function of type $\mathcal{L} \rightarrow \mathit{Types} \rightarrow \mathit{Types}$:

$$
\begin{array}{l}
de\_aggregate(l_i)(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = \\
\quad < l_1 : \upsilon_1, \cdots, l_{i-1} : \upsilon_{i-1}, l'_1 : \sigma_1, \cdots, l'_m : \sigma_m, l_{i+1} : \upsilon_{i+1}, \cdots, l_n : \upsilon_n > \\
\quad \text{if } l_i \in \{l_1, \cdots, l_n\} \text{ and } \upsilon_i = < l'_1 : \sigma_1, \cdots, l'_m : \sigma_m > \\
\quad \text{and } \{l'_1, \cdots, l'_m\} \cap (\{l_1, \cdots, l_{i-1}, l_{i+1}, \cdots, l_n\} = \emptyset \\
de\_aggregate(l_i)(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) =
\end{array}
$$

$< l_1 : \upsilon_1, \cdots, l_{i-1} : \upsilon_{i-1}, l'_1 : \sigma_1, \cdots, l'_m : \sigma_m, l_{i+1} : \upsilon_{i+1}, \cdots, l_n : \upsilon_n >$
if $l_i \in \{l_1, \cdots, l_n\}$ and $\upsilon_i = \mu t. < l'_1 : \sigma_1, \cdots, l'_m : \sigma_m >$
and $\{l'_1, \cdots, l'_m\} \cap (\{l_1, \cdots, l_{i-1}, l_{i+1}, \cdots, l_n\} = \emptyset$ and $t \notin fvars(< l'_1 : \sigma_1, \cdots, l'_m : \sigma_m >)$
$de\_aggregate(l_i)(\mu t.\alpha) = \mu t.(de\_aggregate(l_i)(\alpha))$.

For all other cases, we have: $de\_aggregate(l)(\upsilon) = \upsilon$. The set of basic type transformations, denoted by $\mathcal{BT}$, is given by:

$$\mathcal{BT} = \mathcal{BT}_{ren} \cup \{tuple(l) \mid l \in \mathcal{L}\} \cup \{de\_tuple\} \cup \{aggregate(L)(l) \mid L \subseteq \mathcal{L} \wedge l \in \mathcal{L}\} \cup \{de\_aggregate(l) \mid l \in \mathcal{L}\}.$$

□

Complex type transformations are obtained by combining basic type transformations.

Definition 33. The set of complex type transformations, denoted by $\mathcal{CT}$, is inductively defined by:

1. if $F \in \mathcal{BT}$, then $F \in \mathcal{CT}$
2. if $F_1 \in \mathcal{CT}$ and $F_2 \in \mathcal{CT}$, then $F_1 \circ F_2 \in \mathcal{CT}$
3. if $F \in \mathcal{CT}$, then $\{F\} \in \mathcal{CT}$
4. if $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}$ is a set of n distinct labels and $\{F_1, \cdots, F_n\} \subseteq \mathcal{CT}$, then $< l_1 : F_1, \cdots, l_n : F_n > \in \mathcal{CT}$.

Complex type transformation $F_1 \circ F_2$ is the composition of F_1 and F_2:

$$F_1 \circ F_2(\upsilon) = F_1(F_2(\upsilon)).$$

Complex type transformation $\{F\}$ transforms set types and leaves other types unchanged:

$$\{F\}(\{\upsilon\}) = \{F(\upsilon)\}.$$

Complex type transformation $F = < l_1 : F_1, \cdots, l_n : F_n >$ transforms record types and leaves other types unchanged:

$$F(< l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = < l_1 : F_1(\upsilon_1), \cdots, l_n : F_n(\upsilon_n) >$$
$$F(\mu t. < l_1 : \upsilon_1, \cdots, l_n : \upsilon_n >) = \mu t. < l_1 : F_1(\upsilon_1), \cdots, l_n : F_n(\upsilon_n) >.$$

The set of complex renaming operations, denoted by $\mathcal{CT}_{ren}$, is obtained by replacing $\mathcal{BT}$ by $\mathcal{BT}_{ren}$ and $\mathcal{CT}$ by $\mathcal{CT}_{ren}$. □

Transformational type equality and type equivalence are defined in terms of type transformations.

Definition 34. Let τ_1 and τ_2 be closed types. Transformational equality of τ_1 and τ_2, denoted by $\tau_1 =_{trans} \tau_2$, is defined as follows:

$$\tau_1 =_{trans} \tau_2 \Leftrightarrow \exists F \in \mathcal{CT}\ [F(\tau_1) =_D \tau_2].$$

Transformational equivalence, denoted by $\cong_{trans}$, is defined by:

$$\tau_1 \cong_{trans} \tau_2 \Leftrightarrow \exists \upsilon_1 \in Types \exists \upsilon_2 \in Types[\tau_1 \cong_D \upsilon_1 \wedge \tau_2 \cong_D \upsilon_2 \wedge \upsilon_1 =_{trans} \upsilon_2].$$

□

Types can be normalised by applying de-tupling and de-aggregation operations.

