Formal Aspects of Computing (1995) : To appear

Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping[1]

Giuseppe Castagna[2]

C.N.R.S. Laboratoire d'Informatique de l'Ecole Normale Supérieure, 45 rue d'Ulm, Paris.

Keywords: type theory, typed foundation of object-oriented programming, second order λ-calculus, bounded polymorphism.

Abstract. In this paper we define an extension of $F_{\leq}$ [CG92] to which we add functions that dispatch on different terms according to the type they receive as argument. In other words, we enrich the explicit parametric polymorphism of $F_{\leq}$ by an explicit "ad hoc" polymorphism (according the classification of [Str67]). We prove that the calculus we obtain, called $F_{\leq}^{\&}$, enjoys the properties of Church-Rosser and Subject Reduction and that its proof system is coherent. We also define a significant subcalculus for which the subtyping is decidable.
This extension has not only a logical interest but it is strongly motivated by the foundation of a broadly used programming style: object-oriented programming. The connections between $F_{\leq}^{\&}$ and object-oriented languages are widely stressed, and the modeling by $F_{\leq}^{\&}$ of some features of the object-oriented style is described, continuing the work of [CGL92].

In this paper we present a calculus where the computation of a function can depend on the type the function is applied to. There are two main motivations to this work. The first is to start a type theoretic foundation of second order "ad hoc" polymorphism. The second is to solve the problem of the "loss of information" in the overloading based model of object-oriented programming [CGL92] in the same way as $F_{\leq}$ solved it for the record-based one [GM94].

[1] Part of this work has appeared under the title "$F_{\leq}^{\&}$: integrating parametric and 'ad hoc' second order polymorphism" in the 4th International Workshop on Database Programming Languages. New York City, August 1993

[2] The author was supported by the grant n. 203.01.56 of the Consiglio Nazionale delle Ricerche - Comitato Nazionale delle Scienze Matematiche to work at LIENS

Correspondence and offprint requests to: Giuseppe Castagna, LIENS, 45 rue d'Ulm, 75005 Paris, FRANCE. e-mail: castagna@dmi.ens.fr.

The overloading-based model has as advantage with respect to the record-based one that it can model multiple dispatch[3] , and therefore it gives a typed account both for the class-based object-oriented languages (such as Smalltalk) and for the generic-functions-based ones (such as CLOS). However we do not lean for any of the two styles: multiple dispatch is indeed essential for generic functions, but it is of great importance also in class-based languages (see [Cas95a]).

The awkwardness of the theory developed in this paper is an example of the disadvantages of the overloading-based with respect to the record-based one

1. Introduction

System F is a language that allows us to write functions that take types as inputs; however these functions depend on their input in a very strict way: computation does not depend on the input type in the sense that different input types will return always the same (type-free) result but in different types. The practical counterpart of this observation is given by the fact that types are thrown away during the computation which is then performed on the *erasure* of the terms. $F_{\leq}$ is a conservative extension of F, which allows the specification of bounds on the types that are passed to a function; the type-checker uses this further information to type the body of the function. But the functions of $F_{\leq}$ still have the same kind of dependence as in System F, since types again disappear during the computation. Here we want to extend $F_{\leq}$ by a type dependency also affecting the computation. We want to have functions that dispatch on different codes according to the type passed as argument. As a side effect, types will no longer be erasable at runtime.

This research fits into a larger framework: In language theory, polymorphism has two orthogonal classifications: "parametric vs. *ad hoc*" (see [Str67]) and "explicit vs. implicit". Parametric polymorphism, i.e. the capability of performing the same code on different types, has been widely studied, both in the explicit form (where types participate directly in the syntax; e.g. System F) and in the implicit one (where types participate via the terms they type; e.g. ML). "Ad hoc" polymorphism, i.e. the capability of performing a different code for each different type, has not received the same attention. In [CGL92], with the definition of the λ&-calculus, we started a theoretical analysis of implicit "ad hoc" polymorphism (on the line of some ideas in [Ghe91]). In this paper we tackle the *explicit* counterpart, by defining $F^{\&}_{\leq}$ a calculus with subtyping, which integrates parametric and "ad hoc" explicit polymorphism. The practical counterpart is the definition of a type discipline that avoids the loss of information in object-oriented programming and fits the paradigms based on generic functions, as explained right below.

Object-oriented programming

This extension is not of mere logical interest but is strongly motivated by the modeling of object-oriented languages and the definition of a type discipline to strongly type them. Let us try to be more specific. In object-oriented languages the computation evolves on objects. Objects are programming items grouped in *classes* and possessing an internal state that is modified by sending messages to the object. When an object receives a message it invokes the method (i.e. code or procedure) associated to that message. The association between methods and messages is described by the class the object belongs

[3] In object-oriented languages, multiple dispatch is the capability of performing the selection of the method not only on the class of the receiver but also on the classes of further parameters.

to. Now, there are two possible ways to implement message-passing: the first is to consider objects as records that associate to each message a method. Thus messages are labels of a record, methods are the values in the fields and message passing corresponds to field selection. This implementation has been extensively studied and corresponds to the "objects as records" analogy of [Car88]. The second way is to consider messages as (identifiers of) special functions which take an object as argument and are able to dispatch on different codes according to the class of that argument (this is done in CLOS: [DG87]). This is the approach taken in [CGL92] where classes are used to type objects and messages are thus overloaded functions, i.e. functions that dispatch on different codes according to the type (the class) of their arguments. There, an overloaded function is a finite collection of ordinary functions (λ-abstractions) that are grouped together to form the different branches, and its type is the set of the types of its branches. More precisely the different branches are glued together by means of "&" (whence the name of λ&-calculus); thus

$$m = (a_1 \& a_2 \& \ldots \& a_n)$$

is an overloaded function with n branches $a_1 \ldots a_n$. If $a_i\colon C_i \rightarrow T_i$ then the type of m is

$$m : \{C_1 \rightarrow T_1, \ldots, C_n \rightarrow T_n\}$$

In object-oriented terms, this means that the "message" m has been associated to a method in the "classes" $C_1, .., C_n$, each method returning a result of type T_i respectively. If we apply m to a value b of type C_j (i.e. if we pass the message m to an object b of class C_j) then the branch (method) a_j is selected and $a_j(b)$ is executed.

Example 1.1. *Suppose that we have defined the following (class) types:*

$\mathrm{2DPoint} \doteq \langle\langle x : \mathit{Int}; y : \mathit{Int}\rangle\rangle$

$\mathrm{3DPoint} \doteq \langle\langle x : \mathit{Int}; y : \mathit{Int}; z : \mathit{Int}\rangle\rangle$

A simple example of a method for these classes is Norm *that can be implemented by the following overloaded function:*

$$\begin{aligned}\mathrm{Norm} \equiv (\ & \lambda \mathrm{self}^{\,2DPoint}.\sqrt{\mathrm{self}.x^2 + \mathrm{self}.y^2} \\ \&\ & \lambda \mathrm{self}^{\,3DPoint}.\sqrt{\mathrm{self}.x^2 + \mathrm{self}.y^2 + \mathrm{self}.z^2} \\ )\ & \end{aligned}$$

whose type is $\{\mathrm{2DPoint} \rightarrow \mathit{Real}, \mathrm{3DPoint} \rightarrow \mathit{Real}\}$*. In this case if we apply* $Norm$ *to an object of type* $2DPoint$ *(i.e. we pass the message* $Norm$ *to that object) then the first branch is selected; if the object has type smaller than or equal to* $3DPoint$ *the second branch is executed.* □

Inheriting methods

In this calculus a subtyping relation is defined on types. Intuitively, a type is smaller than another typer when every value of the former type can be safely used where a value of the latter is expected. Thus in the case as before it may happen that the type C of b does not exactly match one of the C_i's but it is a subtype of one of them. In this case the selected branch is the one that best approximates the type of the argument, i.e. the branch j such that $C_j = \min_{i=1..n}\{C_i | C \leq C_i\}$. On this selection of the minimum relies the mechanism of inheritance, and it corresponds to the usual method look-up of object-oriented languages: in object-oriented terms if we send the message m to the object b of class C then the method defined in the class $C_j = \min_{i=1..n}\{C_i | C \leq C_i\}$ is executed: if this minimum is exactly C, this means that the receiver b uses the method that has been defined in its class; otherwise, that is if this minimum is strictly greater than C, then the receiver uses the method that its class, C, has $inherited$ from the class C_j, which is the least super-class of C in which the message m has been (re)defined.

Example 1.2. *[continued] Suppose that $3DPoint \leq 2DPoint$; this means that $3DPoint$ has been defined as a subclass of $2DPoint$ and that the method defined for $Norm$ in $2DPoint$ (first line of the definition) has been redefined (overridden) by a new method in $3DPoint$ (second line of the definition). Suppose now to have a third class $3DColorPoint \leq 3DPoint$. If an object of this new class is applied to that same $Norm$ then, by the selection of the minimum, the method defined in $3DPoint$ is executed; this means that the class $3DColorPoint$ has inherited the method defined in its superclass $3DPoint$.* □

The problem of loss of information

Suppose we have a message m' which modifies the internal state of a class C_1. Since we are in a functional approach the method in C_1 returns a new object of class C_1. Thus $m'\colon\{..., C_1 \to C_1, ...\}$. Let C_2 be a subclass of C_1 from which it inherits the method at issue. If we pass the message m' to an object of C_2 then the branch defined in C_1 is selected. Since this branch has type $C_1 \to C_1$, the result of message passing has type C_1, rather than C_2 as would be natural. This problem was already pointed out for the record-based model in [Car88] and it is known as the "loss of information problem". In our case the problem is less serious than in Cardelli's calculus: indeed, in the case above we could imagine to add to m' a fake branch $C_2 \to C_2$ which would be used only during the phase of type-checking and then it would be discarded[4] (this has been done in [Cas92]). However this solution is interesting only in practical cases, where there is a finite number of classes; otherwise an infinite branching would be required. Although this solution works whenever the set of classes has a well-founded ordering (as it is always the case in practice) it becomes unmanageable when one starts to distinguish subtyping from subclassing (as done in [CHC90]). In conclusion we need a new type system to account for this problem.

For the record-based model there the solution adopted was to pass to a second order formalism. This yielded the definition of Fun in [CW85], which was further developed in many works (a non exhaustive list includes [CCH$^+$89, Ghe90, CHC90, CL91, CMMS91, BTCGS91, Bru94, PT93]) and, in particular, which gave raise to the definition of $F_{\leq}$ in [CG92].

Here we adopt the same solution w.r.t. the $\lambda\&$-calculus, and we pass to a second order formalism to avoid the problem of loss of information. The idea is to have a type system which types the previous m' in the following way:

$$m'\colon\{..., \forall X{\leq}C_1.X \to X, ...\}$$

For this reason in this paper we define $F_{\leq}^{\&}$ where this type dependency is dealt with in an explicit way[5].

Type dependency

In a programming language a function which performs a dispatch on a type passed as argument would be probably be written as:

[4] Note that such a solution cannot be used also for the record-based model. We cannot replace the function, say, $\lambda x{:}C_1.x$ by $\lambda x{:}C_2.x$ since the latter would no longer accept inputs of type C_1.

[5] The other solution is to deal with it in an implicit way by introducing type schemas *à la* ML, with bounds on the generic variables.

```
Fun(X:*) = case X<T1: exp1
              |  X<T2: exp2
                    :
                    :
              |  X<Tn: exp_n
```

This function executes exp_1 if we pass a type less than or equal to T_1, exp_2 if it is less than or equal to T_2 and so on. The `case` structure suggests a parallel testing. Thus, if there is more than one candidate we select among them the branch with the least bound. In $F^{\&}_{\leq}$ this function is denoted by:

$$(\Lambda X{\leq}T_1.exp_1 \;\&\; \Lambda X{\leq}T_2.exp_2 \;\&\ldots\&\; \Lambda X{\leq}T_n.exp_n)$$

and its type is $\forall X\{T_1.S_1, T_2.S_2, \ldots, T_n.S_n\}$ (where $exp_i\colon S_i$). However this type is a rough approximation yet. Indeed, to obtain a coherent and expressive system, we need strong restrictions on the T_i's and the S_i's.

First of all note that the selected branch may change during the computation. For example take a function f of type $\forall X\{T_1.S_1, T_2.S_2\}$ with $T_2{\leq}T_1$. Consider now the expression $(\Lambda Y \leq T_1.f[Y])$. Since $Y \leq T_1$ we guess that the branch selected in $f[Y]$ will be the one associated to T_1 and thus the type of this expression will be $\forall(X{\leq}T_1)S_1$ (more exactly $\forall(Y \leq T_1)S_1[X := Y]$). But if we pass to the function above the type T_2 then, as Y is bound to T_2, the selected branch will be the second one and the result will have type S_2. System F and $F_{\leq}$ satisfy the subject reduction property, i.e. types are preserved under reductions. If we want reductions to preserve the type also in the new system we must require S_2 to be the same type as S_1. But, it turns out that this is too strong a condition to model object-oriented languages (see the examples in section 6). Thus we adopt a less restrictive discipline, according to which types are allowed to decrease during computation. Thus in the example above it must be possible to deduce $X{\leq}T_2 \vdash S_2{\leq}S_1$. Summing up, the first restriction we impose on an overloaded type $\forall X\{T_i.S_i\}$ is that if $\vdash T_i \leq T_j$ then $X \leq T_i \vdash S_i \leq S_j$ (we call it the *covariance condition*, since it accounts for a longstanding debate on covariance vs. contravariance in the subtyping of the arrow types: see more on it in [CGL92]). Note the use of sequents: the premise records the subtyping relation on the type variables; we call it a type constraint system.

Definition 1.3. *Ø is a type constraint systems (tcs);* dom*(Ø)=Ø. If C is a tcs, $X \notin$* $\mathrm{dom}(C)$ *and for every $Y \in FV(T)$, $Y \in \mathrm{dom}(C)$ then $C \cup \{X{\leq}T\}$ is a type constraint system and* dom*($C \cup \{X{\leq}T\}$)=*$\mathrm{dom}(C) \cup \{X\}$. □

Sometimes we will use the notation $C(X)$ to denote the bound associated to X in C. By the definition above for a given tcs C and a type variable $X \in dom(C)$ there always exists a least non variable type T greater than X. We denote it by $\mathcal{B}(X)_C$ (the $\mathcal{B}$ stands for *bound*). More precisely we have the following definition.

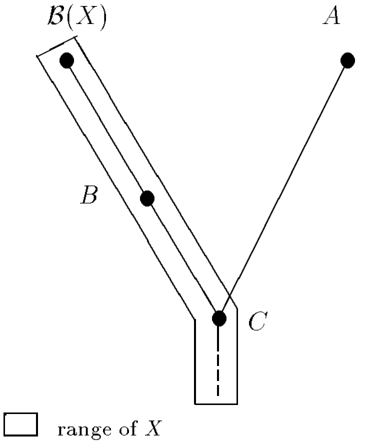
Definition 1.4. *Let C be a tcs and T a* raw[6] *type such that $FV(T) \subseteq dom(C)$ then*
1. $\mathcal{B}(T)_C = T$ if T is not a type variable.
2. $\mathcal{B}(T)_C$ =$\mathcal{B}(C(T))_C$ otherwise. □

In the rest of the paper we omit the subscript C in $\mathcal{B}(T)_C$ when it is clear from the context.

We limit our study to the case where the bounds of an overloaded function range over basic types (e.g. Bool, Int, Real ...). Indeed, the use of arrow types in the bounds

[6] A raw type is a type that may be not well formed. See section 2

poses many non-trivial problems, due to the contravariance of the left argument in the subtyping relation. Therefore the second restriction we impose is that $\forall X\{T_i.S_i\}_{i\in I}$ is well-formed only if for every $i\in I$ $\mathcal{B}(T_i)$ is a basic type[7].



Thus every bound T_i must be an atomic type, i.e. either a basic type or a type variable. When the bound is a type variable, say X, the basic type $\mathcal{B}(X)$ plays an important role, since the set of its subtypes (denoted by $\mathcal{P}(\mathcal{B}(X))$) is the range of X. When we apply an overloaded function to a type, a selection rule picks the branch to execute. As we already said, this rule selects the branches with a bound provably larger than or equal to the type passed as the argument, and among them it chooses the one with the least bound. Some conditions are required to assure that this minimum exists. In $\lambda\&$-calculus this existence was assured by requiring that the bounds had to form a partial downward semi-lattice.[8] But there we had only closed types. Now with type variables this restriction no longer suffices: consider the example of the figure above; it is clear that X and A have no common lower bound (since the only judgment we can prove for X is that $X \leq \mathcal{B}(X)$).

Nevertheless if X takes the value B, it can enter in conflict with A since they have a common lower bound C.

Thus if a variable X appears in an overloaded type as a bound then conflicts must be checked taking into account every type in $\mathcal{P}(\mathcal{B}(X))$. To this purpose we require that every set of bounds satisfies the property of $\cap$-*closure*, defined as follows:

Definition 1.5. *Let C be a type constraint system. Given a set of atomic types $\{A_i\}_{i\in I}$ we write $C \vdash \{A_i\}_{i\in I}$ $\cap$-closed iff for all $i,j \in I$ if $\mathcal{B}(A_i)_C \Downarrow \mathcal{B}(A_j)_C$ then there exists $h \in I$ such that $C \vdash A_h = A_i \cap A_j$.* □

Here $C \vdash A_h = A_i \cap A_j$ means that from C it is provable that A_h is the g.l.b. of A_i and A_j (i.e. we need a derivation for $C \vdash A_h \leq A_i$, one for $C \vdash A_h \leq A_h$ and one for $C \vdash A \leq A_h$ for every A which is a common lower bound of A_i and A_h—note that they is a finite number of such A), and $B_1 \Downarrow B_2$ that B_1 and B_2 have a common lower bound (in this case we need the proof only for two judgments).

Note that $\cap$-closure is quite a draconian restriction. Indeed $\cap-$closed sets of bounds have a very precise form (see proposition 2.5): they are partial downward semi-lattices, i.e. formed by disjoint unions of downward semi-lattices. These semi-lattices are divided in two parts: the upper part is a semi-lattice formed only by basic types; the lower part is formed by a chain of type variables starting from the least element of a semi-lattice of basic types. Any of these two parts may be missing. A pictorial representation of the situation is given in figure 1.

[7] The major drawback of this restriction is that we cannot obtain the quantification of System F as a special case of the overloaded one and thus we will be obliged to add it explicitly. See also sections 6.3 and 7

[8] A set S is a partial downward semi-lattice iff for all $a,b \in S$ if $a \Downarrow b$ then $a \cap b \in S$. Here $a \Downarrow b$ means that a and b have a common lower bound (in S) and $a \cap b$ denotes their greatest lower bound.

