# Complexity of kernel Fun subtype checking

Giorgio Ghelli *

Dipartimento d'Informatica

Corso Italia 40, Pisa, ITALY

e-mail: ghelli@di unipi.it

## Abstract

System kernel Fun is an abstract version of the system Fun defined by Cardelli's and Wegner's seminal paper [CW85], and is strictly related to system $F_{\leq}$. Extensions of these two systems are currently the basis of most proposals for strong type systems for object-oriented languages.

We study here the problem of subtype checking for system kernel Fun, presenting the following results. We show that the standard kernel Fun subtype checking algorithm has an exponential complexity, and generates an exponential number of different subproblems. We then present a new subtype checking algorithm which has a polynomial complexity. In the process we study how variable names can be managed by a kernel Fun subtype checker which is not based on the De Bruijn encoding, and we show how to perform kernel Fun subtype checking with a "constraint generating" technique.

The algorithm we give is described by a set of type rules, which we prove to be equivalent to the standard one. This new presentation of kernel Fun type system is characterized by a "multiplicative" behaviour, and it may open the way to new presentations for system $F_{\leq}$ as well.

**Keywords:** type theory, type checking, subtyping, polymorphism.

*This author has been partially supported by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica"



**ICFP '96 5/96 PA, USA**
**© 1996 ACM 0-89791-771-5/96/0005...$3.50**

## 1 Introduction

### 1.1 $F_{\leq}$ and kernel Fun

The importance of a strong foundation for systems integrating parametric polymorphism with subtyping has become increasingly evident in recent years, thanks to the growing interest in the definition of strong type systems for object-oriented languages. Most of the type systems which have been proposed for this purpose are based on system $F_{\leq}$ and kernel Fun (see, for example, [DT88], [CHO88], [CCHO89], [CHC90], [Mit90], [CMMS94], [Ghe91], [Bru91], [PT94]. [MHF93], [GM94], [Bru94], [ESTZ94], [HP95], [BSvG95], [FM94], [EST95], [AC94]). $F_{\leq}$ and kernel Fun are two different abstract versions of the language Fun, introduced by Cardelli and Wegner [CW85].

System $F_{\leq}$ enjoys many interesting properties: strong normalization, the existence of a minimum type for every term, transitivity and subsumption elimination, etc. [Ghe90, CL91, BCGS91, CMMS94, CG92, CG94]. However, it also has some significant problems: $F_{\leq}$ subtyping, and $F_{\leq}$ type-checking, are semidecidable only [CG92, Pie94]; $F_{\leq}$ types fail to form a semilattice [Ghe90, GP92]; $F_{\leq}$ extension with recursive types is not conservative [Ghe93b]. The technical source of all of these problems lies in the rule which $F_{\leq}$ uses to compare bounded universal types. These problems disappear when bounded universal types are compared according to the more restrictive rule defined for system Fun in Cardelli's and Wegner's original paper [CW85]; the resulting system is called kernel Fun.

System kernel Fun enjoys all the good properties of $F_{\leq}$ mentioned above, and the missing ones too (semilattice, decidability, conservativity with recursion). For this reason it has been used, instead of $F_{\leq}$, as the basis of some higher order type systems with subtyping ([Com94, PS]), even if it is somehow less expressive than system $F_{\leq}$.

Though kernel Fun subtyping, which is the hard core

of the type-checking procedure, is known to be decidable (folk theorem, see [Ghe93a]), nothing was known about the complexity of the subtyping procedure. In this paper we show that the problem is polynomial, but that the standard subtype checking algorithm is exponential, and we actually present a polynomial algorithm. This polynomial algorithm is rather different from the standard one, and is based on the adoption of a different presentation of kernel Fun subtype rules. Though we present this result in the context of the kernel Fun system, the standard, exponential, subtype checking algorithm is essentially the one that is currently used for every type system which belongs to the family of Cardelli's and Wegner's Fun. Hence, our result may be applied to a wider class of type systems. We will not study this point here, while we will show in the paper that our algorithm can *not* be easily transferred to system $F_{\leq}$; this observation may give some more insight on the differences between the two systems.

This study has been carried out in the attempt of finding more efficient algorithms to type check the object-oriented database programming language Fibonacci, realized by the group lead by Antonio Albano in Pisa University [AGO95], and some of the features of the algorithm we present have already been included in the Fibonacci type-checker.

However, the main interest of our result is, in our opinion, that it shows that the standard subtype checking algorithm is not the only reasonable one, and that the standard approach used to write kernel Fun rules is not the only possible one. Using linear logic jargon, the main difference between the sets of rules which describe the subtype-checking and the constraint-generating algorithms is that the former manages environments in an additive way, while the latter manages environments in a multiplicative way [Gir87].[1] From an algorithmic point of view, this is simply a consequence of the fact that environments are an input parameter (for the subtype checker) in the first case, and an output parameter in the second case. However, it is intriguing to wonder whether a logical interpretation of this difference exists, and whether it may be possible, and worthwhile, to look for a multiplicative presentation of system $F_{\leq}$ itself.

### 1.2 Technical outline and plan of the paper

The definition of the polynomial algorithm proceeds in three steps.

First, we define a specific form of the standard kernel Fun subtype checking algorithm, "the $\alpha$-free subtype checking algorithm", characterized by the fact that, during the process of comparing two types, it never creates new type variables, nor does it renames any type variable. This aim is reached in three steps. First of all, the algorithm starts with a judgement where no name clashes exist between different variables. Secondly, a comparison $\forall t \leq T.U \leq \forall t' \leq T.U'$ is not reduced to $[t'/t]U \leq U'$, but the algorithm collects the unification $t = t'$ in its "unification environment" $E$, and reduces the comparison $E \vdash \forall t \leq T.U \leq \forall t' \leq T.U'$ to $E, t = t' \vdash U \leq U'$. Finally, when a variable is substituted by its bound during the subtype checking process, this algorithm does not rename the variables which are defined inside that bound. We prove that this algorithm is equivalent to the standard algorithm. This is a crucial point, since this result cannot be immediately extended to $F_{\leq}$: any subtype checking algorithm for $F_{\leq}$ must create new type variables in the process [Ghe95].

Then, we define a new "constraint generating" algorithm, characterized by the fact that, while the $\alpha$-free one takes a subtype judgement $E \vdash T_1 \leq T_2$ and checks it, the new one takes a pair of types $T_1 \leq T_2$ and returns a set of constraints that must be satisfied by $E$ for $E \vdash T \leq U$ to hold. More specifically, to prove that $E \vdash \forall t \leq T.U \leq \forall t' \leq T.U'$, the $\alpha$-free algorithm explores the two types $U$ and $U'$ in parallel in a unification environment $E, t = t'$ and, when it finds a subproblem $u \leq v$, it answers by checking whether $u = v$ is in the unification environment. On the other hand, to analyze the same $\forall t \leq T.U \leq \forall t' \leq T.U'$ pair, the constraint generating algorithm first explores $U$ and $U'$ collecting all the subproblems with shape $u \leq v$ it finds (the "constraints"), ignoring, in this phase, the unification of $t$ with $t'$. Only after this collection has been completed, the algorithm cancels those constraints, such as $t \leq t'$, $t' \leq t$, that are solved by the unification of $t$ and $t'$, and returns the remaining ones. Hence, instead of having the unification environment as input, this algorithm gives a constraint set as output.

Finally, we show that a memoization technique[2] can be applied to the constraint generating algorithm to make it polynomial. Without memoization, both algorithms may take exponential time, as will be shown. However, both algorithms work by reducing the analysis of $(E \vdash) T \leq U$ to the analysis of a set of subproblems $(E_i \vdash) T_i \leq U_i$, where $T_i$ and $U_i$ are subterms of $T$ and $U$ ($E, E_i$ are only needed by the subtype checking algorithm), without generating any new types during the proof. If the size of $T$ and $U$ is $n$, then there are at most $n^2$ different $T_i, U_i$ pairs of subterms of $T, U$. Hence, if the constraint algorithm remembers and reuses the set

[1] In linear logic, a binary rule is additive when the premises have equal contexts, and is multiplicative when the context in the consequence is a combination of the contexts in the premises.

[2] Memoizing means storing in a table the already solved subproblems together with their solution, so that, if a subproblem is met again, it can be solved by a table lookup.

of constraints generated by each already checked subproblem, then it calls itself recursively no more than $n^2$ times. The same optimization applied to the subtype-checking algorithm would not make it polynomial, since in this case we have to store the already visited triples $E, T, U$, and a single judgement can generate, as we will show, an exponential number of different $E, T, U$ subproblems, due to the presence of an exponential number of different environments. This is the reason behind the switch from the "environment as input" to the "environment as output" viewpoint.