Definition 35. Let τ be a type. The normal form of τ, denoted by $nf(\tau)$, is defined as follows:

$$
\begin{array}{ll}
nf(t) = t & \text{if } t \in \mathit{TypeVar} \\
nf(B) = B & \text{if } B \in \mathit{BTypes} \\
nf(\{v\}) = \{nf(v)\} & \\
nf(< l_1 : v_1, \cdots, l_n : v_n >) = & \\
\quad collapse(< l_1 : nf(v_1), \cdots, l_n : nf(v_n) >) & \\
nf(\mu t.\alpha) = \mu t.(nf(\alpha)) & \text{if } t \in fvars(\alpha) \\
nf(\mu t.\alpha) = nf(\alpha) & \text{if } t \notin fvars(\alpha),
\end{array}
$$

where $collapse(\tau')$ is obtained from τ' by applying the following rewrite rules until they cannot be applied any more:

1. $< l_1 : v_1, \cdots, l_{i-1} : v_{i-1}, l :< l_i : v_i, \cdots, l_j : v_j >, l_{j+1} : v_{j+1}, \cdots, l_n : v_n >$
 $\longrightarrow < l_1 : v_1, \cdots, l_{i-1} : v_{i-1}, l\_l_i : v_i, \cdots, l\_l_j : v_j, l_{j+1} : v_{j+1}, \cdots, l_n : v_n >$
2. $< l : v > \longrightarrow v.$

□

The following lemma gives the relation between transformational equality and normal forms of types.

Lemma 11. Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{trans} \tau_2 \;\Leftrightarrow\; nf(\tau_1) \approx nf(\tau_2),$$

where $\approx$ is defined as follows: $v_1 \approx v_2$ if and only if there is a renaming operation $F \in \mathcal{CT}_{ren}$, such that $F(v_1) =_D v_2$.
Proof of $\Rightarrow$. Let τ be a type and F be a basic type transformation. By a simple case distinction w.r.t. F, it follows that:

$$nf(\tau) \approx nf(F(\tau)).$$

Let τ be a type and F be a complex type transformation. By a simple induction on the structure of F, it follows that:

$$nf(\tau) \approx nf(F(\tau)).$$

Now, suppose $\tau_1 =_{trans} \tau_2$, i.e., there is a type transformation F, such that $F(\tau_1) =_D \tau_2$. Then:

$$nf(\tau_1) \approx nf(F(\tau_1)) =_D nf(\tau_2).$$

From the definition of $\approx$, it follows that: $nf(\tau_1) \approx nf(\tau_2)$.
Proof of $\Leftarrow$. Let τ be a type and G be a basic de-aggregation operation. Then there is a basic aggregation operation G', such that $G'(G(\tau)) = \tau$.

Now, suppose $nf(\tau_1) \approx nf(\tau_2)$. Then there is a renaming operation $F \in \mathcal{CT}_{ren}$, such that $F(nf(\tau_1)) =_D nf(\tau_2)$. Note that every rewrite step can be obtained by a combination of a renaming operation and a de-aggregation operation. Hence, there are type transformations F_1 and F_2, such that:

$$F(F_1(\tau_1)) =_D F_2(\tau_2).$$

Furthermore, there is a complex aggregation operation F_2', such that $F_2'(F(F_1(\tau_1))) =_D F_2'(F_2(\tau_2)) = \tau_2$. Hence, $\tau_1 =_{trans} \tau_2$. □

5.1 Soundness

In this subsection, we introduce semantic type equality based on data capacity and prove that transformational type equality is sound w.r.t. semantic type equality.
First, we need a number of preliminary definitions.

Definition 36. Let τ be a type, such that, for every type variable t, if both $\mu t.\alpha$ and $\mu t.\beta$ occur in τ, then $\alpha = \beta$. The set of applied type variables in τ, denoted by $avars(\tau)$, is defined as:

$$\begin{array}{l} avars(t) = \{t\} \ \text{ if } t \in TypeVar \\ avars(B) = \emptyset \ \text{ if } B \in BTypes \\ avars(\{v\}) = avars(v) \\ avars(< l_1 : v_1, \cdots, l_m : v_m >) = avars(v_1) \cup \cdots \cup avars(v_m) \\ avars(\mu t.\alpha) = avars(\alpha). \end{array}$$

The head of τ, denoted by $hd(\tau)$, is defined as:

$$\begin{array}{l} hd(t) = t \ \text{ if } t \in TypeVar \\ hd(B) = B \ \text{ if } B \in BTypes \\ hd(\{v\}) = \{hd(v)\} \\ hd(< l_1 : v_1, \cdots, l_m : v_m >) = < l_1 : hd(v_1), \cdots, l_m : hd(v_m) > \\ hd(\mu t.\alpha) = t. \end{array}$$