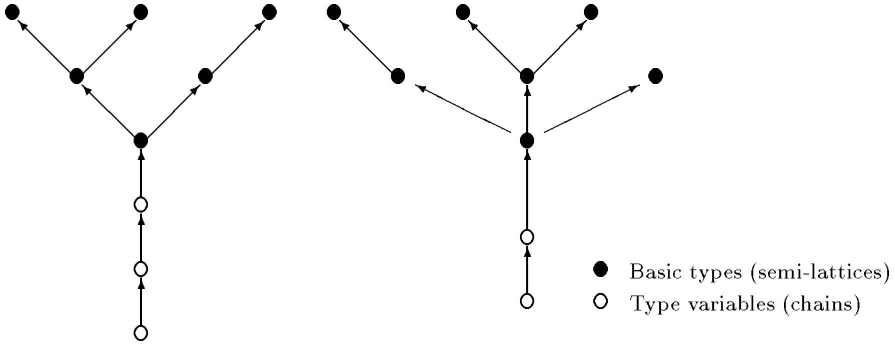


Fig. 1. Meet-closed sets of atomic types

Remark that $\cap-$closure is always relative to a tcs C since it determines the order for type variables (we assume that the order on basic types is predefined).

2. Type system

In this section we describe the type system. We first define the raw types. Among them we select the types, i.e. those raw types that do not contain overloaded types not satisfying the three rules we hinted in the introduction. In other terms $\forall X\{A_i.T_i\}_{i\in I}$ must:

1. have bounds ranging over basic types, i.e. for each $i\in I$, $\mathcal{B}(A_i)$ must be a basic type.
2. have a $\cap$-closed set of bounds.
3. satisfy covariance, i.e. if $A_i \leq A_j$ then $X{\leq}A_i \vdash T_i{\leq}T_j$

We suppose we have a predefined ordering on basic types which forms a partial lattice. This partial order is extended to higher types by a set of subtyping rules that are mutually recursive with those selecting the types.

Raw Types

$$\begin{array}{lcll}
A & ::= & X \mid B & \text{(atomic types [B basic types])} \\
\\
T & ::= & A \mid \mathsf{Top} & \text{(raw $F^{\&}_{\leq}$ types)} \\
 & \mid & T \to T & \\
 & \mid & \forall(X{\leq}T)T & \\
 & \mid & \forall X\{A_1.T_1, \ldots, A_n.T_n\} & \text{(also denoted by $\forall X\{A_i.T_i\}_{i=1..n}$)}
\end{array}$$

Judgments

We have three kinds of judgment: for type well-formedness ($C \vdash T$ type), for the subtyping relation ($C \vdash T{\leq}T'$) and for the typing relation ($C \vdash a{:}\,T$). We call the first two kinds of judgments *type judgments*. Throughout the paper we also use some informal judgements: for example "$C \vdash T = \min_{i\in I}\{T_i\}$" stands for "$T \in \{T_i\}_{i\in I}$ and for all $i{\in}I$ $C \vdash T \leq T_i$".

Types

(Basic$_{type}$)
$$C \vdash B \text{ type}$$

(Top$_{type}$)
$$C \vdash \mathsf{Top} \text{ type}$$

(Vars$_{type}$)
$$\frac{C \vdash T \text{ type}}{C \cup \{X {\leq} T\} \vdash X \text{ type}} \qquad (*)$$

($\rightarrow_{type}$)
$$\frac{C \vdash T \text{ type} \qquad C \vdash T' \text{ type}}{C \vdash T \rightarrow T' \text{ type}}$$

($\forall_{type}$)
$$\frac{C \cup \{X {\leq} T\} \vdash T' \text{ type} \qquad C \vdash T \text{ type}}{C \vdash \forall(X {\leq} T) T' \text{ type}} \qquad (*)$$

($\{\,\}_{type}$)
$$\frac{\begin{array}{c} C \vdash A_i \text{ type} \\ C \vdash \{A_i\}_{i=1..n}\ \cap\text{-closed} \\ C \cup \{X {\leq} A_i\} \vdash T_i \text{ type} \\ \text{if } C \vdash A_i \leq A_j \text{ then } C \cup \{X {\leq} A_i\} \vdash T_i {\leq} T_j \end{array}}{C \vdash \forall X\{A_1.T_1, \ldots, A_n.T_n\} \text{ type}} \qquad (**)$$

$(*)$ $\quad X \notin dom(C)$
$(**)$ $\quad \mathcal{B}(A_h)_C$ *is a basic type for* $h = 1..n$ *and* $X \notin dom(C)$ *and* $i, j \in [1..n]$

Subtyping

(refl)
$$\frac{C \vdash T \text{ type}}{C \vdash T {\leq} T}$$

(trans)
$$\frac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$$

(taut)
$$\frac{C \vdash T \text{ type}}{C \cup \{X {\leq} T\} \vdash X {\leq} T} \qquad (*)$$

(Top)
$$\frac{C \vdash T \text{ type}}{C \vdash T {\leq} \mathsf{Top}}$$

($\rightarrow$)
$$\frac{C \vdash T_1' {\leq} T_1 \qquad C \vdash T_2 {\leq} T_2'}{C \vdash T_1 \rightarrow T_2 {\leq} T_1' \rightarrow T_2'}$$

($\forall$)
$$\frac{C \vdash T_1' {\leq} T_1 \qquad C \cup \{X {\leq} T_1'\} \vdash T_2 {\leq} T_2' \qquad C \vdash \forall(X {\leq} T_1) T_2 \text{ type}}{C \vdash \forall(X {\leq} T_1) T_2 {\leq} \forall(X {\leq} T_1') T_2'} \qquad (*)$$

($\{\,\}$)
$$\frac{\begin{array}{c} C \vdash \forall X\{A_j.T_j\}_{j \in J} \text{ type} \quad C \vdash \forall X\{A_i'.T_i'\}_{i \in I} \text{ type} \\ \text{for all } i \in I \text{ exists } j \in J \text{ s.t.} C \vdash A_i' {\leq} A_j \quad C \cup \{X {\leq} A_i'\} \vdash T_j {\leq} T_i' \end{array}}{C \vdash \forall X\{A_j.T_j\}_{j \in J} {\leq} \forall X\{A_i'.T_i'\}_{i \in I}} \qquad (*)$$

Types are considered equal modulo the order in overloaded types. The substyping rules above define a pre-order. Antisymmetry does not hold but on atomic types (therefore lub’s and glb’s are univocally defined).

2.1. Some useful results

Theorem 2.1. If $C \vdash T {\leq} T'$ then $C \vdash T$ type and $C \vdash T'$ type

Proof. By induction on the depth of the proof of $C \vdash T {\leq} T'$. □

Let us introduce some terminology: we say that two types *have the same shape* if they are both constant types or both type variables, or both Top, or both arrow types, or they are both overloaded or both parametric types

The following result on the form of the judgements will be frequently used in the rest of the paper

Proposition 2.2. Let $C \vdash T_1 \leq T_2$. Then

1. If T_1 is not a variable then T_2 either is Top or it has the same shape as T_1
2. If T_2 is not Top then T_1 either is a variable or it has the same shape as T_2

Proof. By induction on the depth of the proof of $C \vdash T_1 \leq T_2$, performing a case analysis on the last applied rule of the proof. □

Another useful fact that will be extensively used in the proofs of this paper is the following one:

Proposition 2.3. If $C \vdash T_1 \leq T_2$ then $C \vdash \mathcal{B}(T_1)_C \leq \mathcal{B}(T_2)_C$

Lemma 2.4. If $C \vdash X \leq Y$ then $\mathcal{B}(X)_C = \mathcal{B}(Y)_C$

Proof. An easy induction on the number of steps of the definition of $\mathcal{B}(X)_C$. □

The following proposition describes the the form of the ∩-closed set of types:

Proposition 2.5. If $C \vdash \{A_i\}_{i \in I}$∩-closed then for any pair of elements A_i and A_j such that $\mathcal{B}(A_i)_C \Downarrow \mathcal{B}(A_j)_C$ one of these cases must hold:

1. $\mathcal{B}(A_i)_C$ and $\mathcal{B}(A_j)_C$ are unrelated (w.r.t. the subtyping relation), A_i and A_j are both basic types and their g.l.b. is in $\{A_i\}_{i \in I}$
2. $\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$ and both A_i and A_j are basic types (or the reverse).
3. $\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$, A_i is a variable and A_j is a basic type.
4. $\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$, A_i and A_j are both variables and $C \vdash A_i \leq A_j$ (or the reverse).

Proof. Let us examine all the possible cases:

1. A_i and A_j are both basic types. Then all the possible cases are covered by the first two points of the proposition.
2. A_i is a variable and A_j is a basic type.[8] Then we want to prove that $\mathcal{B}(A_i)_C \leq A_j$. Consider $A_h = A_i \cap A_j$. Since $C \vdash A_h {\leq} A_i$ then by proposition 2.2 A_h is a variable too. By lemma 2.4 $\mathcal{B}(A_h)_C$ = $\mathcal{B}(A_i)_C$. Since $C \vdash A_h {\leq} A_j$ then by proposition 2.3 we obtain the result

[8] Without loss of generality, we can consider for this case and for the case 4 that $\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$ holds. Thus in this proof and in those that follow we will skip the reverse case.

3. A_i and A_j are both variables. Consider $A_h = A_i \cap A_j$. By proposition 2.2 A_h is also a variable and by lemma 2.4 $\mathcal{B}(A_h)_C = \mathcal{B}(A_i)_C = \mathcal{B}(A_j)_C$. Thus both A_i and A_j appear in the chain from A_h to $\mathcal{B}(A_h)_C$. Therefore either $C \vdash A_i \leq A_j$ or $C \vdash A_j \leq A_i$ holds, according to the order they appear in the chain.

□

2.2. Transitivity elimination

The rules of subtyping given above do not describe a deterministic algorithm: a subtyping judgment does not univocally determine neither the rule to prove it nor the parameters that such a rule must have. In particular non-determinism is introduced by the rules (refl) and (trans):

Consider the judgment $C \vdash T \leq T$; if T is not a variable nor Top then the judgment can be proved by at least two different derivations, one consisting just of the rule (refl) the other obtained by applying the structural rule for T (e.g. (→) if T is an arrow type) and the rule (refl) to the components of T. This kind of non-determinism can be easily solved by choosing either to use (refl) as soon as possible or to use it as late as possible (i.e. only on atomic types). We choose this second solution thus we substitute the rule (refl) above by the following one:

(refl) $$\frac{C \vdash A \text{ type}}{C \vdash A \leq A}$$

It is then very simple to prove that this new system is sound and complete w.r.t. the previous one: soundness is obvious and completeness is given by the following lemma:

Lemma 2.6. For each C and T such that $C \vdash T$ type the judgment $C \vdash T \leq T$ is provable using reflexivity only on atomic types.

Proof. A straightforward induction on the structure of T □

Also the rule (trans) produces a non-determinism similar to the one of (refl): we have always the choice to apply transitivity or to push it to the subcomponents. But, besides that, (trans) introduces a deeper form of non-determinism quite harder to eliminate. Indeed, the (trans) rule does not respect the so-called "sub-formula property", according to which all the types appearing at the premises of a rule must appear in its consequence, too. When proving $T_1 \leq T_3$ by transitivity, a new level of non-determinism is introduced by the choice of the *intermediate type* T_2 such that $T_1 \leq T_2$ and $T_2 \leq T_3$.

The reader will have recognized in it a cut elimination problem. Indeed, transitivity elimination in subtyping systems corresponds to cut elimination in Gentzen's sequent calculus for the first order logic. Both of them lead to a coherence result of the corresponding proof system, by returning a canonical derivation for each provable judgment. The resemblance is even stronger since we can use the Gentzen's technique for cut elimination to prove also transitivity elimination. Namely, we define a weakly normalizing rewriting system on the derivations of subtyping judgments. This system will push the transitivity rules towards the leaves of the derivation; whenever it has to choose between pushing transitivity up into a left or a right subderivation it (arbitrarily) chooses the one on the right. The derivations in normal form will have all the (trans) rules applied to a leaf of the derivation tree.

Since it is difficult to work directly with derivations, we use the Curry-Horward isomorphism [How80] to define a set a terms to univocally codify subtyping derivations. We follow for their definition [CG92], where these terms are called *coercion expressions*.

The syntax of the coercion expressions is:

$$c ::= \;\; K_{B_1B_2} \mid Id_A \mid X_T \mid \mathsf{Top}_T \mid c \to c' \mid \forall(X{\leq}c)c' \mid c\,c' \mid \forall^{\phi}_{T}X\{c_1.c'_1, ..., c_n.c'_n\}$$

where ϕ denotes a total function between two sets of indexes $\phi : I \to J$.

We next show how to use coerce expressions to codify derivations. In the rules that follow we do not consider the judgements of type formation ($C \vdash T\,\mathrm{type}$) and we concentrate only on the subtyping judgements. Considering them would greatly complicate the exposition, without bringing any benefit: firstly the rules defining type formation describe a deterministic algorithm (note indeed that type formation uses subtyping only on atomic types, thus there is not a real mutual recursion), and thus they do not pose any coherence problem; secondly, all the proofs in the rest of this section will work on a given type and on its syntactical sub-formulas; if we suppose by hypothesis that the type is well formed then the proofs will be valid also when restricted to well formed types (sub-formulae of well formed types are well formed types).

Thus the derivations we codify involve only subtyping judgements and work under the hypothesis that all the types appearing in them are well-formed. We also use (refl) defined only for atomic types.

$$\text{(basic)} \qquad C \vdash K_{B_1B_2}\!: B_1 \leq B_2 \qquad (\dagger)$$

$$\text{(refl)} \qquad C \vdash Id_A\!: A{\leq}A$$

$$\text{(trans)} \qquad \frac{C \vdash c\!: T_1 \leq T_2 \qquad C \vdash c'\!: T_2 \leq T_3}{C \vdash c'\,c\!: T_1 \leq T_3}$$

$$\text{(taut)} \qquad C \cup \{X{\leq}T\} \vdash X_T\!: X{\leq}T$$

$$\text{(Top)} \qquad C \vdash \mathsf{Top}_T\!: T{\leq}\mathsf{Top}$$

$$(\to) \qquad \frac{C \vdash c_1\!: T'_1{\leq}T_1 \qquad C \vdash c_2\!: T_2{\leq}T'_2}{C \vdash c_1 \to c_2\!: T_1 \to T_2{\leq}T'_1 \to T'_2}$$

$$(\forall) \qquad \frac{C \vdash c_1\!: T'_1{\leq}T_1 \qquad C \cup \{X{\leq}T'_1\} \vdash c_2\!: T_2{\leq}T'_2}{C \vdash \forall(X{\leq}c_1)c_2\!: \forall(X{\leq}T_1)T_2{\leq}\forall(X{\leq}T'_1)T'_2}$$

$$(\{\,\}) \qquad \frac{\forall i \in I \qquad C \vdash c_i\!: A'_i{\leq}A_{\phi(i)} \quad C \cup \{X{\leq}A'_i\} \vdash c'_i\!: T_{\phi(i)}{\leq}T'_i}{C \vdash \forall^{\phi}_{\forall X\{A_j.T_j\}}X.\{c_i.c'_i\}_{i\in I}\!: \forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A'_i.T'_i\}_{i\in I}} \qquad (\dagger\dagger)$$

(†) *for all basic types* B_1, B_2 *such that* $B_1{\leq}B_2$
(††) $\phi\!: I \to J$ *total*

Note that the term associated to transitivity is the composition of the terms associated to the sub-derivations.

The last rule shows the use of the function ϕ: during the subtyping of two overloaded types, ϕ associates each branch of the greater overloaded type with the branch in the smaller type to which it has been compared in the proof of subtyping. Note that this information would not suffice to univocally determine the derivation codified by a given coercion expression; in case of overloaded types we need also to know the type on the left-hand side of the relation, which is recorded in the lower index of $\forall$.

Proposition 2.7. There is a 1-1 correspondence between well-typed coerce expressions and subtyping derivations.

Proof. A simple induction on the rules[9]. □

2.2.1. The rewriting system

We now define a rewriting system on the derivations of subtyping judgements. In view of the proposition 2.7 this is equivalent to defining it directly on the coerce expressions. We borrow the rewriting system from [CG92], to which we add the rules ($\{\}$') and ($\{\}$") to deal with overloaded types. In the rules that follow we suppose that $C \vdash c\colon S{\leq}T$ and $C \vdash c_i\colon A_i \leq A'_{\phi(i)}$:

$$\begin{array}{llcl}
\text{(Assoc)} & (c\,d)\,e & \leadsto & c\,(d\,e) \\
(\rightarrow') & (c \rightarrow d)\,(c' \rightarrow d') & \leadsto & (c'\,c) \rightarrow (d\,d') \\
(\rightarrow'') & (c \rightarrow d)\,((c' \rightarrow d')\,e) & \leadsto & ((c'\,c) \rightarrow (d\,d'))\,e \\
(\forall') & (\forall(X{\leq}c)d)\,(\forall(X{\leq}c')d') & \leadsto & \forall(X{\leq}c'\,c)(d\,d'[X_T\colon= c\,X_S]) \\
(\forall'') & (\forall(X{\leq}c)d)\,((\forall(X{\leq}c')d')\,e) & \leadsto & (\forall(X{\leq}c'\,c)(d\,d'[X_T\colon= c\,X_S]))\,e
\end{array}$$

$$\begin{array}{ll}
(\{\}') & (\forall^{\phi}_{T}X\{c_i.d_i\}_{i\in I})\,(\forall^{\psi}_{T'}X\{c'_j.d'_j\}_{j\in J}) \\
 & \qquad\leadsto\quad \forall^{\psi\circ\phi}_{T'}X\{c'_{\phi(i)}\,c_i.d_i\,(d'_{\phi(i)}[X_{A'_{\phi(i)}} := X_{A_i}]\}_{i\in I} \\
(\{\}'') & (\forall^{\phi}_{T}X\{c_i.d_i\}_{i\in I})\,((\forall^{\psi}_{T'}X\{c'_j.d'_j\}_{j\in J})\,e) \\
 & \qquad\leadsto\quad (\forall^{\psi\circ\phi}_{T'}X\{c'_{\phi(i)}\,c_i.d_i\,(d'_{\phi(i)}[X_{A'_{\phi(i)}} := X_{A_i}]\}_{i\in I})\,e
\end{array}$$

A simple analysis of the rules shows that the normal forms of this rewriting system are subterms of $(c \rightarrow d)\,e_1\,\ldots\,e_n$ or of $(\forall(X{\leq}c)d)\,e_1\,\ldots\,e_n$ or of $(\forall^{\phi}_{T}X\{c_i.d_i\})\,e_1\,\ldots\,e_n$ where c, c_i, d, d_i are in normal form and $e_1, \ldots, e_n$ are either X_t or Top_T of $K_{BB'}$ (composition is right associative). These normal forms correspond to derivations in which every left premise of a (trans) rule is a leaf. Thus the rewriting system pushes the transitivity up to the leaves. It remains to prove two facts:

1. The rewriting system is sound, i.e. it rewrites a valid derivation for a certain judgment into another valid derivation for the same judgment. By the Curry-Howard isomorphism this is equivalent to proving the subject reduction theorem for the calculus of the coercion expressions; namely we have to show that a well typed coerce expression rewrites only to well typed coerce expressions of the same type.
2. The rewriting system is weakly normalizing. In this case there exists a reduction strategy which transforms every derivation into another that proves the same judgment and is in normal form (i.e. with the (trans) rules at the right places).