We will describe the subtype checking and the constraint generating algorithms through two sets of subtype rules, and will prove their correctness and completeness by showing that these two sets of rules are equivalent, and are both equivalent to the standard, non algorithmic, set of subtype rules for kernel Fun.

In Section 2 we define kernel Fun by giving its standard nameless presentation, where "nameless" means that variable names are irrelevant since $\alpha$ conversion can always be applied. In Section 3 we introduce a type system which models the $\alpha$-free subtype checking algorithm, characterized by the fact that it neither renames type variables nor generates new types, and we prove its equivalence to the nameless one. In Section 4 we introduce the type system which models the constraint generating algorithm, and prove its equivalence to the previous systems. In Section 5 we introduce the memoization technique which makes the constraint generating algorithm polynomial, and we also show that the standard algorithm is exponential and would remain exponential even if the memoization technique were applied to it. In Section 6 we comment on related work. In Section 7 we draw some conclusions, and make suggestions for future work.

Though we have carried out all the proofs in full detail, we can only give the outline of the main proofs in the paper, for space reasons.

## 2 The nameless rules for kernel Fun

### 2.1 The system

In this section we report the standard presentation of system kernel Fun, which is characterized by the irrelevance of variable names. We formalize this feature with an explicit rule which allows $\alpha$ renaming to be freely performed during a subtyping proof. We consider the following syntax for kernel Fun types and subtyping judgements.

$$
\begin{array}{lcl}
T & ::= & t \mid T \to T \mid \forall t{\leq}T.T \mid \mathrm{Top} \\
\Gamma & ::= & () \mid \Gamma, t{\leq}T \\
P & ::= & (\Gamma) \mid \langle \Gamma, \{T\} \rangle \mid \langle \Gamma, \{T, T\} \rangle \\
J & ::= & \Gamma \vdash_{\mathcal{G}} \mathcal{G}\text{-Env} \mid \Gamma \vdash_{\mathcal{G}} T \ \mathrm{Type} \mid \Gamma \vdash_{\mathcal{G}} T \leq T
\end{array}
$$

Pre-judgements $P$ are used to denote "judgements which have not yet been proved"; for an introduction to the other syntactic categories, and to the language, see [CW85, Ghe90, CG92, CMMS94].

In this standard nameless presentation, the names of bound variables are irrelevant, i.e. whenever a judgement $J$ is provable, so is any $J'$ which is $\alpha$-equivalent to $J$. This fact is usually assumed in the notation. For example, it is often stipulated that any term (or type, or judgement) actually denotes its whole $\alpha$ equivalence class, or that variable names are only a readable version of the corresponding De Bruijn indexes. In this paper, however, we have to deal with renaming explicitly, since the fact that kernel Fun subtype checking can be performed without $\alpha$ renaming is one of our results and is the basis of the application of the memoization technique. It is interesting to note that this is not true for $F_{\leq}$, where some use of $\alpha$ renaming cannot be avoided (if $F_{\leq}$ subtyping were checkable without $\alpha$ renaming, it would be decidable; see [Ghe95]). For this reason, we will give an explicit rule which allows $\alpha$ renaming.

We now give the traditional presentation of kernel Fun, defined by the following set of rules. In this set of rules variable names only stand for themselves; the freedom of $\alpha$ renaming is explicitly stated by rule $(\alpha \leq)$.[3] We do not report the good formation rules for types and environments, which state that, in a well-formed judgement, no type variable is free and no type variable is defined in the scope of another variable with the same name; the scope of the variable $t$ in $\forall t{\leq}T.U$ is $U$, while the scope of $t$ in $\Gamma', t{\leq}T, \Gamma'' \vdash_{\mathcal{G}} U \leq U'$, is $\Gamma'', T, U$ (see [CMMS94]). We also need the following definitions and notations. A pre-judgement $(\Gamma)$ is well formed when $\Gamma \vdash_{\mathcal{G}} \mathcal{G}$-Env; $\langle \Gamma, \{T\} \rangle$ is well formed when $\Gamma \vdash_{\mathcal{G}} T$ Type; $\langle \Gamma, \{T, U\} \rangle$ is well formed when both $\Gamma \vdash_{\mathcal{G}} T$ Type and $\Gamma \vdash_{\mathcal{G}} U$ Type hold. $P_1 =_\alpha P_2$ means that the pre-judgements $P_1$ and $P_2$ are $\alpha$ equivalent and are both well-formed, while $T_1 =_\alpha T_2$ means that $T_1$ and $T_2$ are two $\alpha$ equivalent types. If $\Gamma \equiv t_1{\leq}T_1, \ldots, t_n{\leq}T_n$, then $\mathrm{def}(\Gamma) \equiv \{t_1, \ldots, t_n\}$. If $\Gamma \equiv \Gamma', t \leq T, \Gamma''$ and $t \notin \mathrm{def}(\Gamma'')$, then $\Gamma(t)$ denotes the bound $T$ of $t$. Here are the rules.

[3] kernel Fun is presented here without transitivity, as it has been presented in [CW85]. The equivalence of this presentation with the one with transitivity, and with reflexivity defined on every type, can be proved either by adapting the proof in [CG92], or by a much simpler induction.

$$\frac{\Gamma, t{\leq}T, \Gamma' \vdash_{\mathcal{G}} \mathcal{G}\text{-Env}}{\Gamma, t{\leq}T, \Gamma' \vdash_{\mathcal{G}} t \leq t} \quad (\text{Refl}{\leq})$$

$$\frac{\Gamma \vdash_{\mathcal{G}} T \text{ Type}}{\Gamma \vdash_{\mathcal{G}} T \leq \text{Top}} \quad (\text{Top}{\leq})$$

$$\frac{t \neq U \quad t \in \text{def}(\Gamma) \quad \Gamma \vdash_{\mathcal{G}} \Gamma(t) \leq U}{\Gamma \vdash_{\mathcal{G}} t \leq U} \quad (\text{Var}{\leq})$$

$$\frac{\Gamma \vdash_{\mathcal{G}} T' \leq T \quad \Gamma \vdash_{\mathcal{G}} U \leq U'}{\Gamma \vdash_{\mathcal{G}} T \to U \leq T' \to U'} \quad (\to{\leq})$$

$$\frac{\Gamma, t{\leq}T \vdash_{\mathcal{G}} U \leq U'}{\Gamma \vdash_{\mathcal{G}} \forall t{\leq}T.U \leq \forall t{\leq}T.U'} \quad (\forall{\leq})$$

$$\frac{\Gamma \vdash_{\mathcal{G}} T \leq U \quad \langle \Gamma, \{T, U\}\rangle =_{\alpha} \langle \Gamma', \{T', U'\}\rangle}{\Gamma' \vdash_{\mathcal{G}} T' \leq U'} \quad (\alpha{\leq})$$

Rule $(\forall{\leq})$ is the essential rule which distinguishes kernel Fun from $F_{\leq}$, and which makes kernel Fun subtyping decidable.

The freedom of applying $\alpha$ renaming whenever it is needed makes some rules, namely $(\forall{\leq})$ and $(\text{Var}{\leq})$, easier to write. For example, in this formulation, rule $(\forall{\leq})$ apparently requires that two comparable quantified types have the same bound variable with exactly the same bound. But the rule can also be used to compare types with different bounds and variables, such as: $\forall t{\leq}(\forall u{\leq}\text{Top}.u).t \leq \forall t'{\leq}(\forall u'{\leq}\text{Top}.u').\text{Top}$ by combining it with $\alpha$ renaming. More subtly, without $\alpha$, $(\text{Var}{\leq})$ cannot be used to prove provable judgements "$\Gamma \vdash_{\mathcal{G}} t \leq U$" where $\Gamma(t)$ is not well formed in $\Gamma$, such as: $t{\leq}\forall u{\leq}\text{Top}.u,\ u{\leq}\text{Top} \vdash_{\mathcal{G}} t \leq \forall v{\leq}\text{Top}.v$. This use of the $\alpha$ rule should not be overlooked.