Let t be a bound type variable in τ. The tail of τ w.r.t. t, denoted by $tl(\tau, t)$, is defined as:

$$\begin{array}{l} tl(\{v\}, t) = tl(v, t) \\ tl(< \cdots, l : v, \cdots >, t) = tl(v, t) \\ tl(\mu t.\alpha, t) = \mu t.\alpha \\ tl(\mu t'.\alpha', t) = tl(\alpha', t) \ \text{ if } t' \neq t \end{array}$$

where $t \in bvars(v)$ and $t \in bvars(\alpha')$. Finally, the unfolded counterpart of τ w.r.t. t, denoted by $unfold(\tau, t)$, is defined as:

$$\begin{array}{l} unfold(\{v\}, t) = \{unfold(v, t)\} \\ unfold(< \cdots, l : v, \cdots >, t) = < \cdots, unfold(v, t), \cdots > \\ unfold(\mu t.\alpha, t) = eliminate(\alpha[t \setminus \mu t.\alpha], \emptyset) \\ unfold(\mu t'.\alpha', t) = \mu t'.(unfold(\alpha', t)) \ \text{ if } t' \neq t, \end{array}$$

where $t \in bvars(v)$ and $t \in bvars(\alpha')$, and *eliminate* eliminates the second occurrence of $\mu t'$ whenever one $\mu t'$ occurs in the range of another $\mu t'$:

$$\begin{array}{l} eliminate(t') = t' \ \text{ if } B \in TypeVar \\ eliminate(B, V) = B \ \text{ if } B \in BTypes \\ eliminate(\{v\}, V) = \{eliminate(v, V)\} \\ eliminate(< l_1 : v_1, \cdots, l_m : v_m >, V) = < l_1 : eliminate(v_1, V), \cdots, l_m : eliminate(v_m, V) > \\ eliminate(\mu t'.\alpha', V) = \mu t'.(eliminate(\alpha, V \cup \{t'\})) \ \text{ if } t' \notin V \\ eliminate(\mu t'.\alpha', V) = eliminate(\alpha', V) \ \text{ if } t' \in V. \end{array}$$

□

The set of preterms of a type is the set of terms of depth 1.

Definition 37. First, for every basic type B and every natural number n, we choose $Cons(B, n)$ to be a subset of $Cons_B$ consisting of n elements, and, for every type variable t and every natural number n, we choose $Var(t, n)$ to be a subset of Var_t consisting of n elements.

Let τ be a type, such that $avars(\tau) = \{t_{i_1}, \cdots, t_{i_n}\}$, where $i_1 < \cdots < i_n$. Furthermore, let $\vec{p} = (p_1, p_2, p_3, p_4)$ and $\vec{q} = (q_1, \cdots, q_n)$ be natural number vectors. The set of preterms of τ w.r.t. $\vec{p}$ and $\vec{q}$, denoted by $preterms(\tau, \vec{p}, \vec{q})$, is defined as follows:

$$\begin{array}{l} preterms(t_{i_j}, \vec{p}, \vec{q}) = Var(t_{i_j}, q_j) \ \text{ if } t_{i_j} \in TypeVar, \\ preterms(B_i, \vec{p}, \vec{q}) = Cons(B_i, p_i) \ \text{ if } B_i \in BTypes, \\ preterms(\{v\}, \vec{p}, \vec{q}) = \wp_{fin}(preterms(v, \vec{p}, \vec{q})), \\ preterms(< l_1 : v_1, \cdots, l_n : v_n >, \vec{p}, \vec{q}) = \\ \quad \{< l_1 = e_1, \cdots, l_n = e_n > \mid e_1 \in preterms(v_1, \vec{p}, \vec{q}) \wedge \cdots \wedge e_n \in preterms(v_n, \vec{p}, \vec{q})\}, \\ preterms(\mu t.\alpha, \vec{p}, \vec{q}) = \{\mu x.e \mid x \in Var(t, 1) \wedge e \in preterms(\alpha, \vec{p}, \vec{q})\}, \end{array}$$

where B_1 denotes type oid, B_2 denotes type integer, B_3 denotes type rational, and B_4 denotes type string. $\Box$

The data capacity function of a type is defined as follows.