[9] Strictly speaking this theorem is true modulo weakenings of the tcs'

2.2.2. Soundness of the rewriting system

The proof of the soundness of the rewriting system is very similar to the corresponding one in [CG92]. We first have to prove the following lemmas:

Lemma 2.8. (weakening) If $C \vdash c{:}\,\Delta$ is provable and $C \cup \{X \leq T\}$ is a tcs then also $C \cup \{X{\leq}T\} \vdash c{:}\,\Delta$ is provable.

Proof. By a simple induction on the proof of $C \vdash c{:}\,\Delta$ □

Lemma 2.9. (substitution) If $C \cup \{X{\leq}\,T\} \vdash c{:}\,U \leq V$ and $C \cup \{X{\leq}\,S\} \vdash d{:}\,X \leq T$ are provable then $C \cup \{X{\leq}S\} \vdash c[X_T{:}= d]{:}\,U \leq V$ is provable too.

Proof. By induction on the structure of c. We only detail the proof when c is a variable; all the other cases are either trivial ($K_{BB'}, \mathsf{Top}_T$ and Id_A) or they are solved by a straightforward use of the induction hypothesis ($\rightarrow$, $\forall$, $\forall^{\phi}_{T}$).
There are two subcases:
1. $c{\equiv}X_T$. The result becomes $C{\cup}\{X{\leq}S\} \vdash d{:}\,X \leq T$ which is satisfied by hypothesis.
2. $c{\equiv}Y_V$. The hypothesis becomes $C{\cup}\{X{\leq}T\} \vdash Y_V{:}Y \leq V$. Therefore $C \vdash Y_V{:}Y \leq V$. By a weakening (lemma 2.8) we obtain the result $C \cup \{X{\leq}S\} \vdash Y_V{:}Y \leq V$. □

Now we are able to prove the soundness of the rewriting system

Theorem 2.10. If $c \overset{*}{\leadsto} d$ and $C \vdash c{:}\,\Delta$ then $C \vdash d{:}\,\Delta$

Proof. Follow the proof of section 5.2 in [CG92]. For the cases ($\{\}$') and ($\{\}$") lemma 2.9. The detailed proof of these cases can be found in [Cas94]. □

2.2.3. Weak normalization

The task of proving that the rewriting system is weakly normalizing is very simple since all the work has already been done in [CG92]: define

$$\begin{aligned}
size(A) = size(\mathsf{Top}) &\overset{def}{=} 1\\
size(S \rightarrow T) = size(\forall(X{\leq}S)T) &\overset{def}{=} size(S) + size(T)\\
size(\forall X\{A_i.T_i\}_{i\in I}) &\overset{def}{=} \sum_{i\in I}(size(A_i) + size(T_i))
\end{aligned}$$

Let m and m' be two multisets of natural numbers; define

$$m < m' \overset{def}{\Longleftrightarrow} \forall n' \in m' \exists n \in m \;\; n < n'$$

Definition 2.11. ([CG92]) *Define the* intermediate type *of a coerce composition* $d\,e$*, where* $e{:}\,S{\leq}\,T$ *and* $d{:}T{\leq}\,U$*, as the type* T*. Then the* complexity measure *of a coerce expression* c *is the multiset of the sizes of the intermediate types of all the redexes of* c*, modulo* (Assoc). □

Theorem 2.12. Every innermost strategy for $\leadsto$ strictly decreases the complexity measure and thus terminates.

Proof. The proof is strictly the same as the one of section 5.3.3 in [CG92] modulo some slight modifications for the cases involving overloaded types. □

2.3. Subtyping algorithm and coherence of the system

Consider the following rewriting rules

(id$_l$)	$Id_T\, c$	$\leadsto$	c
(id$_r$)	$c\, Id_S$	$\leadsto$	c
(bas$'$)	$K_{BC}\, K_{AB}$	$\leadsto$	K_{AC}
(bas$''$)	$K_{BC}\, (K_{AB}\, c)$	$\leadsto$	$K_{AC}\, c$
(top)	$\mathsf{Top}_T\, c$	$\leadsto$	Top_S
(varTop)	X_{Top}	$\leadsto$	Top_X

These rules perform some cleaning of the derivations, basically by erasing useless coercions.

This set of rules clearly constitutes a strongly normalizing rewriting system (use as metrics for the coercion expressions the lexicographical order of the pairs formed by the number of compositions in the expression and by the number of variables occurring in it). Furthermore no rule increases the complexity measure given in the previous section for weak normalization, and they are all sound. Therefore we can safely add these rules to the previous rewriting system: all the results of the previous section still hold. In the rest of this section we will always consider the rewriting system formed by the old rules and those introduced above.

2.3.1. The shape of the normal forms

It is very important to analyze the shape of the normal forms of the composed rewriting system. We have the following theorem:

Proposition 2.13. Every well-typed coerce expression in normal form has the form $c_0\, c_1\, \ldots\, c_n$ with $n \geq 0$, where c_0 can be any coerce expression different from composition, whose subformulae are in normal form, and $c_1 \ldots c_n$ are variables.

Proof. This proposition can be easily proved by induction on n (where n is the number of outer compositions of the normal form at issue). For $n = 0$ the result is obvious. The inductive case ($n > 0$) is proved by a case analysis on the shape of c_0, by using proposition 2.2 and the reduction rules. First of all note that because of the rewriting rules (top) and (id$_l$) c_0 can be neither Top_T nor Id_A:

$c_0 \equiv X_T$. Consider c_1. It cannot be a composition because of (Assoc). By proposition 2.2 it can be nothing but a variable: indeed we have that $c_1 : S \leq X$ thus S must be a type variable, say, Y and therefore $c_1 \equiv Y_X$. The result follows by induction hypothesis.

$c_0 \equiv K_{B_1 B_2}$. Consider c_1: it cannot be a composition because of (Assoc); it cannot be a constant because of (bas$'$) if $n = 1$, because of (bas$''$) if $n > 1$; it cannot be Top_T or $c \to c'$ or $\forall(X \leq c)c'$ or $\forall^{\phi}_{T} X\{c_1.c'_1, \ldots, c_n.c'_n\}$ because of proposition 2.2. Thus it can be but a variable. The result follows by induction hypothesis.

$c_0 \equiv c \to c'$. Consider c_1: it cannot be a composition because of (Assoc); it cannot be a $d \to d'$ because of ($\to'$) if $n = 1$, because of ($\to''$) if $n > 1$; it cannot be Top_T or $c \to c'$ or $\forall(X \leq c)c'$ or $\forall^{\phi}_{T} X\{c_1.c'_1, \ldots, c_n.c'_n\}$ because of proposition 2.2. Thus it can be but a variable. The result follows by induction hypothesis.

All the other cases are solved as the last two cases. □

This theorem has two important consequences: the coherence of the proof system for the subtyping relation and the definition of a subtyping algorithm.

2.3.2. Coherence

Lemma 2.14. For every provable subtyping judgment there exists only one coerce expression in normal form proving it.

Proof. We follow the pattern of the proof of the corresponding proposition in [CG92]. Let c be a well typed coercion expression in normal form. From propositions 2.13 and 2.2 it follows almost immediately that we have only these possible cases:

1. if $c\colon A \leq A$ then $c \equiv Id_A$
2. if $c\colon X \leq Y$ then c is a composition of variables, which is determined in an unique way by the tcs.
3. if $c\colon B_1 \leq B_2$ then $c \equiv K_{B_1 B_2}$
4. if $c\colon S \to S' \leq T \to T'$ then c is a $\to$ coercion.
5. if $c\colon \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2$ then c is a $\forall$ coercion.
6. if $c\colon \forall X\{A_j.T_j\}_{j \in J} \leq \forall X\{A'_i.T'_i\}_{i \in I}$ then c is a $\forall^{\phi}_{T}$ coercion.
7. if $c\colon X \leq B$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of class 3 if $\mathcal{B}(X) \neq B$
8. if $c\colon X \leq T \to T'$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of class 4 if $\mathcal{B}(X) \neq T \to T'$
9. if $c\colon X \leq \forall(X \leq T_1)T_2$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of class 5 if $\mathcal{B}(X) \neq \forall(X \leq T_1)T_2$
10. if $c\colon X \leq \forall X\{A'_i.T'_i\}_{i \in I}$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of class 6 if $\mathcal{B}(X) \neq \forall X\{A'_i.T'_i\}_{i \in I}$
11. if $c\colon T \leq \mathsf{Top}$ then c is Top_T

After this simple observation then the result can be proved by induction on the structure of c. □

Theorem 2.15. (coherence) Let Π_1 and Π_2 be two proofs of the same subtyping judgment $C \vdash \Delta$. If c_1 and c_2 are the corresponding coerce expressions then c_1 and c_2 are equal modulo the rewriting system.

Proof. By the weak normalization there exist two coercion expressions in normal form d_1 and d_2 such that $c_1 \overset{*}{\leadsto} d_1$ and $c_2 \overset{*}{\leadsto} d_2$. By the soundness of the rewriting system (theorem 2.10) it follows that $C \vdash d_1\colon \Delta$ and $C \vdash d_2\colon \Delta$. But then by lemma 2.14 we have that $d_1 \equiv d_2$ (note that this constitutes also a proof that $\leadsto$ is Church-Rosser.) □

2.3.3. Subtyping algorithm

Consider once more the normal forms of proposition 2.13. These correspond to derivations in which every application of a (trans) rule has as left premise an application of the rule (taut). From this observation one directly derives the definition of the following subtyping algorithm:

(AlgRefl) $$C \vdash_{\mathcal{A}} X \leq X$$

(AlgTrans) $$\frac{C \vdash_{\mathcal{A}} C(X) \leq T}{C \vdash_{\mathcal{A}} X \leq T}$$

(AlgTop) $$C \vdash_{\mathcal{A}} T \leq \mathsf{Top}$$

$$(\text{Alg}{\rightarrow}) \qquad \frac{C \vdash_{\mathcal{A}} T_1' {\leq} T_1 \qquad C \vdash_{\mathcal{A}} T_2 {\leq} T_2'}{C \vdash_{\mathcal{A}} T_1 \rightarrow T_2 {\leq} T_1' \rightarrow T_2'}$$

$$(\text{Alg}\forall) \qquad \frac{C \vdash_{\mathcal{A}} T_1' {\leq} T_1 \qquad C \cup \{X {\leq} T_1'\} \vdash_{\mathcal{A}} T_2 {\leq} T_2'}{C \vdash_{\mathcal{A}} \forall(X {\leq} T_1)T_2 \leq \forall(X {\leq} T_1')T_2'} \qquad (*)$$

$$(\text{Alg}\{\ \}) \ \frac{\text{for all } i \in I \text{ exists } j \in J \text{ s.t.} C \vdash A_i' {\leq} A_j \quad C \cup \{X {\leq} A_i'\} \vdash T_j {\leq} T_i'}{C \vdash \forall X \{A_j.T_j\}_{j \in J} {\leq} \forall X \{A_i'.T_i'\}_{i \in I}} \qquad (*)$$

(∗) $X \notin dom(C)$

This set of rules denotes a deterministic algorithm since the form of the input —the judgment one has to prove— unequivocally determines the rule that must be used and all the parameters of any recursive calls

This algorithm (which is a semi-decision procedure) is sound and complete w.r.t. our first system. This means that the sets of provable judgments of the two systems are the same. This is stated by the following theorem:

Theorem 2.16. $C \vdash_{\mathcal{A}} \Delta \Longleftrightarrow C \vdash \Delta$

Proof. Soundness (⇒) is easily proved by induction on the depth of the derivation of $C \vdash_{\mathcal{A}} \Delta$. Completeness (⇐) stems directly from the work of this section: take any proof of $C \vdash \Delta$, apply to it the complete rewriting system with an innermost strategy; replace in the obtained normal form all the sequences of (taut) (trans) rules by an (AlgTrans) rule; add the index $\mathcal{A}$ to every turnstile and you have obtained a proof for $C \vdash_{\mathcal{A}} \Delta$. □

3. Terms

In this section we describe the terms of the language. We start by the definition of the *raw terms*, among which we distinguish the *terms*, i.e. those raw terms that possess a type. Roughly speaking, (raw) terms are divided in three classes: terms of the simply typed λ-calculus, terms for parametric polymorphism and terms for overloading. Overloaded functions are built in a list fashion, starting by an *empty* overloaded function ε and concatenating new branches by &. The &'s are indexed by a list of types which is used to type the term and to perform the selection of the branch.

Indexes

$$I ::= [A_1.T_1 \parallel \ldots \parallel A_n.T_n]$$

Raw Terms

$$\begin{array}{rcll} a & ::= & x^T \mid (\lambda x^T.a) \mid a(a) & \text{simply typed } \lambda\text{-calc} \\ & \mid & \mathsf{top} \mid \Lambda X {\leq} T.a \mid a(T) & F_{\leq} \\ & \mid & \varepsilon \mid (a \&^I a) \mid a[A] & \text{overloading} \end{array}$$

We required that the bounds of an overloaded function range over atomic types. Therefore the argument of an overloaded function can be restricted to be an atomic type ($a[A]$) since a term of the form, say, $a[S \rightarrow T]$ would be surely rejected by the type checker.

Terms

We use the meta notation: $a[x := b]$, $a[X := S]$, $T[X := S]$ for substitutions and ∪ for set-theoretic union. Also we use $C \vdash a\colon S \leq T$ to denote that $C \vdash a\colon S$ and $C \vdash S \leq T$. Type substitutions are performed on indexes, too. Terms are selected by the rules below; since term variables are indexed by their type, the rules do not need assumptions of the form $(x\colon T)$:

[Vars] $$C \vdash x^T\colon T \qquad\qquad C \vdash T \text{ type}$$

[$\rightarrow$Intro] $$\frac{C \vdash a\colon T'}{C \vdash (\lambda x^T.a)\colon T \rightarrow T'} \qquad\qquad C \vdash T \text{ type}$$

[$\rightarrow$Elim] $$\frac{C \vdash a\colon T' \qquad C \vdash b\colon S' \leq S}{C \vdash a(b)\colon T} \qquad\qquad \mathcal{B}(T')_C = S \rightarrow T$$

[Top] $$C \vdash \mathsf{top}\colon \mathsf{Top}$$

[$\forall$Intro] $$\frac{C \cup \{X \leq T\} \vdash a\colon T'}{C \vdash \Lambda X \leq T.a\colon \forall(X \leq T)T'} \qquad\qquad C \vdash T \text{ type}$$

[$\forall$Elim] $$\frac{C \vdash a\colon T' \qquad C \vdash S' \leq S}{C \vdash a(S')\colon T[X := S']} \qquad\qquad \mathcal{B}(T')_C = \forall(X \leq S)T$$

[ε] $$C \vdash \varepsilon\colon \forall X\{\}$$

[$\{\}$Intro] $$\frac{C \vdash a\colon S_1 \leq \forall X\{A_i.T_i\}_{i \leq n} \quad C \vdash b\colon S_2 \leq \forall(X \leq A)T}{C \vdash (a\&^{[A_1.T_1\|\ldots\|A_n.T_n\|A.T]}b)\colon \forall X(\{A_i.T_i\}_{i \leq n} \cup \{A.T\})} \qquad (\ddagger)$$

[$\{\}$Elim] $$\frac{C \vdash a\colon T \qquad C \vdash A_j = min_{i \in I}\{A_i | C \vdash A \leq A_i\}}{C \vdash a[A]\colon T_j[X := A]} \qquad (\ddagger\ddagger)$$

($\ddagger$) $C \vdash \forall X(\{A_i.T_i\}_{i \leq n} \cup \{A.T\})$ type
($\ddagger\ddagger$) $\mathcal{B}(T)_C = \forall X\{A_i.T_i\}_{i \in I}$

Note the form of the premises in the rule [$\{\}$Intro]; we cannot require that the components of an & must have the same type as the one specified in the index: since it is possible to reduce inside an & then the types of the components may decrease (see the subject reduction theorem 4.6) and cannot be fixed (the index does not change with the reduction thus even if types are equal modulo the ordering in overloaded types, terms are *not* equal modulo index reordering).

A first non trivial result for this system is given by the following theorem.

Theorem 3.1. If $C \vdash a\colon T$ then $C \vdash T$ type

Proof. The proof is an easy induction on the depth of the proof of $C \vdash a\colon T$ by performing a case analysis on the last applied rule. The cases for [$\forall$Elim] and [$\{\}$Elim] are solved by using the lemma 4.3. □

As the careful reader will have noticed, we do not use subsumption in the type checking; since the selection of a branch is done according to the type of the argument we want, to avoid ambiguities, that every well typed term has a unique type. This is stated by the following theorem:

Theorem 3.2. If $C \vdash a\colon T_1$ and $C \vdash a\colon T_2$ then and $T_1 = T_2$

Proof. An easy induction on sum of the depths of the derivation of $C \vdash a\colon T_1$ and $C \vdash a\colon T_2$, by performing a case analysis on the structure of a □

By theorem 2.15, we can associate to every provable judgment a canonical derivation.

Theorem 3.3. Let Π_1 and Π_2 be two derivations for the same judgment $C \vdash a\colon T$. Let $(\Pi_i)^*$ $(i = 1, 2)$ denote the derivation Π_i in which every (sub-)derivation of a subtyping judgment has been replaced by its canonical form. Then $\Pi_1 \equiv \Pi_2$.

Proof. By induction on the structure of a (which univocally determines the typing rule to apply). □

By combining the result of this two theorems we obtain that every well typed term has a canonical derivation for its type.

Thus one would expect that it is possible to define a type-checking algorithm for the raw terms. This is the case, indeed: if in the system above you replace every subtyping judgment $C \vdash S{\leq}T$ by $C \vdash_{\mathcal{A}} S{\leq}T$ you have a type-checking algorithm that can be easily proved sound and complete w.r.t. the original system.

4. Reduction

In this section we give the equational theory of the terms of $F^{\&}_{\leq}$. We present it under the form of reduction rules. We assume we work modulo α-conversion for term variables; note that no clash is possible for type variables because of the definition of tcs. The reduction rules are context dependent.

Notions of reduction

(β) $C \vdash (\lambda x^T.a)(b) \rhd a[x^T := b]$

$(\beta_\forall)$ $C \vdash (\Lambda X{\leq}T.a)(T') \rhd a[X := T']$

$(\beta_{\{\}})$ If $A, A_1 \ldots, A_n$ are closed then

$$C \vdash (a\&^{[A_1.T_1\|\ldots\|A_n.T_n]}b)[A] \rhd \begin{cases} b(A) & \text{if } A_n = min_{1\leq i\leq n}\{A_i | C \vdash A{\leq}A_i\} \\ a[A] & \text{otherwise} \end{cases}$$

Note that the selection of the branch is made on the index. Therefore while overloaded types are equal module reordering of their components, in indexes the order is meaningful since to a different ordering may correspond a different selection.

Besides these rules there are the usual rules for the context; among these the only one that deserves a note is the rule for Λ, for it changes the tcs of the reduction:

$$\frac{C \cup \{X{\leq}T\} \vdash a \rhd a'}{C \vdash (\Lambda X{\leq}T.a) \rhd (\Lambda X{\leq}T.a')}$$

For what it concerns the rules note that in $\beta_{\{\}}$ we require that the types involved in the selection of a branch are closed. In this way we always select the most precise branch (i.e. the one with the smallest possible bound). This correspond in object-oriented programming to the implementation of the *dynamic binding* (for a wider discussion on this topic see [CGL93].)

