Divergence of $F_{\leq}$ type checking

Giorgio Ghelli[1]

Abstract

System $F_{\leq}$ is an extension of second-order typed lambda calculus, where a subtype hierarchy among types is defined, and bounded second-order lambda abstraction is allowed. This language is a basis for much of the current research on integration of typed functional languages with subtypes and inheritance. An algorithm to perform type checking for $F_{\leq}$ expressions has been known since the language Fun was defined. The algorithm has been proved complete, by the author and P.-L. Curien, which means that it is a semi-decision procedure for the type-checking problem. In this paper we show that this algorithm is not a decision procedure, by exhibiting a term which makes it diverge. This result was the basis of Pierce's proof of undecidability of typing for $F_{\leq}$. We study the behavior of the algorithm to show that our diverging judgement is in some sense contained in any judgement which makes the algorithm diverge. On the basis of this result, and of other results in the paper, we claim that the chances that the algorithm will loop while type-checking a "real program" are negligible. Hence, the undecidability of $F_{\leq}$ type-checking should not be considered as a reason to prevent the adoption of $F_{\leq}$ as a basis for defining programming languages of practical interest. Finally, we show the undecidability of an important subsystem of $F_{\leq}$.

1 Introduction

The language '*Fun*' was introduced in [Cardelli Wegner 85] to formalize the relationships between subtyping, polymorphism, inheritance and modules in a strongly typed language. *Fun* in its entirety is very rich, but a subsystem of it, called $F_{\leq}$, has been recognized as a minimal kernel which collects the main technical substance of the recursion-free part of the language. Technically, system $F_{\leq}$ is an extension of the second-order λ-calculus defined by Girard and Reynolds, [Girard 72] [Reynolds 74], with subtypes, bounded second-order abstraction, and a maximum type *Top* which allows unbounded quantification. $F_{\leq}$ was formalized in [Curien Ghelli 92], by modifying the system defined in [Bruce Longo 90]. Extensions of *Fun* and $F_{\leq}$ are the basis of most current research on the integration of the capabilities of object-oriented languages and functional languages in a strongly typed context (see, e.g., [Canning Hill Olthoff 88]

[1]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125, Pisa, Italy, ghelli@di.unipi.it.
This work was carried out with the partial support of E.C., Esprit Basic Research Action 6309 FIDE2, of the Italian National Research Council, "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" grant No. 92.01561.PF69, and of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica.

[Cook 89] [Canning et al 89] [Mitchell 90] [Danforth Tomlinson 88] [Cardelli Martini Mitchell Scedrov 91][Ghelli 91] [Bruce 91] [Cardelli Mitchell 91] [Mitchell et al 91] [Bruce 92] [Hofmann Pierce 94] [Bruce 93] [Ghelli 93a] [Gunter Mitchell 93] [Pierce Turner 93]).

An algorithm to assign a type to every well-typed *Fun* term and to check whether a type is a subtype of another was already known when the language was presented, and can be attributed to Luca Cardelli. In [Curien Ghelli 92] this typing algorithm was formalized for $F_{\leq}$ and proved correct and complete. In the same paper the algorithm is shown to be the "natural" one with respect to a notion of "normal form" of type-checking proofs. Correctness and completeness mean that the algorithm successfully terminates on all and only the typable terms, but do not imply that it terminates on non-typable terms. The termination of subtype checking would immediately imply the termination of type checking, but it was not even known whether subtype checking terminates or not.

In this paper we show that there are terms which make that algorithm diverge, contradicting the faulted termination proof given in [Ghelli 90]. Then we study the features which characterize those judgements which make the algorithm diverge (the "diverging judgements").

The basic aim of this study was to settle a basis on which to determine whether the $F_{\leq}$ typing problem is decidable. We were successful in this, since our result was actually the basis of Benjamin Pierce's proof of undecidability of $F_{\leq}$ [Pierce 93]. Even though this part of the problem has been closed, our analysis of the algorithm behavior is still useful to understand what makes the problem difficult. This kind of information may be used to design decidable variants of the language; such variants have been recently proposed, for example, in [Katiyar Sankar 92] and [Castagna Pierce 94].

Another reason to study diverging judgements is to understand whether they may appear in "real programs". Here we claim that diverging judgements are artificial ones, which do not arise "naturally" in real programming, and we substantiate this claim by defining a set of features which must be shared by all diverging judgements. The awkwardness of these constraints supports our belief that the undecidability of $F_{\leq}$ is not a problem of practical concern, hence that $F_{\leq}$ can be safely used as a basis for designing programming languages.