Definition 38. Let τ be a type, such that $avars(\tau) = \{t_{i_1}, \cdots, t_{i_n}\}$, where $i_1 < \cdots < i_n$. The data capacity function of type τ, denoted by χ_τ, is defined as:

$$\lambda\vec{p}\,\lambda\vec{q}.\ \chi(\tau, \vec{p}, \vec{q}),$$

where $\vec{p} = (p_1, p_2, p_3, p_4)$ and $\vec{q} = (q_1, \cdots, q_n)$ are natural number vectors and:

$$\begin{array}{l}
\chi(t_{i_j}, \vec{p}, \vec{q}) = q_j \ \text{ if } t_{i_j} \in VarType, \\
\chi(\mathrm{B}_i, \vec{p}, \vec{q}) = p_i \ \text{ if } B_i \in BTypes, \\
\chi(\{\upsilon\}, \vec{p}, \vec{q}) = 2^{\chi(\upsilon, \vec{p}, \vec{q})}, \\
\chi(< l_1 : \upsilon_1, \cdots, l_m : \upsilon_m >, \vec{p}, \vec{q}) = \chi(\upsilon_1, \vec{p}, \vec{q}) \times \cdots \times \chi(\upsilon_m, \vec{p}, \vec{q}), \\
\chi(\mu t.\alpha, \vec{p}, \vec{q}) = \chi(\alpha, \vec{p}, \vec{q}),
\end{array}$$

where B_1 denotes type oid, B_2 denotes type integer, B_3 denotes type rational, and B_4 denotes type string. $\Box$

The data capacity function of a type gives the number of preterms of the type.

Lemma 12. Let τ be a type, such that $avars(\tau) = \{t_{i_1}, \cdots, t_{i_n}\}$, where $i_1 < \cdots < i_n$. For every natural number vector $\vec{p} = (p_1, p_2, p_3, p_4)$ and every natural number vector $\vec{q} = (q_1, \cdots, q_n)$:

$$\chi_\tau(\vec{p}, \vec{q}) = |\ preterms(\tau, \vec{p}, \vec{q})\ |.$$

Proof. The lemma follows from an induction argument on the structure of τ. $\Box$

Semantic type equality and type equivalence are defined in terms of data capacity functions.

Definition 39. Let τ_1 and τ_2 be types. Semantic equality of τ_1 and τ_2, denoted by $\tau_1 =_{sem} \tau_2$, is defined as follows:

$$\tau_1 =_{sem} \tau_2 \Leftrightarrow \exists\upsilon_1[\tau_1 =_D \upsilon_1 \wedge \chi_{\upsilon_1} = \chi_{\tau_2} \wedge \forall t \in avars(\upsilon_1)[\chi_{unfold(\upsilon_1,t)} = \chi_{unfold(\tau_2,t)}]].$$

Semantic equivalence, denoted by $\cong_{sem}$, is defined by:

$$\tau_1 \cong_{sem} \tau_2 \ \Leftrightarrow\ \exists\upsilon_1 \in Types\exists\upsilon_2 \in Types[\tau_1 \cong_D \upsilon_1 \wedge \tau_2 \cong_D \upsilon_2 \wedge \upsilon_1 =_{sem} \upsilon_2].$$

$\Box$

Finally, we can prove that transformational equality is sound w.r.t. semantic equality.

Theorem 8. Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{trans} \tau_2 \Rightarrow \tau_1 =_{sem} \tau_2.$$

Proof. Let τ be a type and F be a basic type transformation. By a case distinction w.r.t. F, it follows that:

$$F(\tau) =_{sem} \tau.$$

Let τ be a type and F be a complex type transformation. By an induction on the structure of F, it follows that:

$$F(\tau) =_{sem} \tau.$$

Now, suppose $\tau_1 =_{trans} \tau_2$, i.e., there is a type transformation F, such that $F(\tau_1) =_D \tau_2$. Then:

$$\tau_1 =_{sem} F(\tau_1) =_D \tau_2.$$

From the definition of $=_{sem}$, it follows that: $\tau_1 =_{sem} \tau_2$. $\Box$

5.2 Completeness

In this subsection, we prove that transformational type equality is complete w.r.t. semantic type equality.
First, we prove a number of lemmas. The following lemma gives the relation between the head and the tails of a type.

Lemma 13. Let σ be a type and V be $avars(hd(\sigma)) \cap bvars(\sigma)$. Then:

$$hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \sigma.$$

Proof.
The lemma follows from an induction argument on the structure of type σ. If σ is a basic type or a type variable, then $V = \emptyset$ and, hence, $hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \sigma$. If $\sigma = \{\sigma'\}$, then:

$$\begin{array}{l} hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \\ \{hd(\sigma)\}[t \setminus tl(\sigma', t) \mid t \in V] = \\ \{hd(\sigma)[t \setminus tl(\sigma', t) \mid t \in V]\} = \\ \{\sigma'\}. \end{array}$$

If $\sigma = < l_1 : \sigma_1, \cdots, l_n : \sigma_n >$, then:

$$\begin{array}{l} hd(\sigma)[t \setminus tl(\sigma, t) \mid t \in V] = \\ < l_1 : hd(\sigma_1), \cdots, l_n : hd(\sigma_n) > [t \setminus tl(\sigma, t) \mid t \in V] = \\ < l_1 : hd(\sigma_1)[t \setminus tl(\sigma_1, t) \mid t \in V_1], \cdots, l_n : hd(\sigma_n)[t \setminus tl(\sigma_n, t) \mid t \in V_n] = \\ < l_1 : \sigma_1, \cdots, l_n : \sigma_n >. \end{array}$$

If $\sigma = \mu t'.\alpha$, then $V = \{t'\}$ and $hd(\sigma)[t' \setminus tl(\sigma, t')] = t'[t' \setminus \sigma] = \sigma$. □

The following two lemmas give the relation between the data capacity functions of a type and its unfolded counterpart.