### 2.2 The *UBD* variant

The algorithm we will define in Section 3 can avoid variable renaming thanks to the presence of the "unification environment", but also because it analyzes a specific class of pre-judgements. These we will call "valid pre-judgements", and they enjoy the properties that (a) no name clash is possible in a valid pre-judgement, (b) the backward application of any subtyping rule reduces a valid pre-judgement to another valid pre-judgement[4] and (c) two variables with the same name have the same bound. To make it easier to compare the standard system with the one in the next section, we define here a variant of the standard system which enjoys properties (a), (b), and (c), by first defining what a valid pre-judgements is.

[4] Informally, the backward application of a rule to a judgement is the process of unifying the judgement with the conclusion of the rule, and substituting the judgement with the rule premises, instantiated by the unification.

A valid pre-judgement $(\Gamma)$, $\langle \Gamma, \{T\}\rangle$, $\langle \Gamma, \{T, U\}\rangle$ should be, first of all, well-formed (i.e. no variable should be defined in the scope of a different variable with the same name), but this is not enough, since this property is not preserved by the backward application of rule $(\text{Var}{\leq})$. Consider for example, the following judgement ($\forall t.U$ abbreviates $\forall t{\leq}\text{Top}.U$):

$$t{\leq}\forall u.u,\ u{\leq}\text{Top} \vdash_{\mathcal{G}} t \leq \forall u'.u'$$

becomes, by $(\text{Var}{\leq})$:

$$t{\leq}\forall u.u,\ u{\leq}\text{Top} \vdash_{\mathcal{G}} \forall u.u \leq \forall u'.u'.$$

(Notice that the original judgement is well-formed and provable, but we need the $\alpha$ rule to prove it.) Hence we have to resort to a stronger requirement. We say that a pre-judgement satisfies the *UniDef* property when it is well-formed and it is still well-formed after the repeated substitution of each variable by its bound. This property is strong enough to guarantee against name clashes, and the subtype checker always transforms a *UniDef* pre-judgement into a set of *UniDef* pre-judgements.[5]

It can be proved that the *UniDef* property can be checked in polynomial time. Finally, property (c), which we will call *UniBound*, is defined as follows.

**Definition 2.1** ($\mathcal{B}^*()$) *$\mathcal{B}^*()$ applied to a judgement or to a pre-judgement collects all variable definitions, with their bounds, contained in the judgement or pre-judgement.*

For example,

$$\mathcal{B}^*(t{\leq}\forall u{\leq}\text{Top}.\text{Top} \vdash_{\mathcal{G}} \forall u{\leq}t.u \leq \forall v{\leq}t.v) \equiv \{t{\leq}\forall u{\leq}\text{Top}.\text{Top}, u{\leq}\text{Top}, u{\leq}t, v{\leq}t\}.$$

**Definition 2.2** (UniBound()) *A pre-judgement belongs to* UniBound *iff whenever two different variables have the same name, they have the same bound.*

$$\text{UniBound}(P) \Leftrightarrow (\{t{\leq}T_1, t{\leq}T_2\} \subseteq \mathcal{B}^*(P) \Rightarrow T_1 \equiv T_2)$$

Notice that the *UniBound* and *UniDef* properties do not imply each other, and their combination gives us our validity condition.

**Definition 2.3** (*UBD*) *A pre-judgement $P$ satisfies* UBD *iff* UniDef($P$) *and* UniBound($P$) *both hold.*

*A judgement is provable in the* UBD *system iff it is provable in the nameless system by a proof where every judgement satisfies* UBD.

[5] The simpler and stronger property that no two variables in the same judgement have the same name, regardless of their scope, would guarantee against name clashes, but is not preserved by $(\text{Var}{\leq})$, which would transform, e g., $t{\leq}\forall u.u \vdash_{\mathcal{G}} t \leq U$ into $t{\leq}\forall u.u \vdash_{\mathcal{G}} \forall u.u \leq U$, which only satisfies the *UniDef* property.

We now prove that no subtyping is lost in the $\vdash_{\mathcal{G}}$ system if only *UBD* judgements are allowed, i.e. that if a judgement $J$ is $\alpha$-equivalent to a *UBD* judgement $J'$, then $J$ is provable in the nameless system iff $J'$ is provable in the *UBD* system.

**Theorem 2.4** *Let* $\vdash_{\mathcal{G}}^{UBD}$ *mean (just for this theorem) "provable in a system where only* UBD *judgements are well-formed"; then:*

$$\langle \Gamma, \{T, U\}\rangle =_\alpha \langle \Gamma', \{T', U'\}\rangle,\ \mathrm{UBD}(\Gamma', T', U') \Rightarrow (\Gamma \vdash_{\mathcal{G}} T \leq U \Leftrightarrow \Gamma' \vdash_{\mathcal{G}}^{UBD} T' \leq U').$$

**Proof Outline** *The ($\Leftarrow$) part is trivial. The ($\Rightarrow$) part can be proved by induction on the size of the proof of* $\Gamma \vdash_{\mathcal{G}} T \leq U$, *and by cases on the last applied rule. Cases* (Top$\leq$), ($\rightarrow\leq$) *and* (Var$\leq$) *are solved by showing that, if* $\langle \Gamma, \{T, U\}\rangle =_\alpha \langle \Gamma', \{T', U'\}\rangle$, *then the premises of the last rule applied to prove* $\Gamma \vdash_{\mathcal{G}} T \leq U$ *are $\alpha$ equivalent to* UBD *judgements which can be used to prove* $\Gamma \vdash_{\mathcal{G}} T \leq U$. *Case* ($\forall\leq$) *is different, since in this case $T$ and $U$ are two universal types with the same variable name and bound, while this may not be true for $T'$ and $U'$. Hence, we first inductively prove an* UBD *judgement* $\langle \Gamma'', \{T'', U''\}\rangle$ *which is $\alpha$ equivalent to* $\langle \Gamma, \{T, U\}\rangle$ *and which has equal variable names and bounds, and then prove* $\Gamma' \vdash_{\mathcal{G}}^{UBD} T' \leq U'$ *by applying* ($\alpha\leq$), *in the* UBD *system, to* $\Gamma'' \vdash_{\mathcal{G}} T'' \leq U''$. *In case* ($\alpha\leq$), *if* $\langle \Gamma, \{T, U\}\rangle$ *has been proved by* ($\alpha\leq$) *from* $\langle \Gamma'', \{T'', U''\}\rangle$, *then provability of* $\Gamma' \vdash_{\mathcal{G}}^{UBD} T' \leq U'$ *follows by induction* with no need to use ($\alpha\leq$) in the *UBD* system. □

The proof above shows that, in the *UBD* system, the $\alpha$ rule can be eliminated by substituting rule ($\forall\leq$) with the following one:

$$\frac{\Gamma \vdash_{\mathcal{G}} \forall t'{\leq}T'.U' \text{ Type} \quad T =_\alpha T' \qquad \Gamma, t{\leq}T \vdash_{\mathcal{G}} U \leq [t{=}t'](U')}{\Gamma \vdash_{\mathcal{G}} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'} \quad (\forall{\leq}\text{-}\alpha)$$

Without the *UBD* restriction, we claim that it would also be necessary to substitute rule (Var$\leq$) as follows, where the empty substitution $[\,](\Gamma(t))$ renames any variable which is defined inside $\Gamma(t)$.

$$\frac{t \neq U \quad t \in \mathrm{def}(\Gamma) \quad \Gamma \vdash_{\mathcal{G}} [\,](\Gamma(t)) \leq U}{\Gamma \vdash_{\mathcal{G}} t \leq U} \quad (\mathrm{Var}{\leq}\text{-}\alpha)$$

Hence, the *UBD* restriction can be seen as a technique to avoid $\alpha$ renaming when the (Var$\leq$) rule is applied. In the next section we will introduce a technique to avoid $\alpha$ renaming when ($\forall\leq$) is applied.

From now on we will forget about the standard nameless system and will only consider its *UBD* restriction.

## 3 The $\alpha$-free subtype checking algorithm

### 3.1 The system

In this section we introduce what we call "the $\alpha$-free subtype checking algorithm", where "subtype checking" distinguishes it from the "constraint generating" one, which will be presented in the next section.

This algorithm is characterized by three main features, the fact that it is only defined on *UBD* judgements, the fact that it has a priori knowledge of the bounds of all the variables, and the presence of the "unification environment". These features are related to the behaviour of some real subtype checking algorithms, and they are also needed to obtain the property that the algorithm never creates nor renames types during subtype checking, which is essential for the polynomial variant we will describe later. We now discuss informally how these features are related to real subtype checkers and to the non-creation property, and how they are captured in our formalization.