4.1. Subject Reduction

In this section we prove that the type-checking system of $F^{\&}_{\leq}$ is well behaved w.r.t. the reduction rules. More precisely we prove that every (well-typed) term rewrites to another (well-typed) term, whose type is smaller than or equal to the type of the former. The proof of subject reduction is very technical and complex. The crux of the problem is to prove that the property of $\cap$-closure is conserved under reductions, more precisely under (feasible) substitutions. For this reason we suggest the reader to skip at first reading the proofs of the three lemmas that follows.

We need first some notation:

Notation 4.1. Let $C \cup \{X \leq T\}$ be a tcs. Define $(C \cup \{X \leq T\})[Y\colon= S]$ as $(C[Y\colon= S] \cup \{X \leq T[Y\colon= S]\})$ and $\emptyset[X\colon= S]$ as $\emptyset$. Let $C \vdash \Delta$ be a type judgment. Then $C \vdash \Delta[X\colon= S]$ is defined as $C \vdash T[X\colon= S]$ type if $\Delta \equiv T$ type, as $C \vdash T_1[X\colon= S] \leq T_2[X\colon= S]$ if $\Delta \equiv T_1 \leq T_2$.

The proof of subject reduction requires an assumption and three technical lemmas:

Assumption 4.2. Recall that the proof of $C \vdash \{A_i\}_{i=1..n}\cap$-closed is indeed an appropriate set of proofs with final judgments of the form $C \vdash A_h \leq A_k$ proving the meet closure of $\{A_i\}_{i=1..n}$. In particular we suppose that this set contains *at least* one proof of $C \vdash A_i \leq A_j$ for every i, j in $[1..n]$ for which such a proof exists.

Lemma 4.3. (main lemma) If $C \cup \{X \leq S\} \vdash \Delta$ is a provable type judgment, $X \notin FV(S')$ and $C[X\colon= S'] \vdash S' \leq S$ is also provable, then $C[X\colon= S'] \vdash \Delta[X\colon= S']$ is provable, too.

Before proving the lemma, we want clarify a point: indeed the reader may wonder why in this lemma, as well as in lemma 4.5, we used the tcs $C[X\colon= S']$ rather than C. Actually if you replace $C[X\colon= S']$ by C the theorem can no longer be proved, since at some points it is not possible to use the induction hypothesis (more precisely when you introduce a new variable in the tcs). The intuitive reason is that even if $C \cup \{X \leq S\}$ and $C[X\colon= S']$ are well formed tcs's this does not imply the good formation of C. For example take $S' \equiv S \equiv B$ and $C \equiv \{Y \leq X\}$: C is not well formed but $C[X\colon= S'] \equiv Y \leq B$ and $C \cup \{X \leq S\} \equiv \{Y \leq X\} \cup \{X \leq B\}$ are well formed.[10] We can now prove the lemma.

Lemma 4.4. (term substitution) If $C \vdash b\colon T' \leq T$ and $C \vdash a\colon S$ then $C \vdash a[x^T\colon= b] : S' \leq S$.

Lemma 4.5. (type substitution) If $C \cup \{X \leq S\} \vdash a\colon T$, $C[X\colon= S'] \vdash S' \leq S$ and $X \notin FV(S')$ then $C[X\colon= S'] \vdash a[X\colon= S']\colon T' \leq T[X\colon= S']$

Lemmas 4.3 and 4.5 constituted the hard part of the proof. It is then rather straightforward to prove the theorem of subject reduction by using the same technique of [CGL92].

Theorem 4.6. (subject reduction) If $C \vdash a\colon T$ and $C \vdash a \rhd b$ then $C \vdash b\colon T'$ and $C \vdash T' \leq T$

Proof. The proof is by induction on the depth of the proof of $C \vdash a \rhd b$. Instead of presenting the proof for the base case (the rules (β), $(\beta_\forall)$ and $(\beta_{\{\}})$) and for the inductive case (the context rules), we think that a case analysis on the structure of a is more intelligible. However we change only the order of presentation of the proof not the proof itself:

[10] The order in tcs is not important

$a \equiv x^T$ trivial.

$a \equiv \varepsilon$ trivial

$a \equiv \mathsf{Top}$ trivial

$a \equiv \lambda x^{T_1}.a'$, $C \vdash a' \rhd b'$ and $b \equiv \lambda x^{T_1}.b'$. This case is solved by a straightforward use of the induction hypothesis.

$a \equiv \Lambda X{\leq}T_1.a'$ $C \cup \{X{\leq}T_1\} \vdash a' \rhd b'$ and $b \equiv \Lambda X{\leq}T_1.b'$. This case is solved by a straightforward use of the induction hypothesis.

$a \equiv (a_1 \&^I a_2)$ just note that whichever reduction is performed the reductum is well-typed and the type does not change

$a \equiv a_1(a_2)$ where $C \vdash a_1 : W$, $C \vdash a_2 : S' \leq S$ and $\mathcal{B}(W)_C = S \rightarrow T$. Then there are three possible subcases:

1. $a_1 \equiv \lambda x^S.a_3$ and $b \equiv a_3[x^S := a_2]$. this case follows from lemma 4.4
2. $C \vdash a_1 \rhd a_1'$. Then by induction hypothesis we have $C \vdash a_1' : T'' \leq W$. By proposition 2.3 $C \vdash \mathcal{B}(T'')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(T'')_C$ is a not a type variable then it is of the form $S'' \rightarrow T'$ with $C \vdash S' \leq S \leq S''$ and $C \vdash T' \leq T$. Thus b is well-typed and with type $T' \leq T$.
3. $C \vdash a_2 \rhd a_2'$. Then by induction hypothesis we have $C \vdash a_2' : S'' \leq S' \leq S$. Thus $C \vdash b : T$

$a \equiv a'(S)$ where $C \vdash a' : W$, $C \vdash S \leq S'$, $\mathcal{B}(W)_C = \forall(X{\leq}S')S''$ and $T \equiv S''[X := S]$. Since $\mathcal{B}(W)_C = \forall(X{\leq}S')S''$, then

$$C \vdash \forall(X{\leq}S')S'' \text{ type}$$

this holds only if

$$C \cup \{X{\leq}S'\} \vdash S'' \text{ type}$$

from which we deduce that $X \notin dom(C)$. From this and from $C \vdash S \leq S'$ we deduce that $X \notin FV(S)$.
Now there are two possible subcases:

1. $a' \equiv \Lambda X{\leq}S'.a''$ and $b \equiv a''[X := S]$. But since $C \vdash S \leq S'$ and $X \notin FV(S)$ we can apply lemma 4.5. The result follows from $X \notin dom(C)$.
2. $C \vdash a' \rhd b'$. thus by induction hypothesis and by proposition 2.3 we obtain $C \vdash b' : T'' \leq W$ and $C \vdash \mathcal{B}(T'')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(T'')_C$ is not a type variable then it is of the form $\forall(X \leq U')U''$ with $C \vdash S \leq S' \leq U'$ and $C \cup \{X \leq S'\} \vdash U'' \leq S''$. Thus b is well typed and $C \vdash b : U''[X := S]$. The result follows from the main lemma applied to $C \vdash U'' \leq S''$ and the fact that $X \notin dom(C)$

$a \equiv a'[A]$ where $C \vdash a' : W$ and $\mathcal{B}(W)_C = \forall X\{A_i.T_i\}_{i \in I}$. As in the case before it is possible to prove that $X \notin dom(C)$ and that $X \notin FV(A)$. Let $A_h = \min_{i \in I}\{A_i \mid C \vdash A \leq A_i\}$. Then $T \equiv T_h[X := A]$. Again we have two subcases:

1. $a' \equiv (a_1 \&^{[A_1.T_1 \| \cdots \| A_n.T_n]} a_2)$ and $A, A_1, \ldots, A_n$ are closed and a $\beta_{\{\}}$-reduction is performed. Then either $b \equiv a_1[A]$ (case $A_h \neq A_n$) or $b \equiv a_2(A)$ (case $A_h = A_n$). In both cases, by $[\{\}\textsc{Elim}]$ or by $[\forall\textsc{Elim}]$ according to the case, it is easy to prove that the terms have type $T' \leq T_h[X := A]$: just use the induction hypothesis and then apply the main lemma.
2. $C \vdash a' \rhd a''$. Then by induction hypothesis $C \vdash a'' : W' \leq W$ and by proposition 2.3 $C \vdash \mathcal{B}(W')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(W')_C$ is not a type variable $\forall X\{A_j'.T_j'\}_{j \in J}$. Thus by the subtyping rule ($\{\}$) there exists $\tilde{h} \in J$ such that $C \vdash A \leq A_h \leq A_{\tilde{h}}'$. Therefore the set $\{A_j' \mid C \vdash A \leq A_j', j \in J\}$ is not empty,

and by the meet-closure of $\{A'_j\}_{j\in J}$ it has also a minimum. Call this minimum A'_k. Then $C \vdash b : T'_k[X:=A]$. Since $S \equiv T_h[X:=A]$ we have to prove that

$$C \vdash T'_k[X:=A] \leq T_h[X:=A]$$

Take again the previous $\tilde{h}$; by the rule $(\{\})$ we have

$$C \vdash \forall(X \leq A'_{\tilde{h}})T'_{\tilde{h}} \leq \forall(X \leq A_h)T_h \tag{1}$$

By the definition of A_h:

$$C \vdash A \leq A_h \tag{2}$$

From (1):

$$C \vdash A_h \leq A'_{\tilde{h}}$$

From (trans):

$$C \vdash A \leq A'_{\tilde{h}} \tag{3}$$

From (1):

$$C \cup \{X \leq A_h\} \vdash T'_{\tilde{h}} \leq T_h \tag{4}$$

From the definition of A'_k and from (3) we obtain

$$C \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule $(\{\}_{type})$ applied to $\forall X\{A'_j.T'_j\}_{j\in J}$ it follows

$$C \cup \{X \leq A_h\} \vdash T'_k \leq T'_{\tilde{h}} \tag{5}$$

By (2) and by the choice of k we respectively have that $C \vdash A \leq A_h$ and $C \vdash A \leq A'_k$; thus we can apply the main lemma to (4) and (5) to obtain:

$$C[X:=A] \vdash T'_{\tilde{h}}[X:=A] \leq T_h[X:=A]$$
$$C[X:=A] \vdash T'_k[X:=A] \leq T'_{\tilde{h}}[X:=A]$$

But $X \notin dom(C)$, thus the judgments above get

$$C \vdash T'_{\tilde{h}}[X:=A] \leq T_h[X:=A]$$
$$C \vdash T'_k[X:=A] \leq T'_{\tilde{h}}[X:=A]$$

Finally by (trans) we obtain the result:

$$C \vdash T'_k[X:=A] \leq T_h[X:=A]$$

□

4.2. Church-Rosser

In section 2.3 we proved the syntactic coherence of the *proof system* of $F^{\&}_{\leq}$. In this section we prove the syntactic coherence of the *reduction system* of $F^{\&}_{\leq}$.

In the reductions that follow we omit, without loss of generality, all the tcs[11]. To prove the Church-Rosser property (**CR**) we use a method of Hindley [Hin64] and Rosen [Ros73]:

Lemma 4.7. (Hindley-Rosen) Let R_1 and R_2 be two notions of reduction. If R_1 and R_2 are **CR** and $\triangleright^*_{R_1}$ commutes with $\triangleright^*_{R_2}$ then $R_1 \cup R_2$ is **CR**.

[11] The only place where this omission really matters is in the lemma 4.10 whose complete statement should be *If* $C \cup \{X \leq S\} \vdash a \triangleright_{\beta_{\{\}}} a'$ *then* $C \vdash a[X:=T] \triangleright^*_{\beta_{\{\}}} a'[X:=T]$.

Set now $R_1 \equiv \beta_{\{\}}$ and $R_2 \equiv \beta \cup \beta_\forall$; if we prove that these notions of reduction satisfy the hypotheses of the lemma above then we proved $\mathbf{CR}$. It is easy to prove that $\beta \cup \beta_\forall$ is $\mathbf{CR}$: indeed in [Ghe90] it is proved that $\beta \cup \beta_\forall$ is terminating; by a simple check of the conflicts it is possible to prove that it is also locally confluent; since it has no critical pair then by the Knuth-Bendix lemma ([KB70]) it is locally confluent; finally by applying the Newman's Lemma ([New42]) we obtain $\mathbf{CR}$.

Lemma 4.8. $\beta_{\{\}}$ is $\mathbf{CR}$.

Proof. By lemma 3.2.2 of [Bar84] it suffices to prove that the reflexive closure of $\triangleright_{\beta_{\{\}}}$ (denoted by $\triangleright^{=}_{\beta_{\{\}}}$) satisfies the diamond property. Thus by induction on $a \triangleright^{=}_{\beta_{\{\}}} a_1$ we show that for all $a \triangleright^{=}_{\beta_{\{\}}} a_2$ there exists a common $\triangleright^{=}_{\beta_{\{\}}}$ reduct a_3 of a_1 and a_2. We can assume that $a_1 \not\equiv a$, $a_2 \not\equiv a$ and $a_1 \not\equiv a_2$, otherwise the proof is trivial. Let examine all the possible cases:

1. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} b_1[A]$. If $a_2 \equiv (b_1\&b'_2)[A]$ then $a_3 \equiv a_1$; else $a_2 \equiv (b'_1\&b_2)[A]$ then $a_3 \equiv b'_1[A]$.
2. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} b_2(A)$. If $a_2 \equiv (b'_1\&b_2)[A]$ then $a_3 \equiv a_1$; else $a_2 \equiv (b_1\&b'_2)[A]$ then $a_3 \equiv b'_2(A)$.
3. $b_1(b_2) \triangleright^{=}_{\beta_{\{\}}} b'_1(b_2)$. If $a_2 \equiv b_1(b'_2)$ then $a_3 \equiv b'_1(b'_2)$; else $a_2 \equiv b''_1(b_2)$: then by induction hypothesis there exists b_3 common $\triangleright^{=}_{\beta_{\{\}}}$ reduct of b'_1 and b''_1; thus $a_3 \equiv b_3(b_2)$
4. $b_1(b_2) \triangleright^{=}_{\beta_{\{\}}} b_1(b'_2)$ as the case before.
5. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} (b'_1\&b_2)$ as the case before.
6. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} (b_1\&b'_2)$ as the case before.
7. $\lambda x^T.a \triangleright^{=}_{\beta_{\{\}}} \lambda x^T.a'$. Then $a_2 \equiv \lambda x.a''$ and by induction hypothesis there exists b_3 common $\triangleright^{=}_{\beta_{\{\}}}$ reduct of a' and a''. Thus $a_3 \equiv \lambda x^T.b_3$.
8. $\Lambda X{\leq}T.a \triangleright^{=}_{\beta_{\{\}}} \Lambda X{\leq}T.a'$. as the case before (apart from the change of tcs in the induction hypothesis.
9. $a(T) \triangleright^{=}_{\beta_{\{\}}} a'(T)$ as the case before.
10. $a[A] \triangleright^{=}_{\beta_{\{\}}} a'[A]$ as the case before.

□

To prove that the two notions of reduction commute we need three technical lemmas:

Lemma 4.9. If $a \triangleright_{\beta_{\{\}}} a'$ then $a[x\colon= b] \triangleright^{*}_{\beta_{\{\}}} a'[x\colon= b]$

Lemma 4.10. If $a \triangleright_{\beta_{\{\}}} a'$ then $a[x\colon= T] \triangleright^{*}_{\beta_{\{\}}} a'[x\colon= T]$

Lemma 4.11. If $b \triangleright_{\beta_{\{\}}} b'$ then $a[x\colon= b] \triangleright^{*}_{\beta_{\{\}}} a[x\colon= b']$

These lemmas can be proved by a straightforward use of induction (on $a \triangleright_{\beta_{\{\}}} a'$ for the first two and on a for the third). Just for the proof of the second, note that in $\beta_{\{\}}$, the atomic types $A, A_1, \ldots, A_n$ are required to be closed. We can now prove that the two notions of reduction commute.

Lemma 4.12. If $a \triangleright_{\beta\cup\beta_\forall} a_1$ and $a \triangleright_{\beta_{\{\}}} a_2$ then there exists a_3 such that $a_1 \triangleright^{*}_{\beta_{\{\}}}$

a_3 and $a_2 \rhd_{\beta\cup\beta_\forall} a_3$. Pictorially:

$$\begin{tikzcd}
a \arrow[r, "\rhd_{\beta\cup\beta_\forall}"] \arrow[d, "\rhd_{\beta_{\{\}}}"] & a_1 \arrow[d, dashed, "\rhd^*_{\beta_{\{\}}}"] \\
a_2 \arrow[r, dashed, "\rhd_{\beta\cup\beta_\forall}"'] & a_3
\end{tikzcd}$$

(Where full arrows are used for the hypotheses and dashed arrows for the theses.)

Proof. A proof of this lemma can be given by a simple diagram chase. Let $\mathcal{C}[\ ]$ be a context (in the sense of [Bar84])[12]. Then we have the following cases:

$$\begin{tikzcd}
 & & \mathcal{C}[(\lambda x.a)b] \arrow[dll, "\rhd_{\beta_{\{\}}}"'] \arrow[d, "\rhd_\beta"] \arrow[drr, "\rhd_{\beta_{\{\}}}"] & & \\
\mathcal{C}[(\lambda x.a')b] \arrow[dr, "\rhd_\beta"'] & & \mathcal{C}[a[x:=b]] \arrow[dl, "\rhd^*_{\beta_{\{\}}}"] \arrow[dr, "\rhd^*_{\beta_{\{\}}}"'] & & \mathcal{C}[(\lambda x.a)b'] \arrow[dl, "\rhd_\beta"] \\
 & \mathcal{C}[a'[x:=b]] \arrow[u, phantom, "\text{lemma 4.9}"] & & \mathcal{C}[a[x:=b']] \arrow[u, phantom, "\text{lemma 4.11}"] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathcal{C}[(\Lambda X{\leq}S.a)(T)] \arrow[dl, "\rhd_{\beta_{\{\}}}"'] \arrow[dr, "\rhd_{\beta_\forall}"] & \\
\mathcal{C}[(\Lambda X{\leq}S.a')(T)] \arrow[dr, "\rhd_{\beta_\forall}"'] & \text{lemma 4.10} & \mathcal{C}[a[X:=T]] \arrow[dl, "\rhd^*_{\beta_{\{\}}}"] \\
 & \mathcal{C}[a'[X:=T]] &
\end{tikzcd}$$

[12] Avoid confusion between a context, denoted by $\mathcal{C}[\ ]$ and a type constraint system, denoted by C.

$$\begin{tikzcd}
 & & \mathcal{C}[\,(a_1\&^{I} a_2)[A]\,] \arrow[dll, "\rhd_{\beta\cup\beta_\forall}"'] \arrow[dl, "\rhd_{\beta_{\{\}}}"] \arrow[dr, "\rhd_{\beta_{\{\}}}"'] \arrow[drr, "\rhd_{\beta\cup\beta_\forall}"] & & \\
\mathcal{C}[\,(a_1\&^{I} a_2')[A]\,] \arrow[r, "\rhd_{\beta_{\{\}}}"'] \arrow[dr, "\rhd_{\beta_{\{\}}}"'] & \mathcal{C}[\,a_1[A]\,] \arrow[drr, "\rhd_{\beta\cup\beta_\forall}"] & & \mathcal{C}[\,a_2[A]\,] \arrow[dll, "\rhd_{\beta\cup\beta_\forall}"'] & \mathcal{C}[\,(a_1'\&^{I} a_2)[A]\,] \arrow[l, "\rhd_{\beta_{\{\}}}"] \arrow[dl, "\rhd_{\beta_{\{\}}}"] \\
 & \mathcal{C}[\,a_2'[A]\,] & & \mathcal{C}[\,a_1'[A]\,] &
\end{tikzcd}$$

□

Corollary 4.13. $\rhd^*_{\beta_{\{\}}}$ commutes with $\rhd^*_{\beta\cup\beta_\forall}$

Proof. By lemma 3.3.6 in [Bar84]. □

In conclusion all the hypotheses of lemma 4.7 are satisfied, and we can conclude that $F_{\leq}^{\&}$ is $\mathbf{CR}$.