Lemma 14a. Let σ be a type, such that $t \in bvars(\sigma)$, where $t = t_j$ for some j. Then:

$$\lambda\vec{p}\lambda\vec{q}.\chi_{unfold(\sigma,t)}(\vec{p}, \vec{q}) = \lambda\vec{p}\lambda\vec{q}.\chi_\sigma(\vec{p}, \vec{q}[q_j \setminus \chi_{tl(\sigma,t)}(\vec{p}, \vec{y})]),$$

where $\vec{y}$ contains the type variables that occur in $tl(\sigma, t)$.
Proof.
The lemma follows from an induction argument on the structure of type σ. We give a proof for the non-trivial case: $\sigma = \mu t.\alpha$. Let there be n occurrences of t in α. Let α' be α, with every occurrence of t replaced by a unique member of $\{t_{i_1}, \cdots, t_{i_n}\}$, which is a set of new type variables. Then:

$$\begin{array}{l} \chi_{unfold(\sigma,t)} = \chi_{eliminate(\alpha[t \setminus \sigma])} = \\ \qquad \chi_{eliminate(\alpha[t \setminus \sigma])} = \\ \qquad \chi_{\alpha'[t_{i_1} \setminus eliminate(\sigma, V_1), \cdots, t_{i_n} \setminus eliminate(\sigma, V_n)]} = \\ \qquad \lambda\vec{p}\lambda\vec{q}.(\chi_{\alpha'}(\vec{p}, \vec{y})[q_{i_1} \setminus \chi_\sigma(\vec{p}, \vec{q}), \cdots, q_{i_n} \setminus \chi_\sigma(\vec{p}, \vec{q})]) = \\ \qquad \lambda\vec{p}\lambda\vec{q}.(\chi_\alpha(\vec{p}, \vec{q})[q_j \setminus \chi_\sigma(\vec{p}, \vec{q})]) = \\ \qquad \lambda\vec{p}\lambda\vec{q}.(\chi_\sigma(\vec{p}, \vec{q})[q_j \setminus \chi_\sigma(\vec{p}, \vec{q})]), \end{array}$$

where the V_i's are the sets induced by applying *eliminate*, and $\vec{y}$ contains the type variables that occur in α'. □

Lemma 14b. Let σ and σ' be types, such that $t \in bvars(\sigma) \cap bvars(\sigma')$. Then:

$$(\chi_\sigma = \chi_{\sigma'} \wedge \chi_{unfold(\sigma,t)} = \chi_{unfold(\sigma',t)}) \Rightarrow \chi_{tl(\sigma,t)} = \chi_{tl(\sigma',t)}.$$

Proof.
Suppose $\chi_\sigma = \chi_{\sigma'}$ and $\chi_{unfold(\sigma,t)} = \chi_{unfold(\sigma',t)}$. Furthermore, suppose $\chi_{tl(\sigma,t)} \neq \chi_{tl(\sigma',t)}$. Then:

$$
\begin{aligned}
\lambda\vec{p}\lambda\vec{q}.\chi_{unfold(\sigma,t)}(\vec{p},\vec{q}) = {} \\
& \lambda\vec{p}\lambda\vec{q}.\chi_{\sigma}(\vec{p},\vec{q}[q_j \setminus \chi_{tl(\sigma,t)}(\vec{p},\vec{y})]) \neq \\
& \lambda\vec{p}\lambda\vec{q}.\chi_{\sigma}(\vec{p},\vec{q}[q_j \setminus \chi_{tl(\sigma',t)}(\vec{p},\vec{y})]) = \\
& \lambda\vec{p}\lambda\vec{q}.\chi_{\sigma'}(\vec{p},\vec{q}[q_j \setminus \chi_{tl(\sigma',t)}(\vec{p},\vec{y})]) = \lambda\vec{p}\lambda\vec{q}.\chi_{unfold(\sigma',t)}(\vec{p},\vec{q}),
\end{aligned}
$$

where the first step follows from Lemma 14a, the second from the fact that χ_{σ} is injective in q_j and the fact that $\chi_{tl(\sigma,t)} \neq \chi_{tl(\sigma',t)}$, the third from the fact that $\chi_{\sigma} = \chi_{\sigma'}$, and the fourth from Lemma 14a. Contradiction. Hence, $\chi_{tl(\sigma,t)} = \chi_{tl(\sigma',t)}$. □

The following three lemmas give the relation between the data capacity functions of a number of types on one hand and the data capacity function of the combined type obtained by substitution on the other hand.