- *UBD* judgements: the input for a subtype checker is generally produced by a parser which allocates a new data structure for every variable definition, and represents the instances of that variable by a pointer to that data structure. Hence, no two different variables may have the same name (i.e. the same memory address). When the subtype checker applies rule (Var$\leq$), it may break this strong property (see footnote 5), but it maintains the weaker *UBD* invariant. Hence, the assumption that the judgement we have to check satisfies *UBD* is usually satisfied by a real type checker, and, moreover, any judgement may be transformed, if needed, into an $\alpha$ equivalent *UBD* judgement in polynomial time, by giving fresh names to all its variables; hence the *UBD* assumption does not restrict the validity of our analysis. We impose the *UBD* condition since the *UniDef* property is essential to eliminate the use of the $\alpha$ rule which is related to the (Var$\leq$) rule (see Section 2.2 and Theorem 3.4); the *UniBound* condition allows us to describe the global bounds environment $B$, described below, simply as a mapping from variable names to types.
- A priori knowledge of the variable bounds: in a real type checker a variable is typically implemented by a structure which contains a pointer to its bound, and this connection is often estabilished when the parse tree is built, i.e. before the subtype-checker starts. Moreover, our subtype checker never creates new varibles nor renames their bounds, hence this association never changes during type check-

ing. This is formalized by substituting the stack-like environment $\Gamma$ with a "bounds environment" $B$ which is not changed by the subtyping rules and which contains, right from the beginning, the bounds of all the type variables which are involved in the judgement being checked (this fact is enforced by rules (Var$\leq$) and ($\forall\leq$)).

- Unification environments: in order to avoid $\alpha$ substitutions, when two types $\forall t{\leq}T.U$ and $\forall t'{\leq}T'.U'$ have to be compared, the algorithm compares $U$ and $U'$ in a "unification environment" $t{=}t'$ which tells it to consider $t$ and $t'$ to be equal. This allows it to avoid the renaming $[t = t'](U')$, which is performed by the standard algorithm. *To our knowledge*, this technique is only used in the subtype checker of the database language Fibonacci.

We can now give the syntax of the subtyping judgements which describe this algorithm. The $\mathcal{E}$ subscript under the $\vdash$ symbols distinguishes these rules from the other systems in the paper, while the $B$ superscript is a rule metavariable, which we put in this de-emphasizing position because it is, as we explained above, just a read-only variable.

$$\begin{array}{lcl} T & ::= & t \mid T \to T \mid \forall t{\leq}T.T \mid \mathrm{Top} \\ B & ::= & () \mid B, t{\leq}T \\ E & ::= & () \mid E, t{=}t \\ X & ::= & () \mid X, t \\ J & ::= & B \vdash_{\mathcal{B}} \mathcal{B}\text{-Env} \mid E \vdash_{\mathcal{E}}^{B} \mathcal{E}\text{-Env} \\ & & \mid X \vdash_{\mathcal{E}}^{B} \mathcal{X}\text{-Env} \\ & & \mid X \vdash_{\mathcal{E}}^{B} T\ \mathrm{Type} \mid E \vdash_{\mathcal{E}}^{B} T \leq U \end{array}$$

A pre-judgement is well-formed only if it satisfies the *UBD* property. All the rules trivially preserve the *UniDef* invariant, while the good formation and *UniBound* properties are explicitly enforced by the hypothesis of the rules when it is needed. We remark that the ($\forall\leq$) rule of system $F_{\leq}$ does *not* preserve the *UniDef* invariant, and this is the technical reason which prevents us from making a trivial extension of the $\vdash_{\mathcal{E}}^{B}$ rules to system $F_{\leq}$. We now give the complete set of rules we consider for kernel Fun.

**Notation 3.1** ($\mathrm{def}(B), \mathrm{left}(E), \mathrm{right}(E), \mathrm{swap}(E)$)

$$\begin{array}{rcl} \mathrm{def}(t_1{\leq}T_1, \ldots, t_n{\leq}T_n) & =_{\mathrm{def}} & t_1, \ldots, t_n \\ \mathrm{left}(t_1{=}u_1, \ldots, t_n{=}u_n) & =_{\mathrm{def}} & t_1, \ldots, t_n \\ \mathrm{right}(t_1{=}u_1, \ldots, t_n{=}u_n) & =_{\mathrm{def}} & u_1, \ldots, u_n \\ \mathrm{swap}(t_1{=}u_1, \ldots, t_n{=}u_n) & =_{\mathrm{def}} & u_1{=}t_1, \ldots, u_n{=}t_n \end{array}$$

**Notation 3.2** ($B(t)$) *$B(t)$ is defined as $T$, when $B \equiv \ldots, t{\leq}T, \ldots$.*

**Notation 3.3** ($T = U$) *$E \vdash_{\mathcal{E}}^{B} T = U$ stands for $E \vdash_{\mathcal{E}}^{B} T \leq U \wedge \mathrm{swap}(E) \vdash_{\mathcal{E}}^{B} U \leq T$.*

Subtyping rules:

$$\frac{E, t{=}u, E' \vdash_{\mathcal{E}}^{B} \mathcal{E}\text{-Env}}{E, t{=}u, E' \vdash_{\mathcal{E}}^{B} t \leq u} \quad (\mathrm{Refl}\leq)$$

$$\frac{E \vdash_{\mathcal{E}}^{B} \mathcal{E}\text{-Env} \quad \mathrm{left}(E) \vdash_{\mathcal{E}}^{B} T\ \mathrm{Type}}{E \vdash_{\mathcal{E}}^{B} T \leq \mathrm{Top}} \quad (\mathrm{Top}\leq)$$

$$\frac{(t{=}U) \notin E \quad t \in \mathrm{def}(B) \quad t \in \mathrm{left}(E) \quad E \vdash_{\mathcal{E}}^{B} B(t) \leq U}{E \vdash_{\mathcal{E}}^{B} t \leq U} \quad (\mathrm{Var}\leq)$$

$$\frac{\mathrm{swap}(E) \vdash_{\mathcal{E}}^{B} T' \leq T \quad E \vdash_{\mathcal{E}}^{B} U \leq U'}{E \vdash_{\mathcal{E}}^{B} T \to U \leq T' \to U'} \quad (\to\leq)$$

$$\frac{\{t{\leq}T, t'{\leq}T'\} \subseteq B \quad E \vdash_{\mathcal{E}}^{B} T = T' \quad E, t{=}t' \vdash_{\mathcal{E}}^{B} U \leq U'}{E \vdash_{\mathcal{E}}^{B} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'} \quad (\forall\leq)$$

Observe that the conditions with shape: $E \vdash_{\mathcal{E}}^{B} \mathcal{E}\text{-Env}$, $X \vdash_{\mathcal{E}}^{B} T\ \mathrm{Type}$, $t \in X$, which are found in the subtyping rules are not actually checked by a real subtype checker, since they are guaranteed by the tool which builds the subtype checker input, but they are still needed in our formal presentation to formalize which properties must be guaranteed by that input provider.

### 3.2 Equivalence of the nameless and the type checking systems

We want now to prove that the two systems are equivalent, which may be formulated as follows (recall that $\vdash_{\mathcal{G}}$ refers here to the *UBD* variant):

$$\begin{array}{l} \Gamma \vdash_{\mathcal{G}} T \leq U \Rightarrow E_{\Gamma} \vdash_{\mathcal{E}}^{\mathcal{B}^*(\Gamma,T,U)} T \leq U \\ E \vdash_{\mathcal{E}}^{B} T \leq U \Rightarrow \Gamma_{E,B} \vdash_{\mathcal{G}} T \leq [E](U) \end{array}$$

To make this statement precise we will define what $E_{\Gamma}$ and $\Gamma_{E,B}$ are (Theorems 3.7, 3.10).

To prove that the two systems are equivalent, we need the standard subproof, weakening, and strengthening lemmas, plus the crucial lemma which states that, if any judgement $J$ is provable in the $E$ system, than any $\alpha$ variant $J'$ of $J$ is provable.

**Lemma 3.4** *The E-system is $\alpha$ invariant:*

$$\begin{array}{l} \langle B, E, \{T, U\}\rangle =_{\alpha} \langle B_1, E_1, \{T_1, U_1\}\rangle \\ \Rightarrow\ (E \vdash_{\mathcal{E}}^{B} T \leq U \Leftrightarrow E_1 \vdash_{\mathcal{E}}^{B_1} T_1 \leq U_1). \end{array}$$

Before proving that $\Gamma$-subtyping implies $E$-subtyping, we prove the same thing for good formations.