5. A decidable subcalculus

It is well known that the type system of $F_{\leq}$ is undecidable and that this problem comes from the subtyping system [Pie93]. Of course undecidability is inherited by $F_{\leq}^{\&}$. The crux of the problem is the (∀) rule, which is responsible for the loss of many other syntactical properties (for a review see [CP94]).

In [CP94] we have defined a restriction of the subtyping system of $F_{\leq}$, called $F_{\leq}^{\top}$, which enjoys many of the syntactical properties that $F_{\leq}$ lacks, foremost decidability of subtyping. Furthermore nothing seems to be lost from a pragmatical viewpoint since all the programs that in our ken have been written for $F_{\leq}$ are well typed also in our restriction.

The definition of this restriction is very easy: just substitute the incriminated (∀) rule by the following one

$$(\forall\text{-new}) \qquad \frac{C \vdash T_1' \leq T_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash T_2 \leq T_2'}{C \vdash \forall(X \leq T_1)T_2 \leq \forall(X \leq T_1')T_2'} \qquad X \notin dom(C)$$

in which the right-hand premise requires that the bodies be (covariantly) related under *no* assumption about the bound variable. This essentially amounts to considering the subtyping relation relative to an unchanging context, since the type variables added to the context always have trivial bounds; the only type variables with interesting bounds will be those already present in the environment at the point where a subtyping check is required. (These are introduced, as usual, by the quantifier introduction rule.)

This simple change makes the subtyping relation decidable; thus one may wonder if by the same change in $F_{\leq}^{\&}$ one obtains decidability. This is indeed the case, as we show in this section. As we already did when dealing with the transitivity elimination, we concentrate our attention on the subtyping rules, forgetting the type good formation; thus once more we suppose that all the types that appear below are well formed. By analogy with [CP94] we call it $F_{\leq}^{\&\top}$.

5.1. Subtyping algorithm

In order to verify the decidability of the subtyping relation of $F_{\leq}^{\&\top}$, one has first to define a subtyping algorithm which is sound and complete with respect to the type system. This is obtained by replacing the (Alg∀) rule in the algorithm of section 2.3 by the new rule:

$$(\text{Alg}\forall\text{-new}) \qquad \frac{C \vdash_{\mathcal{A}} T'_1 {\leq} T_1 \qquad C \cup \{X {\leq} \mathsf{Top}\} \vdash_{\mathcal{A}} T_2 {\leq} T'_2}{C \vdash_{\mathcal{A}} \forall(X {\leq} T_1)T_2 \leq \forall(X {\leq} T'_1)T'_2} \qquad X \notin dom(C)$$

Note that we do not change the rule (Alg$\{\}$); one might expect that also the bound in $C \cup \{X {\leq}\, A'_i\} \vdash T_j {\leq}\, T'_i$ should be changed form A'_i to Top. This is not necessary to obtain decidability: indeed the bounds used in the overloaded quantification are far less general than those used in standard quantification, since the former can range only on constant types while the latter can range on all types (whence the undecidability). Thus we leave (Alg$\{\}$) as it is; in section 6 we will show how to use its full expressiveness, by defining some methods that would not be typed if (Alg$\{\}$) used the bound Top in comparing the types of different branches (see the definition of $Erase$). This however has a minor drawback, since we are not allowed to use the simple technique of [CP94] to prove transitivity elimination and thus the completeness of the algorithm. We are obliged to use the technique of [CG92] and prove again all the theorems of sections 2.2 and 2.3 from scratch.

We will not rewrite them here since actually very few modifications to the proofs in section 2.2 suffice to do the work. The main modification is in the rewriting system of section 2.2.1 where you have to substitute the rules

$$\begin{array}{lll}(\forall') & (\forall(X {\leq}\, c)d)\,(\forall(X {\leq}\, c')d') & \rightsquigarrow\ \ \forall(X {\leq}\, c'\, c)(d\, d'[X_T\!:= c\, X_S]) \\ (\forall'') & (\forall(X {\leq}\, c)d)\,((\forall(X {\leq}\, c')d')\, e) & \rightsquigarrow\ \ (\forall(X {\leq}\, c'\, c)(d\, d'[X_T\!:= c\, X_S]))\, e\end{array}$$

by the following ones

$$\begin{array}{lll}(\forall') & (\forall(X {\leq}\, c)d)\,(\forall(X {\leq}\, c')d') & \rightsquigarrow\ \ \forall(X {\leq}\, c'\, c)(d\, d') \\ (\forall'') & (\forall(X {\leq}\, c)d)\,((\forall(X {\leq}\, c')d')\, e) & \rightsquigarrow\ \ (\forall(X {\leq}\, c'\, c)(d\, d'))\, e\end{array}$$

Of course now $(\forall(X {\leq}\, c)d)$ codifies the new (∀) rule:

$$(\forall\text{-new}) \qquad \frac{C \vdash c_1\!: T'_1 {\leq} T_1 \qquad C \cup \{X {\leq} \mathsf{Top}\} \vdash c_2\!: T_2 {\leq} T'_2}{C \vdash \forall(X {\leq} c_1)c_2\!: \forall(X {\leq} T_1)T_2 {\leq} \forall(X {\leq} T'_1)T'_2}$$

The reader can now move across the proofs of section 2.2 and check the obvious modifications; the proofs of section 2.3 are essentially unchanged.

5.1.1. Termination

We now prove that this algorithm terminates.

Definition 5.1. *Let C be a tcs and T a type such that $FV(T) \subseteq dom(C)$ then define*

$$\mathcal{L}(T)_C = \begin{cases} 0 & \textit{if } T \textit{ is not a type variable} \\ \mathcal{L}(C(T))_C + 1 & \textit{otherwise} \end{cases}$$

($\mathcal{L}$ stays for “length”) □

Notation 5.2. Let C be a tcs; we denote by $\widehat{C}$ a type variable $Y \in dom(C)$ such that

$$\mathcal{L}(Y)_C = \max_{X \in dom(C)} \{\mathcal{L}(X)_C\}$$

If there is more than one such a variable then choose any of them (e.g. use the textual order)

We can now define a weight $\mathcal{T}$ for a type T with respect to a tcs C (such that T is well formed in C):

$$\begin{aligned}
\mathcal{T}(B)_C &\stackrel{def}{=} 1\\
\mathcal{T}(\mathsf{Top})_C &\stackrel{def}{=} 1\\
\mathcal{T}(X)_C &\stackrel{def}{=} \mathcal{T}(C(X))_C + 1\\
\mathcal{T}(S_1 \to S_2)_C &\stackrel{def}{=} \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C\\
\mathcal{T}(\forall(X{\leq}S_1)S_2)_C &\stackrel{def}{=} \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}}\\
\mathcal{T}(\forall X\{A_i.T_i\}_{i\in I})_C &\stackrel{def}{=} \max_{i\in I}\{\mathcal{T}(A_i)_C, \mathcal{T}(T_i)_{C\cup\{X\leq\widehat{C}\}}\} + 1
\end{aligned}$$

Lemma 5.3. For each type T well formed in a tcs C, the weight $\mathcal{T}(T)_C$ is finite and positive.

Proof. First, it is obvious that the weight $\mathcal{T}(T)_C$ is always positive. Now to prove that it is also finite, we give a well founded rank for $\mathcal{T}(T)_C$ (i.e. we define a weight for the definition of the weight) and we show that it decreases at each stage in the definition of $\mathcal{T}$. To define the rank of $\mathcal{T}(T)_C$ consider all the variables that appear in T and C (no matter whether they appear free or bounded, only in a quantifier or in a bound). Since T is well formed in C, every variable is associated to a unique bound (either in C or in T) apart those appearing in T as a quantification of an overloaded type; to these variables associate as bound $\widehat{C}$. Furthermore it is also possible to totally order these variables in a way that if X_i is defined in the bound of X_j then X_i precedes X_j (C is a tcs so it is $C \cup \{X \leq \widehat{C}\}$ —with $X \notin dom(C)$—, T is well formed in C, thus loops are not possible). If there is more than one order satisfying this condition then choose one arbitrarily. Define the *depth* of each variable as the number of variables that precede it in this order. Then the rank of $\mathcal{T}(T)_C$ is the lexicographical size of the pair (D, L), where D is the maximum depth of any of the variables that appear in T, and L is the textual length of T. This rank is well founded (the least element is $(0, 1)$). Take now the definition of $\mathcal{T}$: it easy to see that for the subproblems on the right-hand side of $\mathcal{T}(S_1 \to S_2)_C$, $\mathcal{T}(\forall(X{\leq}S_1)S_2)_C$ and $\mathcal{T}(\forall X\{A_i.T_i\}_{i\in I})_C$, the component D either is the same or it decreases, while the L component always strictly decreases; for the case $\mathcal{T}(X)_C$, the component D strictly decreases. □

The weight of the types is extended to a weight for type judgments in the obvious way:

$$\mathcal{J}(C \vdash S_1 \leq S_2) = \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C.$$

Now we can show the termination of the algorithm.

Lemma 5.4. Given a tcs C and a type variable X, for all types T_1, T_2 such that $FV(T_i) \subseteq dom(C)$ $(i = 1, 2)$ $\qquad \mathcal{T}(T_1)_{C\cup\{X\leq\mathrm{Top}\}} \leq \mathcal{T}(T_1)_{C\cup\{X\leq T_2\}}$.

Proof. A simple induction on the definition of $\mathcal{T}(T_1)$ (note that one of the consequences of lemma 5.3 is that it is possible to use induction on $\mathcal{T}$). □

Lemma 5.5. Given a tcs C, a type variable $X \notin dom(C)$, two atomic types A and A' such that $\mathcal{B}(A)_C$ and $\mathcal{B}(A')_C$ are constant types, if $\mathcal{L}(A)_C \leq \mathcal{L}(A')_C$ then:

(a) $\mathcal{T}(A)_C \leq \mathcal{T}(A')_C$
(b) for all T such that $FV(T) \subseteq dom(C) \cup \{X\}$, $\mathcal{T}(T)_{C\cup\{X\leq A\}} \leq \mathcal{T}(T)_{C\cup\{X\leq A'\}}$

Proof. there are three possible cases:

1. Both A and A' are constant types: (a) is trivial; (b) follows by a straightforward induction on $\mathcal{T}(T)_{C\cup\{X\leq A\}} + \mathcal{T}(T)_{C\cup\{X\leq A'\}}$, performing a case analysis on T.
2. A is a constant type and A' is a type variable: as the previous case
3. Both A and A' are type variables: we prove (a) by induction on $\mathcal{L}(A)_C + \mathcal{L}(A')_C$. The base case is when $\mathcal{L}(A)_C = \mathcal{L}(A')_C = 1$. In that case it is easy to check that $\mathcal{T}(A)_C = \mathcal{T}(A')_C = 2$. When that sum is strictly larger than 2 then by definition of $\mathcal{T}$
$$\mathcal{T}(A)_C \leq \mathcal{T}(A')_C \Longleftrightarrow \mathcal{T}(C(A))_C \leq \mathcal{T}(C(A'))_C$$
By definition of $\mathcal{L}$, $\mathcal{L}(A)_C \leq \mathcal{L}(A')_C$ implies $\mathcal{L}(C(A))_C \leq \mathcal{L}(C(A'))_C$; therefore we can apply the induction hypothesis to obtain the result.

   Once more, (b) follows by a straightforward induction on $\mathcal{T}(T)_{C\cup\{X\leq A\}}+\mathcal{T}(T)_{C\cup\{X\leq A'\}}$, performing a case analysis on T: use the case (a) of this lemma when $T \equiv X$.

□

Theorem 5.6. At every step of the subtyping algorithm, the weight of each of the premises is strictly smaller than the weight of the conclusion.

Proof. The verification is easy in most cases. The only non-trivial cases are (Alg∀) and (Alg$\{\}$). The first case is proved by the following inequalities:

$$\begin{aligned}
&\mathcal{J}(C \cup \{X{\leq}\mathsf{Top}\} \vdash S_2{\leq}T_2) \quad = \\
&\quad = \quad \mathcal{T}(S_2)_{C\cup\{X\leq\mathsf{Top}\}} + \mathcal{T}(T_2)_{C\cup\{X\leq\mathsf{Top}\}} \\
&\quad \leq \quad \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} && \text{by lemma 5.4} \\
&\quad < \quad \mathcal{T}(S_1)_C + \mathcal{T}(T_1)_C + \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} \\
&\quad = \quad \mathcal{T}(\forall(X{\leq}S_1)S_2)_C + \mathcal{T}(\forall(X{\leq}T_1)T_2)_C \\
&\quad = \quad \mathcal{J}(C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2)
\end{aligned}$$

For (Alg$\{\}$) the proof is given by these inequalities:

$$\begin{aligned}
&\mathcal{J}(C \cup \{X{\leq}A'_i\} \vdash T_j{\leq}T'_i) \quad = \\
&\quad = \quad \mathcal{T}(T_j)_{C\cup\{X\leq A'_i\}} + \mathcal{T}(T'_i)_{C\cup\{X\leq A'_i\}} \\
&\quad \leq \quad \mathcal{T}(T_j)_{C\cup\{X\leq\widehat{C}\}} + \mathcal{T}(T'_i)_{C\cup\{X\leq\widehat{C}\}} && \text{by lemma 5.5} \\
&\quad \leq \quad \max_{j\in J}\{\mathcal{T}(A_j)_C, \mathcal{T}(T_j)_{C\cup\{X\leq\widehat{C}\}}\} + \max_{i\in I}\{\mathcal{T}(A'_i)_C, \mathcal{T}(T'_i)_{C\cup\{X\leq\widehat{C}\}}\} \\
&\quad < \quad \mathcal{J}(C \vdash \forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A'_i.T'_i\}_{i\in I})
\end{aligned}$$

□

Corollary 5.7. The algorithm terminates

5.2. Terms and reduction

Up to now we dealt with the types of $F^{\&\top}_{\leq}$. To end with it, it still remains to describe its terms and reduction rules. The task is easy for the raw terms, which are exactly the same as those for $F^{\&}_{\leq}$. More difficult is instead the case for reduction rules and typing rules; we have two different choices: either we use the same typing rules as for $F^{\&}_{\leq}$ and we do not allow reductions involving free type variables, or we add to these rules the rule of subsumption and we leave the reduction unchanged. As in $F^{\top}_{\leq}$, in the first case we are not able to prove the subject-reduction property (but we gain the decidability of the typing relation), while in the second, as recently remarked by Giorgio Ghelli, the minimal typing property does not hold and the decidability of type checking is an open problem (see also [CP95]).

Note that in both cases there will be less well-typed terms than in $F^{\&}_{\leq}$, since the subtyping relation of $F^{\&\top}_{\leq}$ (with or without subsumption) is strictly contained in that of $F^{\&}_{\leq}$: therefore there will be less well-formed types (some types well-formed in $F^{\&}_{\leq}$ may not satisfy the covariance rule) and some functional applications may no longer result well-typed.

Since subject-reduction is very important from both a theoretical and a practical point of view, we prefer to use the subsumption rule to define the type system of $F^{\&\top}_{\leq}$; in this case the property of subject-reduction still holds. It just requires some work to adapt to the subcalculus the proof of subject reduction of section 4.1: essentially you have to modify the various cases of $a \equiv a'(T)$ to take into account the new subtyping rule ($\forall$-new), and use the subsumption rule in the cases $a \equiv \Lambda X \leq S_1 . a'$ to show that the type *is preserved*; in the case of lemma 4.5 the proof results even simplified. The proof of Church-Rosser for $F^{\&\top}_{\leq}$ is then a consequence of its subject reduction property, and of the fact that $F^{\&}_{\leq}$ is $\mathbf{CR}$: given a term M of $F^{\&\top}_{\leq}$, if $M \rhd^* N_1$ and $M \rhd^* N_2$ then there exists N_3 in $F^{\&}_{\leq}$ such that $N_1 \rhd^* N_3$ and $N_2 \rhd^* N_3$. But, since the notion of reduction is the same in both calculi, the subject reduction theorem for $F^{\&\top}_{\leq}$ guarantees that N_3 is a term of $F^{\&\top}_{\leq}$, too. However decidability of type-checking is an open problem.

Of course, we would choose not to use the subsumption as soon as we proved that $F^{\&\top}_{\leq}$ without subsumption and reductions involving free type variables satisfies the subject-reduction property, since, in this system, the decidability of the subtyping relation implies the decidability of the typing relation. We are comforted in this choice by the fact that the tests we did to check the expressiveness of $F^{\top}_{\leq}$ in [CP93] have been performed by modifying the subtyping algorithm for $F_{\leq}$, i.e. by using a system that does not use the subsumption rule.

Clearly with $F^{\&\top}_{\leq}$ we lose in expressive power since the terms (and the reductions) of $F^{\&\top}_{\leq}$ are strictly contained in those of $F^{\&}_{\leq}$. Thus some terms are lost; but are those terms really interesting? We cannot answer this question as we did in [CP94], where we tried to type-check existing libraries of $F_{\leq}$ programs, by $F^{\top}_{\leq}$: there is no library for $F^{\&}_{\leq}$ since we have just defined it. However we think that for object-oriented programming $F^{\&\top}_{\leq}$ is a good calculus to start from. We will give an idea of this it in the next section where we show how to use this calculus to model object-oriented features; all the examples we will show are typable in $F^{\&\top}_{\leq}$.

6. Object-oriented programming

In this section we want to sketch how the theory developed so far can be used to type object-oriented languages. We mainly consider class-based object-oriented languages since their modeling in the overloading-based model seems less natural than for languages based on generic functions. We will give few hints about generic functions at the end of the section.

From the examples given in the introduction it should be clear that we use the name of a class to type the objects of that class. A message then is (an identifier of) an overloaded function whose branches are the methods associated to that message. The method to be executed is selected according to the type (the class-name) passed as argument which will be the class of the object the message is sent to. Thus the sending of a message $mesg$ to an object a of class A will be modeled by $(mesg[A])a$.