Lemma 15a. Let σ be a type, such that $t \in \mathit{fvars}(\sigma)$, where $t = t_j$ for some j. Furthermore, let τ be a type. Then:

$$\chi_{\sigma[t\setminus\tau]} = \lambda\vec{p}\lambda\vec{z}.(\chi_{\sigma}(\vec{p},\vec{q})[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})]),$$

where $\vec{y}$ contains the type variables that occur in τ and $\vec{z}$ contains the type variables that occur in $\sigma[t \setminus \tau]$.
Proof.
The lemma follows from an induction argument on the structure of type σ. We give a proof for the non-trivial case: $\sigma = t$. Then $\chi_{\sigma[t\setminus\tau]} = \chi_{\tau}$ and:

$$\chi_{\sigma}(\vec{p},\vec{q})[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})]) = q_j[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})]) = \chi_{\tau}(\vec{p},\vec{y}).$$

Since $\vec{z} = \vec{y}$ in this case, it follows that:

$$\lambda\vec{p}\lambda\vec{z}.(\chi_{\sigma}(\vec{p},\vec{q})[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})]) = \lambda\vec{p}\lambda\vec{z}.\chi_{\tau}(\vec{p},\vec{z}) = \chi_{\tau}.$$

□

Lemma 15b. Let σ and σ' be types, such that $t \in \mathit{fvars}(\sigma) \cap \mathit{fvars}(\sigma')$, where $t = t_j$ for some j. Furthermore, let τ, and τ' be types, such that $t \in \mathit{avars}(\tau) \cap \mathit{avars}(\tau')$. Then:

$$(\chi_{\sigma[t\setminus\tau]} = \chi_{\sigma'[t\setminus\tau']} \wedge \chi_{\tau} = \chi_{\tau'}) \;\Rightarrow\; \chi_{\sigma} = \chi_{\sigma'}.$$

Proof.
Suppose $\chi_{\sigma[t\setminus\tau]} = \chi_{\sigma'[t\setminus\tau']}$ and $\chi_{\tau} = \chi_{\tau'}$. Furthermore, suppose $\chi_{\sigma} \neq \chi_{\sigma'}$. Then $\lambda q_j.\chi_{\sigma}(\vec{p},\vec{q}) \neq \lambda q_j.\chi_{\sigma'}(\vec{p},\vec{q})$. Without loss of generality, let Q be a natural number, such that:

$$\forall q_j \geq Q[\chi_{\sigma}(\vec{p},\vec{q}) > \lambda\chi_{\sigma'}(\vec{p},\vec{q})].$$

Since $\forall q_j[\chi_{\tau}(\vec{p},\vec{q}) > q_j]$, it follows that:

a) $\forall q_j \geq Q[\chi_{\sigma}(\vec{p},\vec{q}[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})]) > \chi_{\sigma'}(\vec{p},\vec{q}[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})])]$.

Then:

$$
\begin{aligned}
\chi_{\sigma[t\setminus\tau]} &= \lambda\vec{p}\lambda\vec{z}.(\chi_{\sigma}(\vec{p},\vec{q}[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})])) \neq \\
& \lambda\vec{p}\lambda\vec{z}.(\chi_{\sigma'}(\vec{p},\vec{q}[q_j \setminus \chi_{\tau}(\vec{p},\vec{y})])) = \\
& \lambda\vec{p}\lambda\vec{z}.(\chi_{\sigma'}(\vec{p},\vec{q}[q_j \setminus \chi_{\tau'}(\vec{p},\vec{y})])) = \chi_{\sigma'[t\setminus\tau']},
\end{aligned}
$$

where the first step follows from Lemma 15a, the second from a), the third from the fact that $\chi_{\tau} = \chi_{\tau'}$, and the fourth from Lemma 15a. Contradiction. Hence, $\chi_{\sigma} = \chi_{\sigma'}$. □

Lemma 15c. Let σ and σ' be types, such that $\mathit{fvars}(\sigma) = \mathit{fvars}(\sigma') = \{t_i \mid i \in I\}$, where I is a subset of the natural numbers. Furthermore, let $\{\sigma_i \mid i \in I\}$ and $\{\sigma'_i \mid i \in I\}$ be sets of types, such that $t_i \in \mathit{avars}(\sigma_i) \cap \mathit{avars}(\sigma'_i)$. Define:

$\tau = \sigma[i \setminus \sigma_i \mid i \in I]$
$\tau' = \sigma'[i \setminus \sigma_i' \mid i \in I]$.

Then:

$$(\chi_\tau = \chi_{\tau'} \wedge \forall i \in I[\chi_{\sigma_i} = \chi_{\sigma_i'}]) \;\Rightarrow\; \chi_\sigma = \chi_{\sigma'}.$$

Proof.
The lemma follows from $\mid I \mid$ applications of Lemma 15b. $\square$

The following lemma states that the data capacity functions corresponding to the different type variables in a type uniquely determine the levels at which the type variables are bound.