**Notation 3.5** ($\mathrm{def}^2(\Gamma)$) *If $\mathrm{def}(\Gamma) \equiv (t_1, \ldots, t_n)$ then $\mathrm{def}^2(\Gamma) \equiv (t_1{=}t_1, \ldots t_n{=}t_n)$.*

**Lemma 3.6** ($\Gamma$ **good formation** $\Rightarrow$ $E$ **good form.**)

1. $\Gamma \vdash_{\mathcal{G}} \mathcal{G}\text{-Env},\ B \vdash_{\mathcal{E}} \mathcal{B}\text{-Env},\ B \supseteq \mathcal{B}^*(\Gamma)$
   $\Rightarrow \mathrm{def}(\Gamma) \vdash^B_{\mathcal{E}} \mathcal{X}\text{-Env},\ \mathrm{def}^2(\Gamma) \vdash^B_{\mathcal{E}} \mathcal{E}\text{-Env}$
2. $\Gamma \vdash_{\mathcal{G}} T\ \mathrm{Type},\ B \vdash_{\mathcal{E}} \mathcal{B}\text{-Env},\ B \supseteq \mathcal{B}^*(\Gamma, T)$
   $\Rightarrow \mathrm{def}(\Gamma) \vdash^B_{\mathcal{E}} T\ \mathrm{Type}$

**Theorem 3.7** ($\Gamma$ **subtyping implies** $E$ **subtyping**)

$$\Gamma \vdash_{\mathcal{G}} T \leq U,\ B \vdash_{\mathcal{E}} \mathcal{B}\text{-Env},\ B \supseteq \mathcal{B}^*(\Gamma, T, U) \Rightarrow \mathrm{def}^2(\Gamma) \vdash^B_{\mathcal{E}} T \leq U$$

**Proof Outline** *By induction on the proof of* $\Gamma \vdash_{\mathcal{G}} T \leq U$ *and by cases on the last applied rule. Some care is needed when the last rule is* $(\alpha \leq)$*; in this case,* $\vdash^B_{\mathcal{E}}$ *provability follows by Lemma 3.4. We also heavily exploit Lemma 3.6.* □

The next definition specifies how a variable environment $X$ can be upgraded to an environment $\Gamma$ by associating each variable in $X$ with its bound in a specified bounds environment $B$.

**Definition 3.8** $(B \cap X)$ $\quad B \cap (t_1, \ldots, t_n) =_{\mathrm{def}} t_1 \leq B(t_1), \ldots, t_n \leq B(t_n)$

**Definition 3.9** *We say that an environment* $E$ *is strongly well-formed in* $B$ *if, whenever two variables are unified by* $E$*, their bounds in* $B$ *are unified too, in the following sense:*[6]

$E$ *is strongly well formed in* $B$ *iff*
$E \vdash^B_{\mathcal{E}} \mathcal{E}\text{-Env}, \forall t{=}t' \in E.\ (B(t) =_\alpha [E](B(t')),\ [\mathrm{swap}(E)](B(t)) =_\alpha B(t'))$

The next theorem completes the proof of the equivalence between the named and the nameless system.

**Theorem 3.10** ($E$ **provability** $\Rightarrow$ $\Gamma$ **provability**) *If* $E$ *is strongly well formed in* $B$*, then:*

$$X \vdash^B_{\mathcal{E}} \mathcal{X}\text{-Env} \Rightarrow B \cap X \vdash_{\mathcal{G}} \mathcal{G}\text{-Env} \quad (1)$$

$$E \vdash^B_{\mathcal{E}} \mathcal{E}\text{-Env} \Rightarrow B \cap \mathrm{left}(E) \vdash_{\mathcal{G}} \mathcal{G}\text{-Env},\ B \cap \mathrm{right}(E) \vdash_{\mathcal{G}} \mathcal{G}\text{-Env} \quad (2)$$

$$B \cap X \vdash_{\mathcal{G}} \mathcal{G}\text{-Env}, X \vdash^B_{\mathcal{E}} T\ \mathrm{Type} \Rightarrow B \cap X \vdash_{\mathcal{G}} T\ \mathrm{Type} \quad (3)$$

$$E \vdash^B_{\mathcal{E}} T \leq U \Rightarrow B \cap \mathrm{left}(E) \vdash_{\mathcal{G}} T \leq [E](U), \quad (4l)$$

$$B \cap \mathrm{right}(E) \vdash_{\mathcal{G}} [\mathrm{swap}(E)](T) \leq U \quad (4r)$$

[6] $B(t) =_\alpha [E](B(t'))$ is actually equivalent to $[\mathrm{swap}(E)](B(t)) =_\alpha B(t')$, but this redundant formulation relieves us from having to prove this fact.

**Proof Outline** *(2) is a corollary of (1), and (3) is standard. (1) and (4) are proved together by induction on the size of the proof of the judgement, and by cases on the last applied rule. In case* (Top$\leq$) *of (4r), some care is needed to prove that* $\mathrm{left}(E) \vdash^B_{\mathcal{E}} T\ \mathrm{Type}$ *implies* $B \cap \mathrm{right}(E) \vdash_{\mathcal{G}} [\mathrm{swap}(E)](T)\ \mathrm{Type}$*, needed to prove that* $B \cap \mathrm{right}(E) \vdash_{\mathcal{G}} [\mathrm{swap}(E)](T) \leq \mathrm{Top}$*. In case* (Var$\leq$) *of (4r), the strong good formation hypothesis and the* $(\alpha \leq)$ *rule are needed to prove that* $t{=}t' \vdash^B_{\mathcal{E}} t \leq U$ *implies* $B \cap t' \vdash_{\mathcal{G}} t' \leq U$*. In case* $(\forall \leq)$*,* $E \vdash^B_{\mathcal{E}} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'$ *is proved from* $E \vdash^B_{\mathcal{E}} T = T'$*,* $E, t{=}t' \vdash^B_{\mathcal{E}} U \leq U'$*. We first exploit an irreflexivity lemma to show that* $E \vdash^B_{\mathcal{E}} T = T'$ *(where* $=$ *means* $\leq \wedge \geq$*) inductively implies* $T =_\alpha [E](T')$*,* $[\mathrm{swap}(E)](T) =_\alpha T'$*. This implies that* $E, t{=}t' \vdash^B_{\mathcal{E}} U \leq U'$ *is strongly well formed, hence we can prove by induction that* $B \cap (\mathrm{left}(E), t) \vdash_{\mathcal{G}} U \leq [E, t{=}t'](U')$*, which implies, by* $(\forall \leq)$*,* $B \cap \mathrm{left}(E) \vdash_{\mathcal{G}} \forall t{\leq}T.U \leq \forall t{\leq}T.[E, t{=}t'](U')$*. The thesis follows by rule* $(\alpha \leq)$*.* □

## 4 The constraint based rules

### 4.1 The system

In this section we describe a different approach to the subtype-checking process. Instead of providing the environments $B, E$ and the compared types $T, U$ as input, obtaining only success or failure, we provide $B, T, U$ alone, and obtain as an answer a set of constraints $\Sigma \equiv t_1 \leq u_1, \ldots, t_n \leq u_n$ which specifies the minimal subtyping relation between variables which should be implied by $E$ to make $E \vdash^B_{\mathcal{E}} T \leq U$ true. Thus, as we will detail later, a single problem can only generate a polynomial amount of different subproblems. Hence, by combining the algorithm described in this section with a memoization technique, we can finally obtain a polynomial algorithm for kernel Fun subtype checking. Formally, we define this constraint generating algorithm by a $\vdash^B_{\mathcal{S}}$ entailment relation, with the following syntax. The actual algorithm works by backward applications of these rules; the $\Sigma$ constraint set in the subtyping judgement is the only output variable.

$$\begin{array}{lcl} B & ::= & () \mid B, t{\leq}T \\ X & ::= & () \mid X, t \\ \Sigma & ::= & () \mid \Sigma, t{\leq}t \mid \Sigma, t{\geq}t \\ J & ::= & B \vdash_{\mathcal{E}} \mathcal{B}\text{-Env} \mid X \vdash^B_{\mathcal{E}} \mathcal{X}\text{-Env} \mid \Sigma \vdash^B_{\mathcal{S}} \mathcal{S}\text{-Env} \\ & & \mid X \vdash^B_{\mathcal{E}} T\ \mathrm{Type} \mid \Sigma \vdash^B_{\mathcal{S}} T \leq T \end{array}$$

Notice that the good formation judgements which do not need to be modified are borrowed from the $\alpha$-free subtype checking system.