Class-names are *basic types*. We want to associate to each basic type a *representation type*; in particular we want to associate to each class(-name) the type of the internal state of its objects (i.e. the type of the instances variables). The way to formalize it does not concern the subject of this paper (this is done in [Cas95b]); thus here we follow the rudimentary approach of [CGL92]: we suppose that a program (a $F_{\leq}^{\&}$-term) may be preceded by a declaration of *class types*: a *class type* is a basic type, that is associated by its declaration to a unique *representation type*, which is a record type. Two class types are in subtyping relation if this relation has been explicitly declared and it is *feasible*, in the sense that the respective representation types are in subtyping relation too. There is an operation $\_^{classType}$ to transform a record value $r\colon R$ into a class type value $r^{classType}$ of type $classType$, provided that the representation type of $classType$ is R.

We use *italics* to distinguish class types from the usual types, and $\doteq$ to declare a class type and to give it a name; we will use $\equiv$ to associate a name to a value (e.g. to a function). For example we can declare the following class types:

$\mathit{2DPoint} \doteq \langle\!\langle x : \text{Int}; y : \text{Int}\rangle\!\rangle$
$\mathit{3DPoint} \doteq \langle\!\langle x : \text{Int}; y : \text{Int}; z : \text{Int}\rangle\!\rangle$

and then impose on them that $\mathit{3DPoint} \leq \mathit{2DPoint}$ (which is feasible since it respects the ordering of the record types these class types are associated to)[13] . We can define a message $Norm$ working on these class types[14]:

$$\begin{aligned} Norm \equiv \; (\; & \Lambda MyType \leq \mathit{2DPoint}\ .\lambda self^{\,MyType}.\sqrt{self.x^2 + self.y^2} \\ & \&\Lambda MyType \leq \mathit{3DPoint}\ .\lambda self^{\,MyType}.\sqrt{self.x^2 + self.y^2 + self.z^2} \\ &) \end{aligned}$$

Whose type is $\forall MyType.\{\mathit{2DPoint}.MyType \to \text{Real}, \mathit{3DPoint}.MyType \to \text{Real}\}$ We have used the variable $self$ to denote the receiver of the message and, following the notation of [Bru94], the type variable $MyType$ to denote the type of the receiver. Note however that we do not need, as in [Bru94], recursion for these features since they are just parameters of the message.

Let us consider the meaning of the covariance condition of section 2 in this framework. Define the message $Erase$ that set to zero the internal state of an object

$$\begin{aligned} Erase \equiv \; (\; & \Lambda MyType \leq \mathit{2DPoint}.\lambda self^{\,MyType}.\langle self \leftarrow x = 0, y = 0\rangle^{\mathit{2DPoint}} \\ & \&\Lambda MyType \leq \mathit{3DPoint}.\lambda self^{\,MyType}.\langle self \leftarrow x = 0, y = 0, z = 0\rangle^{MyType} \\ &) \end{aligned}$$

it has type: $\forall MyType.\{\mathit{2DPoint}.MyType \to \mathit{2DPoint}, \mathit{3DPoint}.MyType \to MyType\}$ Since $\mathit{3DPoint} \leq \mathit{2DPoint}$ we check that the covariance condition is satisfied:

$$\{MyType \leq \mathit{3DPoint}\} \vdash MyType \to MyType \leq MyType \to \mathit{2DPoint}$$

[13] Note that records are encodable in $F_{\leq}$, and thus in $F_{\leq}^{\&}$ too.

[14] In the examples we will omit ε and the indexes of &

In general if a method has been defined for the message m in the classes B_i for $i \in I$ then its type is of the form $\forall MyType.\{B_i.MyType \to T_i\}_{i \in I}$. If $B_h \leq B_k$ that means that the method defined for m in the class B_h overrides the one defined in B_k. Since $MyType$ is the same in both branches then the covariance condition reduces to prove that

$$\{MyType \leq B_h\} \vdash T_h \leq T_k$$

In other terms the covariance condition requires that an overriding method returns a type smaller than or equal to the type returned by the overridden one. Note that if a method returns a result of type $MyType$ then a method that overrides it has to return $MyType$ too and it is not allowed to return say the class-name of the class in which the method has been defined (since, by inheritance, this could be a type larger then the actual value of $MyType$)[15].

Suppose now that $3DColoredPoint$ is a subclass of $3DPoint$ from which it inherits the method for $Erase$; then the definition of $Erase$ persists unchanged. If an object b of type $3DColoredPoint$ receives the message $Erase$ then the method selected is the one for $3DPoint$; but since $Erase[3DColoredPoint](b) : 3DColoredPoint$ the loss of information is avoided.

In this framework bounds are always basic types (more precisely class-names); thus the $\cap$-closure reduces to impose that if a message has type $\forall X.\{B_i.T_i\}_{i \in I}$ and there exists $h, k \in I$ such that B_h and B_k have a common subclass then there must be a method defined for the message, in the class that is the g.l.b. of B_h and B_k. In other terms, in a class defined by multiple inheritance if two common ancestors can respond to a same message, then the method for that message cannot be inherited but must be explicitly redefined, as in [CGL92].

The way we have written these methods may seem complicated with respect to the simplicity and modularity of object-oriented languages. Indeed the terms above can be regarded as the result of a compilation (or translation) of the following higher-level object-oriented program:

```
class 2DPoint
  state
    x:Int;
    y:Int
  methods
    Norm = sqrt(x^2 + y^2);
    Erase = update(x=0;y=0);
  interface
    Norm: Real;
    Erase: 2DPoint
endclass
```

```
class 3DPoint is 2DPoint
  state
    z:Int
 methods
    Norm = sqrt(x^2+y^2+z^2);
    Erase = update(x=0;y=0;z=0);
  interface
    Norm: Real
    Erase: Mytype
endclass
```

```
class 3DColorPoint is 3DPoint
  state
    color:String
endclass
```

[15] Of course in the previous example it would have been more reasonable that *Erase* returned *MyType* rather than *2DPoint*.

6.1. Extending classes

Inheritance is not the only way to specialize classes: if every time we had to add a method to a class we were obliged to define a new class, the existing objects of the old class could not use the new method. The same is worth also in the case that a method of a class must be redefined: *overriding* would not suffice. For this reason some object-oriented languages such as Objective-C [NeX91], Dylan [App92] and CLOS [DG87] offer the capability to add new methods to existing classes or to redefine the old ones. The extension of the set of the methods of a class affects all its subclasses, in the sense that when a class is extended with a method then that method is available to the objects of every subclass. For example in Dylan the following expression

```
(define-method isOrigin (self <2DPoint>)
    (and (zero? self.x) (zero? self.y)))
```

adds to the class $2DPoint$ a method responding to the message $isOrigin$[16]. If a method for that message has already been defined in the class then it is replaced by the new one.

This can be implemented in our system by adding a new branch to the overloaded function denoted by the message at issue:

$$\begin{array}{ll}\textbf{let}\ \ isOrigin = (& isOrigin \\ & \&\ \Lambda MyType{\leq}2DPoint.\lambda self^{MyType}.(self.x = 0) \wedge (self.y = 0) \\ &) \end{array}$$

That is, the new definition of $isOrigin$ is given by the old definition of $isOrigin$ concatenated with the new method (if $isOrigin$ was undefined we consider it as equal to ε).

Remark that by this construction one does not define a new class but only new methods; in other terms one does not modify the existing types but only (the environment of) the expressions. This is possible in our system since the type of an object is not bound to the procedures that can work on it (and for this reason it differs from abstract data types and, for those who know, the theoretical "objects as records" approach). Of course this flexibility is paid by a minor protection. For that reason for example Dylan has a function `freeze-methods` which prevents certain methods associated to a message to be replaced or removed.

6.2. First class messages

In this model messages are identifiers of overloaded functions. Since overloaded functions have first class citizenship, then also messages are first class. Thus it is possible to model functions that take as parameter a message, or functions whose result is a message. A trivial example is the implementation of a super-like function: suppose that in the definition of a method you want to send a message to *self* but that the method selected must be the one defined for the objects of a certain class C. This can be obtained by the following function:

$$\textbf{let}\ \text{super\_C} = \lambda m^{\forall X\{C.T\}}.m[C]self$$

This function takes a message m accepting objects of class C and sends it to *self* but

[16] This is not the standard Dylan's syntax where record (slot) selection of the field x of self is written **`(x self)`**

selecting the method defined for the object of class C (of course this function is well typed if $MyType \le C$).

6.3. Multiple dispatch

In this paper we have studied a very kernel calculus. A simple extension of this calculus allows us to model multiple dispatch, i.e. a mechanism of selection of methods (in this case called multi-methods), based not only on the class of the receiver but also on the class of further parameters. The simplest extension of $F_{\le}^{\&\top}$ to obtain multiple-dispatch consists in allowing as bounds of an overloaded function products of basic types. Thus we redefine atomic types in the following way

$$A \quad ::= \quad X \ | \ B \ | \ B \times \ldots \times B \qquad \text{(atomic types [}B\text{ basic types])}$$

we modify the condition in the rule of good formation for overloaded types as follows:

$$(\{\ \}_{type}) \qquad \frac{\begin{array}{c} C \vdash A_i \text{ type} \\ C \vdash \{A_i\}_{i=1..n}\ \cap\text{-closed} \\ C \cup \{X{\le}A_i\} \vdash T_i \text{ type} \\ \text{if } C \vdash A_i \le A_j \text{ then } C \cup \{X{\le}A_i\} \vdash T_i{\le}T_j \end{array}}{C \vdash \forall X\{A_1.T_1, \ldots, A_n.T_n\} \text{ type}} \qquad \begin{array}{c} X \notin dom(C) \\ \mathcal{B}(A_i)_C \text{ basic type} \\ \text{or } A_i = B_1 \times \ldots \times B_m \\ \text{for } i,j \in [1..n] \end{array}$$

and of course we add tuples to terms:

$$a ::= < a, \ldots, a >$$

There are other more general extensions: for example we can change the condition in the good formation of overloaded types into "$\mathcal{B}(A_i)_C$ basic type or product of atomic basic types" allowing as bounds variables ranging on the product of basic types; or we can allow as bounds products formed by type variables and basic types.

However the extension above largely suffice to model multi-methods, and furthermore it is very easy to check that it enjoys all the properties we have already proved for $F_{\le}^{\&}$ (and $F_{\le}^{\&\top}$): just run through the proofs by taking into account that now proposition 2.5 has the following fourth case:

4. A_i and A_j are both products of basic types and their g.l.b. is in $\{A_i\}_{i\in I}$

One example of use of multiple dispatch is the method $Equal$: you want to extend the class $2DPoint$ with a method that compares two points and to redefine it for $3DPoint$; furthermore you want that in comparing a $2DPoint$ with a $3DPoint$ the method for $2DPoint$ is used. In $\lambda\&$ we had that a function $Equal$ of type $\{2DPoint \to 2DPoint \to Bool, 3DPoint \to 3DPoint \to Bool\}$ would not have a well-formed type since covariance is not respected. So in $\lambda\&$ we defined

$$Equal : \{(2DPoint \times 2DPoint) \to Bool, (3DPoint \times 3DPoint) \to Bool\}$$

obtaining in this way multiple dispatching. When Equal is applied to $2DPoint$ and a $3DPoint$ or viceversa the first branch is executed.

In $F_{\le}^{\&}$ the difference is subtler: indeed

$$\forall X\{2DPoint.X \to X \to Bool, 3DPoint.X \to X \to Bool\}$$

is well formed. However to select the right branch you have to pass to a function of this type the greater of the types of the two actual parameters. This is not what one would like to have: one would like to pass to the function both the types of the arguments and leave to the system the task to select the right branch. This can be done by using multi-methods and defining $Equal$ with the following type:

$$\forall X\{2DPoint \times 2DPoint.X \to Bool, 3DPoint \times 3DPoint.X \to Bool\}$$

A possible implementation of $Equal$ is then

$$\begin{array}{lll} Equal \equiv & (\Lambda X \leq 2DPoint{\times}2DPoint. & \\ & & \lambda p^X.(\pi_1(p).x = \pi_2(p).x)\wedge \\ & & \quad (\pi_1(p).y = \pi_2(p).y) \\ & \;\&\; \Lambda X \leq 3DPoint \times 3DPoint. & \\ & & \lambda p^X.(\pi_1(p).x = \pi_2(p).x)\wedge \\ & & \quad (\pi_1(p).y = \pi_2(p).y)\wedge \\ & & \quad (\pi_1(p).z = \pi_2(p).z) \\ &) & \end{array}$$

If we want to use multiple dispatch in class-based languages then the type above must be slightly changed. In class-based languages the method is always chosen according to the class of the receiver but in with multiple dispatching a class may have different specifications for the method; one of these specifications will be selected according to the class of some extra parameters. Thus for example the equality function could be added to `2DPoint` and `3DPoint` in the following way:

```
class 2DPoint                          class 3DPoint is 2DPoint
  state                                  state
    x:Int;                                 z:Int
    y:Int                                methods
  methods                                   Equal(p:2DPoint)= ...
    Equal(p:Mytype)= ...                    Equal(p:Mytype)= ...
           :                                       :
           :                                       :
  interface                              interface
    Equal: Mytype -> Bool                   Equal:{2DPoint->Bool,
           :                                       Mytype ->Bool}
           :                                       :
    endclass                                       :
                                      endclass
```

where horizontal and vertical dots are substituted for the method bodies and the further methods, respectively.

When the message $Equal$ is passed to an object of class `3DPoint` then if the argument is of class `2DPoint` then the first definition of Equal is executed. The second definition is executed if the argument is of a type smaller than or equal to `3DPoint`. This correspond to have an Equal overloaded function of the following type.

$$\begin{array}{l} \forall X\{\ 2DPoint.\ X \to \forall Y\{X.\ Y \to Bool\}\ , \\ \qquad 3DPoint.\ X \to \forall Y\{2DPoint.\ Y \to Bool, X.\ Y \to Bool\} \\ \quad \} \end{array}$$

Here the variable X stands for $MyType$.

In case of languages with generic functions the application of the theory is easier. Consider the language Cecil [CL94, Cha92] and the following subtyping relation:

$$natural \leq integer \leq rational \leq real$$

In Cecil it is possible to define a generic function (multi-method in Cecil's jargon) `plus` with the following signature:

```
signature plus(natural,natural):natural;
signature plus(integer,integer):integer;
signature plus(real,real):real;
```

This correspond to have three different implementations for the operator `plus`, one for each signature. This corresponds to have in $\lambda\&$ the following typing

$$plus : \{natural \times natural \to natural, integer \times integer \to integer, real \times real \to real\}$$

If a and b are two terms of type $rational$ then in Cecil (as well as in $\lambda\&$) $plus(a, b)$ has type $real$ intead of $rational$. If instead we had typed it as

$$plus : \forall X\{natural.\ X \times X \to X, integer.\ X \times X \to X, real.\ X \times X \to X\}$$

then $plus[rational](a, b)$ would have type $rational$. However, some more work is needed in order to define a modification of the Cecil's syntax to take into account type variables. To that end it would be interesting to explore the extension of $F^{\&}_{\leq}$ with the intersection types. A possible syntax for the signature would then be:

```
signature plus(X<natural,Y<natural):X/\Y;
signature plus(X<integer,Y<integer):X/\Y;
signature plus(X<real,Y<real):X/\Y;
```

But this is subject for future work.

7. Future work

In this and in the previous chapter we defined and studied $F^{\&}_{\leq}$ and its decidable variant, and we sketched how they can be used to model object-oriented features. We showed that they account for many features of object-oriented programming and that they also suggest new features to add to the existing paradigms. However there are some features that are not easily handled (e.g. the keyword $super$; see at this purpose [Cas95b]).

The major restriction is that meet-closure allows overloading only on atomic types. In the last section we showed how to weaken this condition to model multiple dispatching; though also this definition still prevents us to model the generic classes of Eiffel [Mey88]. A generic class is a class parameterized by a type variable. For example if X is a type variable, one would like to define a class $Stack[X]$ with methods $pop\colon X$ and $push\colon X \to ()$, and then obtain a stack of integers by instantiating the type variable X in the following way: `new(Stack[Int])`. We believe that it is not difficult to further weaken meet-closure to allow among the bounds of an overloaded function, monotonic type constructors. But we are at a loss to think how to allow non monotonic type constructors. In the same way it should be possible to extend meet-closure to closed types and to add recursive types to implement recursive objects (even if we think that recursive types are not indispensable: see [PT93]).

Meet closure constitutes an even more serious limitation from a proof-theoretical point of view. It would be very interesting to let bounds range over all the types; this would require a suitable definition of $\cap$-closure assuring consistency also for higher order bounds. Note that the proof theory would be greatly complicated since a new level of impredicativity would be added. However this would correspond to a major increase of the expressive power. In that case, indeed, by a slight weakening of the $\beta_{\{\}}$

rule, it would be possible to obtain parametric functions as a special case of overloaded functions with only one branch.

To the end it seems very promising the extension of $F_{\leq}^{\&}$ by intersection types hinted in the last section. The $\cap$-closure would then correspont to requiring that the set of domains is closed for intersections.

Despite these problems $F_{\leq}^{\&}$ is a step forward in the research of the overloading-based model for object-oriented programming, since it gives us the basic type checking rules to deal the problem of the loss of information. At the moment of the redaction of this paper we are studying the integration of generic functions (i.e. overloaded functios with late binding) into the core of ML [MH88], in order to add object-oriented features to the languages of this family. Also underway is the definition of an object-oriented database programming language, whose type system is based on $F_{\leq}^{\&}$.

Acknowledgments

I am very grateful to Martín Abadi and Florian Matthes for their valuable suggestions on an earlier draft. Many thanks also to the two referees of this journal for their valuable advice. I want also to thank Giuseppe Longo for his constant advice, for his patient in reading the proofs and . . . a lot more.

References

[App92] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

[Bar84] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics.* North-Holland, 1984. Revised edition.

[Bru94] K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer-Verlag, 1984.

[Cas92] G. Castagna. Strong typing in object-oriented paradigms. Technical Report 92-11, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, June 1992.

[Cas94] G. Castagna. *Overloading, subtyping and late binding: functional foundation of object-oriented programming.* PhD thesis, Université Paris 7, January 1994. Appeared as LIENS technical report.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 1995. To appear. Available by anonymous ftp from **ftp.ens.fr** in file **/pub/dmi/users/castagna/covariance.dvi.Z**.

[Cas95b] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 1995. To appear. An extended abstract of this paper has appeared in the Proceeding of the *13th Conference on the Foundations of Software Technology and Theoretical Computer Science*; Lecture Notes in Computer Science number 761, December 1993.

[CCH^{+}89] P.S. Canning, W.R. Cook, W.L. Hill, J. Mitchell, and W.G. Olthoff. F-bounded quantification for object-oriented programming. In *ACM Conference on Functional Programming and Computer Architecture*, September 1989. Also in [GM94].

[CG92] P. L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and the type checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1), 1992.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping, 1992. To appear in *Information and Computation.* An extended abstract has appeared in the proceedings of the *ACM Conference on LISP and Functional Programming*, pp.182-192; San Francisco, June 1992.