Lemma 16. Let σ and σ' be types, such that $bvars(\sigma) = bvars(\sigma') = \{t_i \mid i \in I\}$, where I is a subset of the natural numbers. Furthermore, let V be $avars(hd(\sigma)) \cap bvars(\sigma)$ and V' be $avars(hd(\sigma')) \cap bvars(\sigma')$. For every $i \in I$, define:

$\mu\, t_i.\ \alpha_i = tl(\sigma, t_i)$
$\mu\, t_i.\ \alpha_i' = tl(\sigma', t_i)$.

Then:

$$\forall i \in I[\chi_{\alpha_i} = \chi_{\alpha_i'}] \;\Rightarrow\; V = V'.$$

Proof.
Suppose $\forall i \in I[\chi_{\alpha_i} = \chi_{\alpha_i'}]$. In order to prove the lemma, it is sufficient to prove that α_i contains $\mu\, t_j.\ \alpha_j$ if and only if α_i' contains $\mu\, t_j.\ \alpha_j'$.

Suppose α_i contains $\mu\, t_j.\ \alpha_j$. It follows that χ_{α_i} contains χ_{α_j}. Hence, $\chi_{\alpha_i'} = \chi_{\alpha_i}$ has at least one occurrence of x_j. Suppose α_i' does not contain $\mu\, t_j.\ \alpha_j'$ Since χ_{α_i}' has at least one occurrence of x_j, α_j' must contain $\mu\ t_i.\ \alpha_i$. Then $\chi_{\alpha_j} = \chi_{\alpha_j'}$ contains $\chi_{\alpha_i'}$. It follows that $\chi_{\alpha_i} \neq \chi_{\alpha_i'}$. Contradiction. Hence, α_i' contains $\mu\, t_j.\ \alpha_j'$. $\square$

Finally, we can prove that transformational equality is complete w.r.t. semantic equality.

Theorem 9. Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 =_{sem} \tau_2 \Rightarrow \tau_1 =_{trans} \tau_2.$$

Proof.
For non-recursive types, the theorem follows from Theorem 5.4 in [1]. For recursive types, the proof is more involved.

Suppose $\tau_1 =_{sem} \tau_2$. Let v_1 be the type, such that $v_1 =_D \tau_1$ and $\chi_{v_1} = \chi_{\tau_2}$. For every i in $I = \{j \mid t_j \in avars(v_1)\}$, define:

$\mu\, t_i.\ \alpha_i = tl(v_1, t_i)$
$\mu\, t_i.\ \alpha_i' = tl(\tau_2, t_i)$
$V_i = avars(hd(\alpha_i)) \cap bvars(\alpha_i)$.

From $\chi_{v_1} = \chi_{\tau_2}$, $\forall i \in I[\chi_{unfold(v_1,t_i)} = \chi_{unfold(\tau_2,t_i)}]$, Lemma 14b and 16, it follows that, for every i in I:

b) $\chi_{\alpha_i} = \chi_{\alpha_i'}$
c) $V_i = avars(hd(\alpha_i')) \cap bvars(\alpha_i')$.

For every i in I, define $\sigma_i = \mu\, t_i.\ \alpha_i$ and $\sigma_i' = \mu\, t_i.\ \alpha_i'$. Using c) and Lemma 13, we obtain:

$\alpha_i = hd(\alpha_i)[t_j \setminus tl(\alpha_i, t_j) \mid t_j \in V_i]$
$\alpha_i' = hd(\alpha_i')[t_j \setminus tl(\alpha_i', t_j) \mid t_j \in V_i]$.

From this, b), and Lemma 15c, it follows that:

$\chi_{hd(\alpha_i)} = \chi_{hd(\alpha'_i)}$.

Using the fact that $hd(\alpha_i)$ and $hd(\alpha'_i)$ are non-recursive types, we can conclude that:

d) $nf(hd(\alpha_i)) \approx nf(hd(\alpha'_i))$.

By an induction argument on the number of nestings of μ-operators, we can prove that, for every $i \in I$, $nf(\sigma_i) \approx nf(\sigma'_i)$. The induction step of the induction argument is:

$$\begin{aligned} nf(\alpha_i) = {} & nf(hd(\alpha_i)[t_j \setminus tl(\alpha_i, t_j) \mid t_j \in V_i]) = \\ & nf(hd(\alpha_i)[t_j \setminus \sigma_j \mid t_j \in V_i]) = \\ & (nf(hd(\alpha_i)))[t_j \setminus nf(\sigma_j) \mid t_j \in V_i] \approx \\ & (nf(hd(\alpha'_i)))[t_j \setminus nf(\sigma'_j) \mid t_j \in V_i] = \\ & nf(hd(\alpha'_i)[t_j \setminus \sigma'_j \mid t_j \in V_i]) = \\ & nf(hd(\alpha'_i)[t_j \setminus tl(\alpha'_i, t_j) \mid t_j \in V_i]) = \\ & nf(\alpha'_i), \end{aligned}$$

where the first step follows from Lemma 13; the second from the definition of tl and the fact that υ_1 contains α_i; the third from the definition of nf and the fact that every t_j occurs free in α_j; the fourth from the induction hypothesis and d); the fifth from the definition of nf and the fact that every t_j occurs free in α_j; the sixth from the definition of tl and the fact that τ_2 contains α'_i; and the final from Lemma 13 and c).