**Claim 4.1 (Theorems 4.7, 4.8)** $\vdash^B_{\mathcal{E}}$ *and* $\vdash^B_{\mathcal{S}}$ *are equiv-*

*alent; formally, if $T$ and $U$ are well formed in $B$, then:*

$$\Sigma \vdash^B_{\mathcal{S}} T \leq U \Rightarrow (\forall E.\ E \vdash^B_{\mathcal{E}} \Sigma \Rightarrow E \vdash^B_{\mathcal{S}} T \leq U)$$
$$E \vdash^B_{\mathcal{E}} T \leq U \Rightarrow (\exists \Sigma.\ E \vdash^B_{\mathcal{E}} \Sigma \wedge \Sigma \vdash^B_{\mathcal{E}} T \leq U)$$
*Hence:* $\vdash^B_{\mathcal{E}} T \leq U$
$$\Leftrightarrow (\vdash^B_{\mathcal{S}} T \leq U,\ \vdash^B_{\mathcal{E}} T \text{ Type},\ \vdash^B_{\mathcal{E}} U \text{ Type})$$

The $\vdash^B_{\mathcal{S}}$ relation is defined below; as usual, the crucial rule is $(\forall\leq)$; the $-_B$ operator is defined later.

**Notation 4.2** (swap($\Sigma$)) $\text{swap}(t_1 \leq u_1, \ldots, t_n \geq u_n) =_{\text{def}} u_1 \geq t_1, \ldots, u_n \leq t_n$

$$\frac{t \leq u \vdash^B_{\mathcal{S}} \mathcal{S}\text{-Env}}{t \leq u \vdash^B_{\mathcal{S}} t \leq u} \quad (\text{Atom}\leq)$$

$$\frac{\text{notVar}(U) \quad t \in \text{def}(B) \quad \Sigma \vdash^B_{\mathcal{S}} B(t) \leq U}{\Sigma \vdash^B_{\mathcal{S}} t \leq U} \quad (\text{Var}\leq)$$

$$\frac{B \vdash_{\mathcal{E}} \mathcal{B}\text{-Env}}{\vdash^B_{\mathcal{S}} T \leq \text{Top}} \quad (\text{Top}\leq)$$

$$\frac{\Sigma \vdash^B_{\mathcal{S}} T' \leq T \quad \Sigma' \vdash^B_{\mathcal{S}} U \leq U'}{\text{swap}(\Sigma), \Sigma' \vdash^B_{\mathcal{S}} T \to U \leq T' \to U'} \quad (\to\leq)$$

$$\frac{\begin{array}{c}\Sigma \vdash^B_{\mathcal{S}} T' \leq T \quad \Sigma' \vdash^B_{\mathcal{S}} T \leq T' \\ \Sigma'' \vdash^B_{\mathcal{S}} U \leq U' \\ \Sigma'' -_B (t{=}t') \neq \bot \quad \{t{\leq}T, t'{\leq}T'\} \subseteq B\end{array}}{\begin{array}{c}\text{swap}(\Sigma), \Sigma', (\Sigma'' -_B (t{=}t')) \\ \vdash^B_{\mathcal{S}} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'\end{array}} \quad (\forall\leq)$$

notVar($U$) in rule (Var$\leq$) means that $U$ is not a type variable. We make the following observations.

(Atom$\leq$) is the rule which generates the constraints, which are then transmitted by the other rules and eliminated by rule $(\forall\leq)$.

Rule $(\forall\leq)$ analyses $U \leq U'$ without considering the $t{=}t'$ unification; then, once the corresponding constraints $\Sigma''$ have been collected, it removes those which are implied by $t = t'$. The operation $(\Sigma -_B (t = t'))$ gives, informally, the constraints which remain after one knows that $t{=}t'$, and $\Sigma -_B (t{=}t') = \bot$ means that, if $t$ is equated in the environment to $t'$, then $\Sigma$ cannot hold.

Thanks to rule (Top$\leq$), it is possible that $\Sigma \vdash^B_{\mathcal{S}} T \leq U$, even if $T$ or $U$ are not well formed, hence the subproof property $\Sigma \vdash^B_{\mathcal{S}} T \leq U \Rightarrow \text{def}(\Sigma) \vdash^B_{\mathcal{E}} T$ Type is not valid.

Using linear logic jargon, note that the additive behavior of environments in $E$-rules (same context in the consequence and in the premises) and their multiplicative behavior in $\Sigma$-rules (the context in the consequence is a combination of the contexts in the premises) shows that environments are input variables in the first case, and output variables in the second case.

We now define the operation $\Sigma -_B (t = t')$, first on a single constraint $\sigma$, and then on a constraint list $\Sigma$.

**Definition 4.3** $(\Sigma -_B (t{=}t'))$ *Single constraint* $(\sigma -_B (t{=}t'))$*:*

$$(t{\leq}t') -_B (t{=}t') = () \quad (1)$$
$$v' \neq t',\ t{\leq}u \in B \Rightarrow (t{\leq}v') -_B (t{=}t') = u{\leq}v' \quad (2)$$
$$v' \neq t', t{\leq}T \in B,\ \text{notVar}(T) \Rightarrow (t{\leq}v') -_B (t{=}t') = \bot \quad (3)$$
$$v \neq t \Rightarrow (v{\leq}t') -_B (t{=}t') = \bot \quad (4)$$
$$v \neq t, v' \neq t' \Rightarrow (v{\leq}v') -_B (t{=}t') = v{\leq}v' \quad (5)$$
$$(v{\geq}v') -_B (t{=}t') = \text{swap}((v'{\leq}v) -_B (t'{=}t)) \quad (6)$$

*Constraint list:*

$$() -_B (t{=}t') = ()$$

$$(\Sigma, \sigma) -_B (t{=}t') = \begin{cases} \bot & \text{if } \Sigma -_B (t{=}t') = \bot \text{ or } \sigma -_B (t{=}t') = \bot \\ \Sigma -_B (t{=}t'), \sigma -_B (t{=}t') & \text{otherwise} \end{cases}$$

Note that, while the order of pairs inside an equality environment is essential, the order of the constraints in a constraint list is irrelevant. This set of rules defines a terminating algorithm.

**Lemma 4.4** *The algorithm defined by the rules we have given always terminates on a judgement* $\Sigma \vdash^B_{\mathcal{S}} T \leq U$ *such that (a)* $T$ *and* $U$ *are ground terms and (b) there exists* $E$ *such that* $E \vdash^B_{\mathcal{E}} \mathcal{E}$-Env, left($E$) $\vdash^B_{\mathcal{E}} T$ Type, right($E$) $\vdash^B_{\mathcal{E}} U$ Type.

**Proof Outline** *See [Ghe90, KS92, Ghe93a].* □

### 4.2 Equivalence with the subtype checking system

To prove the equivalence between the two systems, we first need a lemma which specifies that the $\Sigma -_B (t{=}t')$ operation behaves as expected.

**Definition 4.5** $(E \vdash^B_{\mathcal{E}} \Sigma \text{ Type}, E \vdash^B_{\mathcal{E}} \Sigma)$ $E \vdash^B_{\mathcal{E}} \Sigma$ Type *means that* left($E$) $\vdash^B_{\mathcal{E}} t$ Type *for each* $t$ *in* left($\Sigma$)*, and the same for* right($E$), right($\Sigma$). $E \vdash^B_{\mathcal{E}} \Sigma$ *means that* $E \vdash^B_{\mathcal{E}} \mathcal{E}$-Env *and that* $E \vdash^B_{\mathcal{E}} t \leq u$ *for each* $t{\leq}u$ *pair in* $\Sigma$*. Note that it is never the case that* $E \vdash^B_{\mathcal{E}} \bot$.

**Lemma 4.6** $E, t{=}t' \vdash^B_{\mathcal{E}} \Sigma$
$$\Leftrightarrow (E, t{=}t' \vdash^B_{\mathcal{E}} \Sigma \text{ Type},\ E \vdash^B_{\mathcal{E}} \Sigma -_B (t{=}t'))$$

We can now prove the equivalence between the two systems.