[CGL93] G. Castagna, G. Ghelli, and G. Longo. A semantics for $\lambda\&$-*early*: a calculus with overloading and early binding. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[Cha92] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP'92*, number 615 in LNCS. Springer Verlag, 1992.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.

[CL91] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CL94] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In *OOPSLA'94*, 1994.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 750–771. Springer-Verlag, September 1991. LNCS 526 (preliminary version). To appear in Information and Computation.

[CP93] A. Compagnoni and B.C. Pierce. Multiple inheritance via intersection types. Unpublished manuscript, 1993.

[CP94] G. Castagna and B.C. Pierce. Decidable bounded quantification. In *21st Annual Symposium on Principles Of Programming Languages*, pages 151–162, Portland, Oregon, January 1994. ACM Press. POPL'94.

[CP95] G. Castagna and B.C. Pierce. Corrigendum: Decidable bounded quantification. In *22nd Annual Symposium on Principles Of Programming Languages*, San Francisco, January 1995. ACM Press.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DG87] L.G. DeMichiel and R.P. Gabriel. Common lisp object system overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1990. Tech. Rep. TD-6/90.

[Ghe91] G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[Hin64] R. Hindley. The Church-Rosser property and a result of combinatory logic. Dissertation, 1964. University of Newcastle-upon-Tyne.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and formalism.* Academic Press, 1980.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall International Series, 1988.

[MH88] J.C. Mitchell and R. Harper. The essence of ML. *15th Ann. ACM Symp. on Principles of Programming Languages*, January 1988.

[New42] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Math.*, 43(2):223–243, 1942.

[NeX91] NeXT Computer Inc. *NeXTstep-concepts. Chapter 3: Object-Oriented Programming and Objective-C*, 2.0 edition, 1991.

[Pie93] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1993. To appear. Also in [GM94]. Preliminary version in proceedings of POPL '92.

[PT93] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Ros73] B. K. Rosen. Tree manipulation systems and Church-Rosser theorems. *Journal of ACM*, 20:160–187, 1973.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

A. Proofs

A.1.

Proof of Lemma 4.3. By induction on the depth of the proof of $C \cup \{X \leq S\} \vdash \Delta$. For depth=1 there are only two possible cases: $\Delta \equiv B\ \mathrm{type}$ or $\Delta \equiv \mathsf{Top}\ \mathrm{type}$. In both cases the result is trivially satisfied. For depth>1 we perform a case analysis on the last rule of the proof:

(refl) a straightforward use of the induction hypothesis

(trans) a straightforward use of the induction hypothesis

(taut) suppose that $\Delta \equiv Y \leq T$ then there are two possible subcases:

1. $Y \not\equiv X$: a straightforward use of the induction hypothesis
2. $Y \equiv X$: then the hypothesis gets $C \cup \{X \leq S\} \vdash X \leq S$; since $X \notin FV(S)$ the result reduces to $C[X := S'] \vdash S' \leq S$ which holds by hypothesis

(top) a straightforward use of the induction hypothesis

$(\rightarrow)$ a straightforward use of the induction hypothesis

$(\forall)$ suppose that $\Delta \equiv \forall(Y \leq T_1)T_2 \leq \forall(Y \leq T_1')T_2'$. Recall that $C[X := S'] \vdash S' \leq S$. Thus by theorem 2.1 $C[X := S'] \vdash S'\ \mathrm{type}$ and therefore $FV(S') \subseteq dom(C[X := S'])$. By hypothesis we have that both $C \cup \{Y \leq T_1'\} \cup \{X \leq S\}$ and $C[X := S']$ are tcs's. Since $X \notin FV(S')$ then $FV(T_1'[X := S']) = (FV(T_1') \cup FV(S')) \backslash \{X\}$; thus also $C[X := S'] \cup \{Y \leq T_1'[X := S']\}$ is a tcs. Once this remark done, then the result follows by a straightforward use of the induction hypothesis.

$(\{\,\})$ As the previous case.

$(\mathbf{Vars}_{type})$ suppose that $\Delta \equiv Y\ \mathrm{type}$. Then there are two possible subcases:

1. $Y \not\equiv X$: a straightforward use of the induction hypothesis
2. $Y \equiv X$: then the result reduces to $C[X := S'] \vdash S'\ \mathrm{type}$ which follows from $C[X := S'] \vdash S' \leq S$ and theorem 2.1

$(\rightarrow_{type})$ a straightforward use of the induction hypothesis

$(\forall_{type})$ After having done the same remark as in the case $(\forall)$ the thesis follows from a straightforward use of the induction hypothesis.

$(\{\}_{type})$ This is the hard case. The pattern of the proof of this case is essentially the same as that of the case $(\forall)$. The hard task is to prove that $C \cup \{X \leq S\} \vdash \{A_i\}_{i=1..n} \cap$-closed, $C[X := S'] \vdash S' \leq S$ and the induction hypothesis imply

$$C[X := S'] \vdash \{A_i[X := S']\}_{i=1..n} \cap -\mathrm{closed}$$

This is equivalent to prove that whenever

$$\mathcal{B}(A_i[X := S'])_{C[X:=S']} \Downarrow \mathcal{B}(A_j[X := S'])_{C[X:=S']} \tag{6}$$

then there exists $h \in [1..n]$ such that

$$C[X := S'] \vdash A_h[X := S'] = A_i[X := S'] \cap A_j[X := S']$$

Suppose that (6) holds, and examine all the possible cases for A_i and A_j:

i. $(A_i$ **and** A_j **basic**$)$. Then $A_i[X\colon= S'] = A_i = \mathcal{B}(A_i[X\colon= S'])_{C[X:=S']} = \mathcal{B}(A_i)_{C\cup\{X\leq S\}}$ and the same for j. From the meet-closure of $\{A_i\}_{i=1..n}$ follows that there exists a basic type $A_h = A_h[X\colon= S'] = A_i \cap A_j = A_i[X\colon= S'] \cap A_j[X\colon= S']$ independently from the tcs we are taking into account.

ii. $(A_i{\equiv}A_j{\equiv}X)$ trivial

iii. $(A_i{\equiv}X$ **and** $A_j \not\equiv X)$. Then the hypothesis gets

$$\mathcal{B}(S')_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']} \tag{7}$$

We prove the result by showing that $S' \cap A_j$ is always either S' or A_j.
From $C[X\colon= S'] \vdash S'{\leq}S$ and proposition 2.3 we deduce that $\mathcal{B}(S')_{C[X:=S']} \leq \mathcal{B}(S)_{C[X:=S']}$ and then from (7) follows that

$$\mathcal{B}(S)_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']} \tag{8}$$

Now, first of all note that by definition of $\mathcal{B}$ one has $\mathcal{B}(X)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C\cup\{X\leq S\}}$.
Then observe that $\mathcal{B}(S)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']}$: this is obvious if S is a basic type; when S is a variable this follows from the fact that the substitution $[X\colon= S']$ does not affect the definition of $\mathcal{B}(S)$. Indeed if

$$C[X\colon= S'] \equiv C' \cup \{S{\leq}X_1\} \cup \{X_1{\leq}X_2\} \cup \ldots \cup \{X_n{\leq}\mathcal{B}(S)\} \qquad (n{\geq}0)$$

then $X \not\equiv X_i$ for all $i \in [1..n]$ otherwise $C \cup \{X{\leq}S\}$ would not be a tcs.
Thus from $\mathcal{B}(X)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C\cup\{X\leq S\}}$ and $\mathcal{B}(S)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']}$ we deduce

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']} \tag{9}$$

Now there are two possible subcases:

a. A_j is a basic type: then $\mathcal{B}(A_j)_{C\cup\{X\leq S\}} = A_j = \mathcal{B}(A_j)_{C[X:=S']}$ and thus (8) gets

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} \Downarrow \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$$

but since $C \cup \{X{\leq}S\} \vdash \{A_i\}_{i=1..n}$$\cap$-closed (and $X \in \{A_i\}_{i=1..n}$) we have that $C \cup \{X \leq S\} \vdash X \leq A_j$ (by proposition 2.5 the variable must be smaller than the basic type) and therefore $\mathcal{B}(S)_{C[X:=S']} = \mathcal{B}(X)_{C\cup\{X\leq S\}} \leq \mathcal{B}(A_j)_{C\cup\{X\leq S\}} = A_j$.
Thus $C[X\colon= S'] \vdash S'{\leq}S{\leq}\mathcal{B}(S)_{C[X:=S']}{\leq}\ A_j$ whence we can conclude that

$$C[X\colon= S'] \vdash S' = S' \cap A_j \tag{10}$$

b. A_j is a variable: then we have that

$$C[X\colon= S'] \equiv C'' \cup \{A_j{\leq}X_1\} \cup \{X_1{\leq}X_2\} \cup \ldots \cup \{X_n{\leq}\mathcal{B}(A_j)\} \qquad (n{\geq}0)$$

If $S' \equiv X_i$ for some $i \in [1..n]$ then $C[X\colon= S'] \vdash A_j \leq S'$ and therefore

$$C[X\colon= S'] \vdash A_j = S' \cap A_j \tag{11}$$

Otherwise if $S' \not\equiv X_i$ for all $i \in [1..n]$ then the substitution $[X\colon= S']$ does not affect the definition of $\mathcal{B}(A_j)$ and thus

$$\mathcal{B}(A_j)_{C\cup\{X\leq S\}} = \mathcal{B}(A_j)_{C[X:=S']}$$

Thus once more (8) and (9) yield

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} \Downarrow \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$$

Recall that both X and A_j are variables contained in $\{A_i\}_{i=1..n}$ and that $C \cup \{X \leq S\} \vdash \{A_i\}_{i=1..n}$ $\cap$-closed. Thus by proposition 2.5 either

$$C \cup \{X \leq S\} \vdash X \leq A_j$$

or

$$C \cup \{X \leq S\} \vdash A_j \leq X$$

must hold. Whichever judgment holds, we supposed in the assumption 4.2 that its proof is contained in the proof of meet closure of $\{A_i\}_{i=1..n}$; thus we can apply the induction hypothesis obtaining either (10) or (11), respectively.

iv. **(A_j and A_j are both different from X and at least one of them is a variable)** Thus $A_i[X\!:= S'] = A_i$ and $A_j[X\!:= S'] = A_j$ and the hypothesis becomes

$$\mathcal{B}(A_i)_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']}$$

Let us open a short parenthesis: suppose to have a type variable $Y \not\equiv X$ with $Y \in dom(C)$ and consider $\mathcal{B}(Y)_{C \cup \{X \leq S\}}$. Then if

$$C \cup \{X \leq S\} \equiv C' \cup \{Y \leq X_1\} \cup \{X_1 \leq X_2\} \cup \ldots \cup \{X_n \leq \mathcal{B}(Y)_{C \cup \{X \leq S\}}\} \qquad (n \geq 0)$$

there are two possible cases

(1) $X \equiv X_h$ for some $h \in [1..n]$ and in this case note that

$$\mathcal{B}(Y)_{C[X:=S']} = \mathcal{B}(S')_{C[X:=S']}$$

(2) $X \not\equiv X_h$ for all $h \in [1..n]$ and in this case

$$\mathcal{B}(Y)_{C[X:=S']} = \mathcal{B}(Y)_{C \cup \{X \leq S\}}$$

After this short remark we can now consider the various cases for A_i and A_j

a. A_i is a variable in the situation like Y in **(1)** and A_j is a basic type. But then by the point **(1)** the hypothesis becomes

$$\mathcal{B}(S')_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']}$$

which has already been solved in (iii).

b. A_i is a variable in a situation like Y in **(2)** and A_j is a basic type. By the meet-closure of $\{A_i\}_{i=1..n}$ and by the point **(2)** we deduce that

$$\mathcal{B}(A_i)_{C[X:=S']} = \mathcal{B}(A_i)_{C \cup \{X \leq S\}} \leq A_j = \mathcal{B}(A_j)_{C \cup \{X \leq S\}}$$

and thus $C[X\!:= S'] \vdash A_i \leq A_j$

c. A_i is a variable in the situation like in **(1)** and A_j is a variable in the situation like in **(2)**; but then we are in a case similar to the one of (a.)

d. A_i and A_j are both variables in the situation like in **(1)**. Then $\mathcal{B}(A_i)_{C[X:=S']}$ = $\mathcal{B}(S')_{C[X:=S']}$ = $\mathcal{B}(A_j)_{C[X:=S']}$. Thus either $C[X\!:= S'] \vdash A_i \leq A_j$ or $C[X\!:= S'] \vdash A_j \leq A_i$ holds.

e. A_i and A_j are both variables in the situation like in **(2)**. Thus $\mathcal{B}(A_i)_{C \cup \{X \leq S\}}$ $\Downarrow \mathcal{B}(A_j)_{C \cup \{X \leq S\}}$ and by the meet-closure either $C \cup \{X \leq S\} \vdash A_i \leq A_j$ or $C \cup \{X \leq S\} \vdash A_j \leq A_i$ holds. But since they are variables like in **(2)** this come to say that either $C[X\!:= S'] \vdash A_i \leq A_j$ or $C[X\!:= S'] \vdash A_j \leq A_i$ holds.

A.2.

Proof of Lemma 4.4. By induction on the structure of a:

$a \equiv y$ if $y \equiv x$ then $S \equiv T$ and $S' \equiv T'$; else if $y \not\equiv x$ the result trivially holds.

$a \equiv \varepsilon$ trivial

$a \equiv \mathsf{Top}$ trivial

$a \equiv \lambda y^{S_1}.a'$ if $y \equiv x$ then the result trivially holds; otherwise $S \equiv S_1 \to S_2$ and $C \vdash a' \colon S_2$. By induction hypothesis $C \vdash a'[x^T := b] : S_2' \leq S$ thus

$$C \vdash a[x^T := b] \equiv \lambda y^{S_1}.a'[x^T := b] : S_1 \to S_2' \leq S_1 \to S_2$$

$a \equiv (a_1 \&^I a_2)$ just note that by induction hypothesis $(a_1[x^T := b] \&^I a_2[x^T := b])$ is well-typed, and that its type is S.

$a \equiv \Lambda X {\leq} S_1.a'$ then $C \cup \{X \leq S_1\} \vdash a' \colon S_2$ with $S \equiv \forall (X \leq S_1) S_2$. By induction hypothesis $C \cup \{X {\leq} S_1\} \vdash a'[x^T := b] : S_2' \leq S_2$. Thus

$$C \vdash a[x^T := b] \equiv \Lambda X {\leq} S_1.a'[x^T := b] : \forall (X {\leq} S_1) S_2' \leq \forall (X {\leq} S_1) S_2$$

$a \equiv a_1(a_2)$ then $C \vdash a_1 \colon S_3$, $\mathcal{B}(S_3)_C = S_1 \to S$ and $C \vdash a_2 \colon S_2 {\leq} S_1$. By induction hypothesis $C \vdash a_1[x^T := b] : U_3 {\leq} S_3$ and $C \vdash a_2[x^T := b] : U_2 {\leq} S_2 {\leq} S_1$. By proposition 2.3 $C \vdash \mathcal{B}(U_3)_C \leq \mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then by proposition 2.2 it is of the form $U_1 \to U$ with $C \vdash S_1 {\leq} U_1$ and $C \vdash U {\leq} S$. Thus we have:

- $C \vdash a_1[x^T := b] : U_3$
- $C \vdash a_3[x^T := b] : U_2 {\leq} U_1$
- $\mathcal{B}(U_3)_C = U_1 \to U$

Then by [$\to$Elim$_{(\leq)}$] we obtain

$$C \vdash a[x^T := b] \equiv a_1[x^T := b](a_2[x^T := b]) : U {\leq} S$$

$a \equiv a'(U)$ then $C \vdash a' \colon S_3$, $\mathcal{B}(S_3)_C = \forall (X \leq S_1) S_2$, $C \vdash U {\leq} S_1$ and $S \equiv S_2[X := U]$. Note that $X \notin dom(C)$ and thus $X \notin FV(U)$. By induction hypothesis $C \vdash a'[x^T := b] : U_3 {\leq} S_3$ and by proposition 2.3 $C \vdash \mathcal{B}(U_3)_C \leq \mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then by proposition 2.2 it is of the form $\forall (X {\leq} S_1') S_2'$. Since $C \cup \{X {\leq} S_1'\} \vdash S_2' \leq S_2$, $C \vdash U {\leq} S_1 {\leq} S_1'$ and $X \notin FV(U)$ we can apply the main lemma and obtain

$$C[X := U] \vdash S_2'[X := U] \leq S_2[X := U]$$

But $X \notin dom(C)$ thus $C[X := U] = C$, from which it follows

$$C \vdash a[x^T := b] : S_2'[X := U] \leq S_2[X := U]$$

$a \equiv a'[A]$ then $C \vdash a' \colon S_3$, $\mathcal{B}(S_3)_C = \forall X \{A_i.T_i\}_{i \in I}$ and $S \equiv T_h[X := A]$ where $C \vdash A_h = \min_{i \in I}\{A_i \mid C \vdash A {\leq} A_i\}$.

By induction hypothesis $C \vdash a'[x^T := b] : U_3 \leq S_3$ and by proposition 2.3 $C \vdash \mathcal{B}(U_3)_C \leq \mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then it is of the form $\forall X \{A_j'.T_j'\}_{j \in J}$. Thus by the subtyping rule ($\{\}$) there exists $\tilde{h} \in J$ such that $C \vdash A \leq A_h \leq A_{\tilde{h}}'$. Therefore the set $\{A_j' \mid C \vdash A \leq A_j', j \in J\}$ is not empty, and by the meet-closure of $\{A_j'\}_{j \in J}$ it has also a minimum. Call this minimum A_k'. Then $C \vdash a'[A][x^T := b] : T_k'[X := A]$. Since $S \equiv T_h[X := A]$ we have to prove that

$$C \vdash T_k'[X := A] \leq T_h[X := A]$$

Take again the previous $\tilde{h}$; by the rule ($\{\}$) we have

$$C \vdash \forall (X {\leq} A_{\tilde{h}}') T_{\tilde{h}}' \leq \forall (X {\leq} A_h) T_h \tag{12}$$

By the definition of A_h:

$$C \vdash A \leq A_h \tag{13}$$

From (12):

$$C \vdash A_h \leq A_{\tilde{h}}'$$

From (trans):

$$C \vdash A \leq A'_{\tilde{h}} \tag{14}$$

From (12):

$$C \cup \{X \leq A_h\} \vdash T'_{\tilde{h}} \leq T_h \tag{15}$$

From the definition of A'_k and from (14) we obtain

$$C \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule $(\{\}_{type})$ applied to $\forall X\{A'_j.T'_j\}_{j\in J}$ it follows

$$C \cup \{X \leq A_h\} \vdash T'_k \leq T'_{\tilde{h}} \tag{16}$$

From $X \notin dom(C)$ and from (13) we deduce that $X \notin FV(A)$; by (13) and by the choice of k we respectively have that $C \vdash A \leq A_h$ and $C \vdash A \leq A'_k$; thus we can apply the main lemma to (15) and (16) to obtain:

$$C[X\!:= A] \vdash T'_{\tilde{h}}[X\!:= A] \leq T_h[X\!:= A]$$
$$C[X\!:= A] \vdash T'_k[X\!:= A] \leq T'_{\tilde{h}}[X\!:= A]$$

But $X \notin dom(C)$, thus the judgements above get

$$C \vdash T'_{\tilde{h}}[X\!:= A] \leq T_h[X\!:= A]$$
$$C \vdash T'_k[X\!:= A] \leq T'_{\tilde{h}}[X\!:= A]$$

Finally by (trans) we obtain the result:

$$C \vdash T'_k[X\!:= A] \leq T_h[X\!:= A]$$

□

A.3.