Case 1: $\upsilon_1 = \mu\ t_i.\ \alpha_i$ for some i. Suppose $\tau_2 \neq \mu\ t_i.\ \alpha'_i$. Since τ_2 must contain $\mu\ t_i.\ \alpha'_i$, it follows that $\chi_{\tau_2} \neq \chi_{\alpha'_i} = \chi_{\alpha_i} = \chi_{\upsilon_1}$. Contradiction. Hence, $\tau_2 = \mu\ t_i.\ \alpha'_i$ and $nf(\upsilon_1) \approx nf(\tau_2)$. That is, $\tau_1 =_{trans} \tau_2$.

Case 2: $\upsilon_1 \neq \mu\ \sigma_j$ for any i. From $\upsilon_1 = hd(\upsilon_1)[t_j \setminus \mu\ t_i.\ \alpha'_i \mid t_j \in V]$, where $V = avars(hd(\upsilon_1)) \cap bvars(\upsilon_1)$, Lemma 13 and 16, it follows that:

$$\tau_2 = hd(\tau_2)[t_j \setminus \sigma'_j \mid t_j \in V].$$

Using $\chi_{\upsilon_1} = \chi_{\tau_2}$, b), and Lemma 15c, we can conclude that $\chi_{hd}(\upsilon_1) = \chi_{hd}(\tau_2)$. Hence, $nf(hd(\upsilon_1)) \approx nf(hd(\tau_2))$ and, because of the fact that $nf(\sigma_j) \approx nf(\sigma'_i)$, $nf(\upsilon_1) \approx nf(\tau_2)$. That is, $\tau_1 =_{trans} \tau_2$. □

Combining Theorem 8 and 9, we can deduce that transformational equivalence is sound and complete w.r.t. semantic equivalence.

Theorem 10. Let τ_1 and τ_2 be closed types. Then:

$$\tau_1 \cong_{trans} \tau_2 \Leftrightarrow \tau_1 \cong_{sem} \tau_2.$$

Proof. The theorem follows from Definition 34, Definition 39, Theorem 8, and Theorem 9. □

6 Conclusion

In this report, a number of completeness results are given that are useful for database integration. Derivable type equivalence is proven sound and complete w.r.t. extensional type equivalence. This means that if a type is equivalent to another type, any instance of the first type is equivalent to some instance of the second type. Furthermore, derivable subtyping is proven sound and complete w.r.t. extensional subtyping. This means that if a type is a subtype of another type, any instance of the first type can be transformed in a canonical way into an instance of the second type. Finally, the set of chosen type transformations is proven sound and complete w.r.t. data capacity. As a consequence, type instances can be transformed uniquely.

These results have a number of implications. First, our formalisation of class hierarchies respects the subclass relation in the following way. Let C_1 be defined as a subclass of C_2. If o is an instance of C_1, then the projection of o onto C_2 is well-defined and is an instance of C_2. Second, if class hierarchies are integrated using derivable type equivalence, derivable subtyping, and the chosen type transformations, then their instances can be integrated as well, by applying projections and transformations.

7 Bibliography

[1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[2] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. Int. Symp. on Principles of Programming Languages*, pages 104–118, 1991.

[3] P. Apers, H. Balsters, R. de By, and C. de Vreeze. Inheritance in an object-oriented data model. Memoranda Informatica 90-77, University of Twente, Enschede, The Netherlands, 1990.

[4] H. Balsters, R. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In *Proc. Computing Science in the Netherlands*, pages 62–77. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1991.

[5] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Datatypes, LNCS 173*, pages 51–67. Springer-Verlag, Berlin, 1984.

[6] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. Int. Symp. on Principles of Database Systems*, pages 417–424, 1990.

[7] C. Koster. On infinite modes. *ACM SIGPLAN Notices*, 4(3):109–112, 1969.

[8] C. Lécluse and P. Richard. The O_2 database programming language. In *Proc. Int. Conf. on Very Large Databases*, pages 411–422. Morgan Kaufmann, Palo Alto, CA, 1989.

[9] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1:81–126, 1992.

[10] C. Thieme and A. Siebes. Schema refinement and schema integration in object-oriented databases. Report CS-R9354, CWI, Amsterdam, The Netherlands, 1993 (available by anonymous ftp from ftp.cwi.nl).

[11] C. Thieme and A. Siebes. An approach to schema integration based on transformations and behaviour. Report CS-R9403, CWI, Amsterdam, The Netherlands, 1994 (available by anonymous ftp from ftp.cwi.nl).