**Theorem 4.7** ($\Sigma$ subtyping implies $E$ subtyping)

$$\Sigma \vdash^B_{\mathcal{S}} T \leq U, . E \vdash^B_{\mathcal{E}} \Sigma,$$
$$\text{left}(E) \vdash^B_{\mathcal{E}} T \text{ Type}, \ \text{right}(E) \vdash^B_{\mathcal{E}} U \text{ Type}$$
$$\Rightarrow E \vdash^B_{\mathcal{E}} T \leq U$$

**Proof Outline** *We prove the thesis by induction on the size of the proof of* $\Sigma \vdash^B_{\mathcal{S}} T \leq U$ *and by cases on the last applied rule. The interesting case is case* $(\forall\leq)$, *where, from* $\text{swap}(\Sigma), \Sigma', (\Sigma'' -_B (t{=}t')) \vdash^B_{\mathcal{S}} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'$, $E \vdash^B_{\mathcal{E}} \text{swap}(\Sigma), \Sigma', (\Sigma'' -_B (t = t'))$, *and* $\Sigma \vdash^B_{\mathcal{S}} T' \leq T$, $\Sigma' \vdash^B_{\mathcal{S}} T \leq T'$, $\Sigma'' \vdash^B_{\mathcal{S}} U \leq U'$ *we must deduce that* $E \vdash^B_{\mathcal{E}} T = T'$, *(immediate) and that* $E, t = t' \vdash^B_{\mathcal{E}} U \leq U'$. *To this aim, we need to prove that* $E, t = t' \vdash^B_{\mathcal{E}} \Sigma''$, *which follows, by Lemma 4.6, from* $E \vdash^B_{\mathcal{E}} \Sigma'' -_B (t = t')$, *once we have proved that* $E, t{=}t' \vdash^B_{\mathcal{E}} \Sigma''$ Type, *which is long but standard.* □

**Theorem 4.8** ($E$ **subtyping implies** $\Sigma$ **subtyping**)

$$E \vdash^B_{\mathcal{E}} T \leq U \Rightarrow (\exists\Sigma.\ E \vdash^B_{\mathcal{E}} \Sigma,\ \Sigma \vdash^B_{\mathcal{S}} T \leq U)$$

**Proof Outline** *We prove it by induction on the size of the proof of* $E \vdash^B_{\mathcal{E}} T \leq U$ *and by cases on the last applied rule. The interesting case is case* $(\forall\leq)$. *Suppose that* $E \vdash^B_{\mathcal{E}} \forall t{\leq}T.U \leq \forall t'{\leq}T'.U'$ *has been proved starting from* $E \vdash^B_{\mathcal{E}} T = T'$, $E, t{=}t' \vdash^B_{\mathcal{E}} U \leq U'$. *By induction, there exist* $\Sigma, \Sigma', \Sigma''$ *such that* $\text{swap}(E) \vdash^B_{\mathcal{E}} \Sigma$, $E \vdash^B_{\mathcal{E}} \Sigma'$, $E, t{=}t' \vdash^B_{\mathcal{E}} \Sigma''$, *and* $\Sigma \vdash^B_{\mathcal{S}} T' \leq T$, $\Sigma' \vdash^B_{\mathcal{S}} T \leq T'$, $\Sigma'' \vdash^B_{\mathcal{S}} U \leq U'$. *By Lemma 4.6,* $E \vdash^B_{\mathcal{E}} \text{swap}(\Sigma), \Sigma', (\Sigma'' -_B (t{=}t'))$. *The thesis follows by applying rule* $(\forall\leq)$. □

## 5 The polynomial algorithm

### 5.1 The standard algorithm is exponential

Before presenting our polynomial algorithm, we show that the standard subtype checking algorithm has an exponential behaviour, and that, moreover, it may reduce a problem to an exponential number of *different* subproblems. To this aim, we define a family of environments $B_i$ and of judgments $J^n_i$ such that the size of $J^n_n$ only grows polynomially with $n$, while the number of different subproblems it generates grows exponentially (in this section we will write $\forall t.U$ as an abbreviation for $\forall t{\leq}\text{Top}.U$).

**Definition 5.1**

| | | |
|---|---|---|
| $T_0$ | = | $t_0$ |
| $T_{i+1}$ | = | $t_{i+1} \to \forall x_{i+1}.T_i$ |
| $U_i$ | = | *the same as* $T_i$, *substituting* $u_j$ *to* $t_j$ *and* $y_j$ *to* $x_j$ |
| $B_1$ | = | $t_0{\leq}\text{Top}$, $u_0{\leq}\text{Top}$, $t_1{\leq}T_0$, $u_1{\leq}U_0$ |
| $B_{i+2}$ | = | $B_{i+1}$, $t_{i+2}{\leq}T_i \to \forall x_{i+1}.t_{i+1}$, $u_{i+2}{\leq}U_i \to \forall y_{i+1}.u_{i+1}$ |
| $E_i$ | = | $u_0 = t_0, \ldots, u_i = t_i$ |
| $J^T_n(E, 0)$ | = | $E \vdash^{B_n}_{\mathcal{E}} U_0 \leq T_0$ |
| $J^T_n(E, i+1)$ | = | $E \vdash^{B_n}_{\mathcal{E}} U_i \to \forall y_{i+1}.u_{i+1} \leq t_{i+1} \to \forall x_{i+1}.T_i$ |
| $J^U_n(E, i)$ | = | *the same as* $J^T_n(E, i)$, *swapping* $u_j$ *with* $t_j$ *and* $y_j$ *with* $x_j$ |

Let us say that $Sub(J)$ is the number of different subproblems, belonging to the $\{J^U_n(E,i)\}_{n,i}$ or to the $\{J^T_n(E,i)\}_{n,i}$ families, generated by the standard algorithm while it is checking judgement $J$. We now show by induction that, for each $i{\leq}n$, $Sub(J^T_n(E_n, i))$ contains $2^i$ different judgements. In the base case, $Sub(J^T_n(E_n, 1))$ contains 2 judgements, $J^U_n(\text{swap}(E_n), 0)$ and $J^T_n((E_n, y_1 = x_1), 0)$ (here $\triangleright$ means "is reduced by the algorithm to"):

$$J^T_n(E_n, 1) \equiv E_n \vdash^{B_n}_{\mathcal{E}} U_0 \to \forall y_1.u_1 \leq t_1 \to \forall x_1.T_0$$
$$\triangleright \text{ By } (\to\leq): \{ \text{swap}(E_n) \vdash^{B_n}_{\mathcal{E}} t_1 \leq U_0, \quad E_n \vdash^{B_n}_{\mathcal{E}} \forall y_1.u_1 \leq \forall x_1.T_0 \}$$
$$\triangleright \{ \text{ by } (\text{Var}\leq): \text{swap}(E_n) \vdash^{B_n}_{\mathcal{E}} T_0 \leq U_0, \quad \text{by } (\forall\leq): E_n, y_1 = x_1 \vdash^{B_n}_{\mathcal{E}} u_1 \leq T_0 \}$$
$$\triangleright \{ \text{swap}(E_n) \vdash^{B_n}_{\mathcal{E}} T_0 \leq U_0, \quad \text{by } (\text{Var}\leq): E_n, y_1 = x_1 \vdash^{B_n}_{\mathcal{E}} U_0 \leq T_0 \}$$

In the same way we can show that $Sub(J^T_n(E_n, i+2))$ is equal to $Sub(J^U_n(\text{swap}(E_n), i{+}1)) \cup Sub(J^T_n((E_n, y_{i+2} = x_{i+2}), i+1))$. To complete the proof we now show that the two sets are disjoint, by observing that no judgement generated (by the standard algorithm) from $J^U_n(\text{swap}(E_n), i+1)$ may contain $y_{i+2} = x_{i+2}$ or $x_{i+2} = y_{i+2}$ in the unification environment, while all judgements generated from $J^T_n((E_n, y_{i+2} = x_{i+2}), i+1)$ contain one of those two pairs. This completes the proof that the size of $Sub(J^T_n(E_n, n))$ is $2^n$.

Observe now that the input size, i.e. the size of a judgement $J^T_n(\text{swap}(E_n), n)$ which generates $2^n$ subproblems, is polynomial (quadratic) in $n$. The dominating component of that size is given by $B_n$: since the size of $T_i$ and $U_i$ grows linearly with $i$, the size of $B_i$ grows as $i^2$, dominating the growth of $E_i$ and of the compared types, which is linear.

Notice that the same judgement would be checked in exponential time by the constraint generating algorithm too. However, we will show that the constraint generating algorithm would only generate a polynomial amount

of different subproblems, hence it is easy to modify it in order to give it a polynomial complexity.