Proof of Lemma 4.5. By induction on the structure of a:

$a\equiv x^T$ then $T' \equiv T[X\!:= S']$.

$a\equiv \varepsilon$ trivial

$a\equiv \mathsf{Top}$ trivial

$a\equiv \lambda x^{T_1}.a'$ where $T \equiv T_1 \to T_2$ and $C \cup \{X\leq S\} \vdash a'\!:T_2$. Thus by induction hypothesis we deduce that $C[X\!:= S'] \vdash a'[X\!:= S'] : T'_2 \leq T_2[X\!:= S']$. Therefore $C[X\!:= S'] \vdash a[X\!:= S'] = \lambda x^{T_1[X:=S']}.a'[X\!:= S'] : T_1[X\!:= S'] \to T'_2 \leq T[X\!:= S']$.

$a\equiv \Lambda Y\leq T_1.a'$ First of all note that $Y \not\equiv X$, since by hypothesis $C \cup \{X\leq S\} \vdash \Lambda Y\leq T_1.a'\!:T$ and we have made the assumption of having all the type variables different in a tcs. Thus $C \cup \{X \leq S\} \cup \{Y \leq T_1\} \vdash a'\!:T_2$ and $T \equiv \forall(Y \leq T_1)T_2$. Note that $C[X\!:= S']$ and $C \cup \{X\leq S\}$ are tcs's, and also that $FV(S') \subseteq C[X\!:= S']$ (since $C[X\!:= S'] \vdash S'\leq S$). Thus we can conclude that also $C[X\!:= S'] \cup \{Y\leq T_1[X\!:= S']\}$ is a tcs, being $dom(C) = dom(C[X\!:= S'])$ and $FV(T_1[X\!:= S']) = (FV(T_1) \cup FV(S'))\backslash\{X\}$ (the latter because $X \notin FV(S)$). Then by a weakening we can prove that $(C \cup \{Y\leq T_1\})[X\!:= S'] \vdash S'\leq S$. By induction hypothesis thus we have $C[X\!:= S'] \cup \{Y\leq T_1[X\!:= S']\} \vdash a'[X\!:= S']\!:T'_2 \leq T_2[X\!:= S']$. Thus by $[\forall\text{INTRO}]$ and $(\forall)$ we have that

$$C[X\!:= S'] \vdash \Lambda Y\leq T_1[X\!:= S'].a'[X\!:= S']\!:\forall(Y\leq T_1[X\!:= S'])T'_2 \leq T[X\!:= S']$$

$a\equiv(a_1\&^{[A_1.T_1\|\ldots\|A_n.T_n]}a_2)$ Thus $T\equiv\forall Y\{A_1.T_1,\ldots,A_n.T_n\}$, $C\cup\{X\leq S\}\vdash a_1 : S_1\leq\forall Y\{A_i.T_i\}_{i=1..n-1}$ and $C\cup\{X\leq S\}\vdash a_2 : S_2\leq\forall(Y\leq A_n)T_n$.
Since $C\cup\{X\leq S\}\vdash S'\leq S$ we can apply the main lemma (lemma 4.3) to the two judgements above obtaining respectively

$$C[X:=S']\vdash S_1[X:=S']\leq\forall Y\{A_i[X:=S'].T_i[X:=S']\}_{i=1..n-1}$$

$$C[X:=S']\vdash S_2[X:=S']\leq\forall(Y\leq A_n[X:=S'])(T_n[X:=S'])$$

Furthermore by induction hypothesis

$$C[X:=S']\vdash a_i[X:=S'] : S_i'\leq S_i[X:=S'] \qquad i=1,2$$

Recall that by definition

$$a[X:=S']=(a_1[X:=S']\&^{[A_1[X:=S'].T_1[X:=S']\|\ldots\|A_n[X:=S'].T_n[X:=S']]}a_2[X:=S'])$$

Therefore using transitivity and the rule $[\{\}\textsc{Intro}]$ we can conclude that

$$C[X:=S']\vdash a[X:=S'] : \forall Y\{A_i[X:=S'].T_i[X:=S']\}_{i=1..n}=T[X:=S']$$

$a\equiv a_1(a_2)$ Let $C\cup\{X\leq S\}\vdash a_1 : W, C\cup\{X\leq S\}\vdash a_2 : U'\leq U$ and $\mathcal{B}(W)_{C\cup\{X\leq S\}}=U\to T$. By induction hypothesis we have:

$$C[X:=S']\vdash a_1[X:=S'] : W'\leq W[X:=S']$$
$$C[X:=S']\vdash a_2[X:=S'] : U''\leq U'[X:=S']$$

Applying the main lemma (4.3) to $C\cup\{X\leq S\}\vdash U'\leq U$ and (trans) we obtain

$$C[X:=S']\vdash U''\leq U[X:=S']$$

By proposition 2.3

$$C[X:=S']\vdash\mathcal{B}(W')_{C[X:=S']}\leq\mathcal{B}(W[X:=S'])_{C[X:=S']} \tag{17}$$

Set $\overline{W}\equiv W[X:=S']$. We want to prove that

$$C[X:=S']\vdash\mathcal{B}(\overline{W})_{C[X:=S']}\leq\mathcal{B}(W)_{C\cup\{X\leq S\}}[X:=S'] \tag{18}$$

If $\overline{W}$ is not a variable this follows from (refl). Otherwise let

$$C\cup\{X\leq S\}\equiv C'\cup\{\overline{W}\leq X_1\}\cup\{X_1\leq X_2\}\cup\ldots\cup\{X_n\leq\mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}\} \qquad (n\geq 0)$$

There are two subcases:

1. $X\not\equiv X_i$ for all $i\in[1..n]$; then $\mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}=\mathcal{B}(\overline{W})_{C[X:=S']}$
2. $X\equiv X_i$ for some $i\in[1..n]$; then
$$\mathcal{B}(\overline{W})_{C[X:=S']}=\mathcal{B}(S')_{C[X:=S']}$$
$$\mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}=\mathcal{B}(S)_{C\cup\{X\leq S\}}$$

   Now it is easy to check that $\mathcal{B}(S')_{C[X:=S']}=\mathcal{B}(S')_{C\cup\{X\leq S\}}$ (otherwise $C[X:=S']$ and $C\cup\{X\leq S\}$ could not both satisfy the conditions of tcs). Thus by proposition 2.3 we obtain
$$\begin{aligned}C\cup\{X\leq S\}\vdash\mathcal{B}(\overline{W})_{C[X:=S']} &= \mathcal{B}(S')_{C[X:=S']}\\ &= \mathcal{B}(S')_{C\cup\{X\leq S\}}\\ &\leq \mathcal{B}(S)_{C\cup\{X\leq S\}}\\ &= \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}\end{aligned}$$

Thus in both cases we have that

$$C\cup\{X\leq S\}\vdash \mathcal{B}(\overline{W})_{C[X:=S']}\leq \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}$$

We can then apply the main lemma and obtain

$$C[X:=S']\vdash \mathcal{B}(\overline{W})_{C[X:=S']}[X:=S']\leq \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}[X:=S']$$

By hypothesis $X\notin FV(S')$; this implies that $X\notin FV(C[X:=S'])$ and thus $\mathcal{B}(\overline{W})_{C[X:=S']}\,[X:=S'] = \mathcal{B}(\overline{W})_{C[X:=S']}$. Therefore to conclude the proof of (18) it just remains to prove the following equation:

$$C[X:=S']\vdash \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}[X:=S']\leq \mathcal{B}(W)_{C\cup\{X\leq S\}}[X:=S'] \qquad (19)$$

This is obvious if W is not a variable (since $X\notin FV(S')$ then the substitution $[X:=S']$ is idempotent) or if it is a variable different from X (then $\overline{W}=W$). If $W\equiv X$ then just note that (19) gets

$$C[X:=S']\vdash \mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S']\leq \mathcal{B}(X)_{C\cup\{X\leq S\}}[X:=S']$$

by observing that $\mathcal{B}(X)_{C\cup\{X\leq S\}}=\mathcal{B}(S)_{C\cup\{X\leq S\}}$ this judgments becomes:

$$C[X:=S']\vdash \mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S']\leq \mathcal{B}(S)_{C\cup\{X\leq S\}}[X:=S'] \qquad (20)$$

To prove it first apply proposition 2.3 to the hypothesis $C[X:=S']\vdash S'\leq S$ and obtain

$$C[X:=S']\vdash \mathcal{B}(S')_{C[X:=S']}\leq \mathcal{B}(S)_{C[X:=S']} \qquad (21)$$

Assume now that we have proved that $\mathcal{B}(S')_{C[X:=S']}=\mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S']$ and $\mathcal{B}(S)_{C[X:=S']}=\mathcal{B}(S)_{C\cup\{X\leq S\}}[X:=S']$. In this case (21) implies (20). So let us prove the assumption: we start with S'. When S' is not a type variable then the result follows from the definition of $\mathcal{B}$ and the fact that $X\notin FV(S')$. If S' is a type variable then

$$C[X:=S']\equiv C'\cup\{S'\leq X_1\}\cup\ldots\cup\{X_n\leq \mathcal{B}(S')_{C[X:=S']}\}$$

Since $C\cup\{X\leq S\}$ is a tcs then $X\not\equiv X_i$ for all $i\in[1..n]$. By this

$$C\cup\{X\leq S\}\equiv C''\cup\{S'\leq X_1\}\cup\ldots\cup\{X_n\leq T''\}\cup\{X\leq S\}$$

Note that T'' cannot be a type variable: it cannot be X otherwise $\mathcal{B}(S')_{C[X:=S']}=S'$ (a loop in a tcs); it cannot be another variable otherwise $C[X:=S'](X_n)$ would be a variable, too. Therefore T'' is not a type variable which implies that $T''=\mathcal{B}(S')_{C\cup\{X\leq S\}}$ and thus $\mathcal{B}(S')_{C[X:=S']}=\mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S']$. A similar proof holds for S, too.
This ends the proof of (18)

From (17) and (18) we obtain:

$$C[X:=S']\vdash \mathcal{B}(W')_{C[X:=S']}\leq U[X:=S']\to T[X:=S']$$

Since $\mathcal{B}(W')_{C[X:=S']}$ is not a variable then it must be of the form $U'''\to T'$ with $C[X:=S']\vdash U[X:=S']\leq U'''$ and $C[X:=S']\vdash T'\leq T[X:=S']$.
Summing up we have:
- $C[X:=S']\vdash a_1[X:=S']:W'$
- $C[X:=S']\vdash a_2[X:=S']:U''\leq U'''$
- $\mathcal{B}(W')_{C[X:=S']}=U'''\to T'$

Then by$[\to\textsc{Elim}_{(\leq)}]$ we obtain

$$C[X:=S']\vdash a[X:=S']\equiv a_1[X:=S'](a_2[X:=S']):T'\leq T[X:=S']$$

$a \equiv a'(U)$ Let $C \cup \{X \leq S\} \vdash a' : W, C \cup \{X \leq S\} \vdash U \leq U', \mathcal{B}(W)_{C\cup\{X\leq S\}} = \forall(Y \leq U')U''$ and $T \equiv U''[Y := U]$. First of all note that $Y \notin dom(C) \cup \{X\}$; then by induction hypothesis

$$C[X := S'] \vdash a'[X := S'] : W' \leq W[X := S']$$

By the main lemma we have that

$$C[X := S'] \vdash U[X := S'] \leq U'[X := S'] \qquad (22)$$

Proceeding exactly as in the previous case we can prove that

$$C[X := S'] \vdash \mathcal{B}(W')_{C[X:=S']} \leq \mathcal{B}(W)_{C\cup\{X\leq S\}}[X := S']$$

Since $\mathcal{B}(W')_{C[X:=S']}$ is not a variable then it is of the form $\forall(Y \leq V')V''$ with

$$C[X := S'] \vdash U'[X := S'] \leq V'$$

$$C[X := S'] \cup \{Y \leq U'[X := S']\} \vdash V'' \leq U''[X := S'] \qquad (23)$$

Thus we have:
- $C[X := S'] \vdash a'[X := S'] : W'$
- $C[X := S'] \vdash U'[X := S'] \leq V'$
- $\mathcal{B}(W')_{C[X:=S']} = \forall(Y \leq V')V''$

Therefore by $[\forall\text{ELIM}]$ we obtain:

$$C[X := S'] \vdash a[X := S'] = a'[X := S'](U[X := S']) : V''[Y := U[X := S']]$$

Now from the hypothesis $C[X := S'] \vdash S' \leq S$ and from $Y \notin dom(C)$ we deduce that $Y \notin FV(S')$; from $C \cup \{X \leq S\} \vdash U \leq U'$ and $Y \notin (dom(C) \cup \{X\})$ we deduce that $Y \notin FV(U)$. Thanks to this and to (22) we can apply the main lemma to (23) and obtain

$$C[X := S'][[Y := U[X := S']] \vdash V''[Y := U[X := S']] \leq U''[X := S'][Y := U[X := S']] \qquad (24)$$

Since $Y \notin FV(S')$ then

$$([X := S'][Y := U[X := S']]) = ([Y := U][X := S'])$$

and then (24) rewrites to

$$C[Y := U][X := S'] \vdash V''[Y := U[X := S']] \leq U''[Y := U][X := S'] = T[X := S']$$

and since $Y \notin dom(C)$ it becomes

$$C[X := S'] \vdash V''[Y := U[X := S']] \leq T[X := S']$$

i.e. the result.

$a \equiv a'[A]$

Let $C \cup \{X \leq S\} \vdash a' : W$, $C \cup \{X \leq S\} \vdash A_h = \min_{i\in I}\{A_i \mid C \cup \{X \leq S\} \vdash A \leq A_i\}$, $\mathcal{B}(W)_{C\cup\{X\leq S\}} = \forall Y\{A_i.T_i\}_{i\in I}$ and $T \equiv T_h[Y := A]$. Again $Y \notin dom(C) \cup \{X\}$. By induction hypothesis

$$C[X := S'] \vdash a'[X := S'] : W' \leq W[X := S']$$

Applying the main lemma we also obtain that
$C[X := S'] \vdash \min_{i\in I}\{A_i[X := S'] \mid C[X := S'] \vdash A[X := S'] \leq A_i[X := S']\} \leq A_h[X := S']$

Proceeding as in the two previous cases we have that

$$C[X := S'] \vdash \mathcal{B}(W')_{C[X:=S']} = \forall Y\{A'_j.T'_j\}_{j\in J} \leq \forall Y\{A_i[X := S'].T_i[X := S']\}_{i\in I} \qquad (25)$$

By the rule ($\{\}$) for each $i \in I$ there exists $j \in J$ such that $C[X\colon= S'] \vdash A_i[X\colon= S'] \leq A'_j$. Thus by the main lemma we have that $\{A'_j \mid C[X\colon= S'] \vdash A[X\colon= S'] \leq A'_j, j \in J\}$ is not empty, and by the meet-closure of $\{A'_j\}_{j \in J}$ it has also a minimum. Therefore if

$$C[X\colon= S'] \vdash A'_k = \min_{j \in J}\{A'_j \mid C[X\colon= S'] \vdash A[X\colon= S'] \leq A'_j\}$$

then $C[X\colon= S'] \vdash a[X\colon= S'] : T'_k[Y\colon= A[X\colon= S']]$. Consider now (25); by the rule ($\{\}$) one has that there exists $\tilde{h} \in J$ such that

$$C[X\colon= S'] \vdash \forall(Y \leq A'_{\tilde{h}})T'_{\tilde{h}} \leq \forall(Y \leq A_h[X\colon= S'])(T_h[X\colon= S']) \tag{26}$$

Since $C \cup \{X \leq S\} \vdash A \leq A_h$ then by the main lemma

$$C[X\colon= S'] \vdash A[X\colon= S'] \leq A_h[X\colon= S'] \tag{27}$$

From (26):

$$C[X\colon= S'] \vdash A_h[X\colon= S'] \leq A'_{\tilde{h}}$$

From (trans):

$$C[X\colon= S'] \vdash A[X\colon= S'] \leq A'_{\tilde{h}} \tag{28}$$

From (26):

$$C[X\colon= S'] \cup \{Y \leq A_h[X\colon= S']\} \vdash T'_{\tilde{h}} \leq T_h[X\colon= S'] \tag{29}$$

From the definition of A'_k and from (28) we obtain

$$C[X\colon= S'] \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule ($\{\}_{type}$) applied to $\forall Y\{A'_j.T'_j\}_{j \in J}$ it follows that

$$C[X\colon= S'] \cup \{Y \leq A'_k\} \vdash T'_k \leq T'_{\tilde{h}} \tag{30}$$

From (27) and $Y \notin dom(C)$ (and thus $Y \notin dom(C[X\colon= S'])$) follows that $Y \notin FV(A[X\colon= S'])$; by (27) and by the choice of k we respectively have that $C[X\colon= S'] \vdash A[X\colon= S'] \leq A_h[X\colon= S']$ and $C[X\colon= S'] \vdash A[X\colon= S'] \leq A'_k$; thus we can apply the main lemma to (29) and (30) to obtain:

$$C[X\colon= S'][Y\colon= A[X\colon= S']] \vdash T'_{\tilde{h}}[Y\colon= A[X\colon= S']] \leq T_h[X\colon= S'][Y\colon= A[X\colon= S']]$$

$$C[X\colon= S'][Y\colon= A[X\colon= S']] \vdash T'_k[Y\colon= A[X\colon= S']] \leq T'_{\tilde{h}}[Y\colon= A[X\colon= S']]$$

But $Y \notin dom(C)$, thus $Y \notin dom(C[X\colon= S'])$ and whence, by the definition of tcs, $Y \notin FV(C[X\colon= S'])$. Then the judgements above get

$$C[X\colon= S'] \vdash T'_{\tilde{h}}[Y\colon= A[X\colon= S']] \leq T_h[X\colon= S'][Y\colon= A[X\colon= S']]$$

$$C[X\colon= S'] \vdash T'_k[Y\colon= A[X\colon= S']] \leq T'_{\tilde{h}}[Y\colon= A[X\colon= S']]$$

By (trans):

$$C[X\colon= S'] \vdash T'_k[Y\colon= A[X\colon= S']] \leq T_h[X\colon= S'][Y\colon= A[X\colon= S']]$$

From $C[X\colon= S'] \vdash S' \leq S$ and $Y \notin dom(C[X\colon= S'])$ follows that $Y \notin FV(S')$. Thus the last judgement becomes:

$$C[X\colon= S'] \vdash T'_k[Y\colon= A[X\colon= S']] \leq T_h[Y\colon= A][X\colon= S'] = T[X\colon= S']$$

□