### 5.2 The polynomial algorithm

The non memoizing, hence exponential, constraint generating algorithm can be described by the following SML program. The program does not explicitly enforce the assumption that, whenever two different `Var` or `All` terms are associated with two `{name, bound}` tuples with the same name, the tuples also have the same bound (and are actually represented by the same memory location, though this is not relevant here). We may explicitly check this fact if we do not trust the input provider, but this check would not affect the complexity of the type checking. Also note that there exists no central $B$ environment, but every variable carries its own bound. Apart from this, the algorithm below exactly mimics the rules we defined, provided that the `swap`, `append`, and `minus` operations are defined as in Section 4 (`#f r` denotes the contents of the `f` field of the `r` record).

```
datatype Type =
  Var of {name: string, bound: Type}
  | Top
  | All of {name: string, bound: Type}*Type
  | Arrow of Type * Type;
type VarPair = {name: string, bound: Type};
datatype Constraint = Less of VarPair*VarPair
  | Greater of VarPair*VarPair;
...
fun getSigma (Var(t)) (Var(u)) = [Less(t,u)]
  | getSigma T (Top) = [ ]
  | getSigma (Var({name,bound})) T
    = getSigma bound T
  | getSigma (All(t,T)) (All(u,U)) =
    let val sigma = getSigma (#bound u) (#bound t)
        val sigma1 = getSigma (#bound t) (#bound u)
        val sigma2 = getSigma T U
    in append [ (swap sigma), sigma1, minus sigma2 (t,u) ]
    end
  | getSigma (Arrow(T,TT)) (Arrow(U,UU)) =
    let val sigma = getSigma U T
        val sigma1 = getSigma TT UU
    in append [ (swap sigma), sigma1 ]
    end;
```

Let us now define the notion of a type being a subterm of another one as the minimal transitive relation such that: `bound` is a subterm of `Var({name,bound})`; `bound` and `T` are subterms of `All({name,bound},T)`; `T` and `U` are subterms of `Arrow(T,U)`. The constraint generating algorithm enjoys the following "subterm property": for every subproblem "`getSigma TT UU`" generated by a problem "`getSigma T U`", `TT` and `UU` are subterms of the types `T` and `U`. We can define the input size of a problem `getSigma TT UU` to be the number of different occurrences of subterms of `TT` and `UU`. Hence, if $n$ is the size of the problem `getSigma TT UU`, its analysis generates no more than $n^2$ different subproblems, one for every different pair of subterms of `TT` and `UU`. Hence, if the algorithm is modified so that it remembers every already generated subproblem, it will call itself recursively no more than $n^2$ times.[7]

The modified algorithm is shown below. It first uses `lookForOldSolution` to search in the `memory` data structure for the constraint set generated by a previous analysis of the same subproblem, and calls `newGetSigma` only the first time it meets a pair of types. `newGetSigma` is identical to `getSigma`, but every direct recursive call to `getSigma T U` is substituted by a call to `checkSigma T U`. The function `rememberSolution` stores the `T, U, result` triple in the `memory` data structure.

```
fun checkSigma T U =
  case lookForOldSolution T U (memory)
  of Solved(result) =>- result
  | Unsolved =>-
      let val result = newGetSigma T U
      in (rememberSolution T U result memory;
          result)
      end
and newGetSigma (Var(t)) (Var(u)) = ...
```

Notice that every branch of the algorithm only performs polynomial operations. The critical ones are the `append` and `minus` operations. `append` should be implemented as a set union, rather than a list concatenation, to prevent constraint lists from becoming too long. Thus, the length of any constraint list is bound by the input size, hence both `append` and `minus` can be executed in polynomial time. This completes the proof of the fact that our algorithm runs in polynomial time.

Once the asymptotic complexity has been fixed, we should now look for more realistic implementations. The "better" algorithm we are currently experimenting with differs from the one presented above because we check and update the `memory` structure only when we apply the (Var$\leq$) rule. The net result is that, for most judgements, where memoization gives little advantage, we save a lot of costly `lookForOldSolution` and `remember` operations, while exponential judgements are still checked in polynomial time.

[7] Nothing would change if we suppose that the algorithm input is not a tree but a graph, so that common subterms can be shared, since in this case the upper bound for the number of recursive calls would still be $n^2$, where $n$ is the number of nodes in the graph.

## 6 Related Work

The complexity of subtype checking problem has not been studied in the literature, to our knowledge, essentially because most interesting subtype checking problems are either obviously polynomial, as happens with first order systems ([Car84]), or undecidable, as happens for system $F_{\leq}$ [Pie94, Ghe95]. When decidable variants of $F_{\leq}$ have been defined, proving their decidability has been the main interest focus ([Bru94, CP94]). We suspect, however, that our results may be easily transferred to those systems.

A lot of work has been performed about the complexity of type inference problems, usually in the context of ML-like systems, sometimes enriched with subtyping, as in, e.g., [LM92]. However, the problem of type inference in quite different from the subtype checking problem we face here, and there is no apparent relation between the results from this field and our result.

## 7 Conclusions

We have presented a polynomial algorithm to perform kernel Fun subtype checking.

In the process, we have formalized a technique to compare universal types without operating substitutions, and have proved the soundness and completeness of this technique. In particular, we have shown that, when applied to kernel Fun, this technique allows subtyping to be checked without creating new types and without renaming type variables (Theorems 3.7, 3.10). Apart from allowing polynomial subtype checking, this property is also important when kernel Fun is extended with recursive types. Subtype checking among recursive types is, in fact, based on the comparison of subtype judgements with the previously explored ones [AC93]. If no new types are generated, this process is guaranteed to terminate, and judgement comparison can be implemented as a pointer comparison. These facts imply that system kernel Fun can be extended with recursive types along the lines of [AC93] without meeting the subtle problems raised by recursive types in system $F_{\leq}$ [Ghe93b].

We have also shown that the standard subtype checking algorithm for system kernel Fun is exponential, and may actually generate an exponential number of different subproblems. These properties of the algorithm were previously unknown.

Finally, we have developed a type theory where "variable names are taken seriously". The traditional approach, which essentially ignores variable names, is still the best when fundamental studies are carried out. However, the lack of studies about variable name management in the subtype checking process is a problem for those who would like to implement a subtype checker without using De Bruijn indexes.

## References


[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC94] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *European Symposium on Programming (ESOP), Edinburgh, Scotland*, 1994.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.

[BCGS91] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in [GM94].

[Bru91] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.

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

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76:138–164, 1988.

[CCHO89] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA 89*, pages 457–467, 1989.

[CG92] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992.

[CG94] P.-L. Curien and G. Ghelli. Decidability and confluence of $\beta\eta\text{top}_{\leq}$ reduction in $F_{\leq}$. *Information and Computation*, 109(1, 2):57–114, 1994.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *POPL '90*, 1990.

[CHO88] Peter Canning, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

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

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[Com94] A.B. Compagnoni. Decidability of Higher-Order Subtyping with Intersection Types Proceedings of Computer Science Logic, September 1994.

[CP94] Giuseppe Castagna and Benjamin Pierce. Decidable bounded quantification. In *POPL '94*. ACM, January 1994.

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

[DT88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, 1995.

[ESTZ94] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA '94*, 1994.

[FM94] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software, Sendai, Japan*, pages 844–885. Springer-Verlag, April 1994. LNCS 789.

[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. Modelling features of object-oriented languages in second order functional languages with subtypes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 311–340, Berlin, 1991. Springer-Verlag.

[Ghe93a] G. Ghelli. Divergence of $F_{\leq}$ type checking. Technical Report 5/93, Dipartimento di Informatica, Università di Pisa, 1993.

[Ghe93b] G. Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezen and J.F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA) 93*, number 664 in LNCS, pages 146–162, Berlin, March 1993. Springer-Verlag.

[Ghe95] G. Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1,2):131–162, 1995.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

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

[GP92] G. Ghelli and B. Pierce. Bounded existentials and minimal typing. Available by ftp, 1992.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1995. To appear.

[KS92] Dinesh Katiyar and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[LM92] P.D. Lincoln and J.C. Mitchell. Algorithmic aspects of type inference with subtypes. In *POPL '92*, pages 293–304, January 1992.

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.

[Mit90] J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *POPL '90*, 1990.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in [GM94].

[PS] B.C. Pierce and M. Steffen. Higher-order subtyping. Submitted for publication. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.