Finally, diverging judgements are related to the addition of recursive types to $F_{\leq}$. In [Ghelli 93a] we showed that, surprisingly enough, type-level recursion is not conservative over $F_{\leq}$ subtyping. This means that there are some $F_{\leq}$ unprovable subtyping judgements which become provable (by transitivity) when recursive types (regular infinite trees) are added to $F_{\leq}$. One of these judgements is in fact the diverging judgement introduced here. In [Ghelli 93a] we also show that the set of non-recursive judgements which become provable by adding recursive types is (properly) included in the set of diverging judgements. This result shows that, even though diverging judgements are defined here in terms of the behavior of a specific algorithm, they have a wider role in $F_{\leq}$, which should be better understood.

In this paper, we also show the undecidability of an important variant of $F_{\leq}$, system *Fbq*. Other variants are discussed in [Ghelli 93b].

This paper is structured as follows. Section 2 introduces the language and the algorithm. In Section 3 we show a judgement which makes the standard type checking algorithm for

$F_{\leq}$ diverge. Section 4 studies the features of any diverging judgment, showing the minimality of the one presented in Section 2. Finally in Section 5 we prove that type checking the subsystem *Fbq* of $F_{\leq}$ is as hard as typechecking the whole $F_{\leq}$.

2 The language and the algorithm

2.1 The language

The language $F_{\leq}$ was defined in [Curien Ghelli 92], as a more essential version of Cardelli and Wegner's *Fun* language. The syntax of $F_{\leq}$ is defined as follows:

Types	$T ::= t \mid Top \mid T \rightarrow T \mid \forall t \leq T.\ T$
Terms	$a ::= x \mid top \mid \lambda x{:}T.\ a \mid a(a) \mid \Lambda t \leq T.\ a \mid a\{T\}$
Environments	$\Gamma ::= () \mid \Gamma, t \leq T \mid \Gamma, x{:}T$
Judgements	$J ::= \Gamma \vdash a : T \mid \Gamma \vdash T \leq T$

$\Lambda t \leq T.a$ is the second-order abstraction of the expression a with respect to the type variable t; the bound $\leq T$ means that only subtypes of T are accepted as parameters. $a\{T\}$ is the corresponding application of a function to a type.

The type *Top* is a supertype of all types, useful for codifying an unbounded second-order lambda abstraction as $\Lambda t \leq Top.a$; *top* is a "canonical" term of type *Top*.

$\forall t \leq T_1.T_2$ is the type of a function $\Lambda t \leq T_1.a$, with T_2, the type of a, generally depending on t.

A judgement $\Gamma \vdash a{:}T$ means that a has type T with respect to the environment Γ, which collects information about the free variables of a and T; $\Gamma \vdash T \leq U$ means that T is a subtype of U, i.e. that an expression of type T can be used in any context where an expression of type U can be used, again with respect to Γ.

The constants $\forall$, λ, Λ bind their variable in their second argument, as usual; similarly a definition $t \leq A$ in the environment binds t in the following part of the judgement; the *scope* of a variable is the part of the judgement where that variable is bound. In a quantified type $\forall t \leq A.B$ and in an environment $\ldots t \leq A \ldots$ we say that A is *a bound* (i.e. an upper limit) for t, and that t is bounded by A.

Throughout the paper we always distinguish between "a variable" and "an occurrence of a variable". The use of these terms is best explained by an example: in the judgement

$$t \leq Top \vdash \forall u \leq \underline{t}.\ \underline{t} \rightarrow \underline{u} \leq Top$$

there are two *variables* (t and u), two *occurrences* of the variable t and one *occurrence* of the variable u; these three occurrences are underlined (a more formal definition of occurrence is in Section 4). Two variables are *different* when there is one α-variant of the judgement where they have different names (α-equivalence is defined as usual). For example, in the following judgement we have three different variables with the same name t, and we have two bound occurrences of each variable:

$$t \leq Top \vdash t \rightarrow (\forall t \leq Top.\ t \rightarrow t) \leq t \rightarrow (\forall t \leq Top.\ t \rightarrow t)$$

A judgement is well formed if all variable occurrences are bound and all different variables have different names; hence the above judgement is not well formed[2].

[2] Names of variables may be seen, as usual, as a readable denotation of their DeBruijn indexes [DeBruijn 72]; however, in this context, carefully managing names of variables helps to avoid some pitfalls.

The typing rules of the language are grouped together in Appendix A for reference. These rules are implicative formulae which may be read as Horn clauses, which define a type-checking algorithm in a Prolog style, by specifying how to reduce the type-checking or subtype-checking problem in the consequence to the type-checking or subtype-checking problems in the premises. But two of these rules, *(Subsump)* and *(Trans)*, would make any Prolog interpreter diverge, since they reduce a problem to the same problem *(Subsump)*, or to a pair of more general problems *(Trans)*. In proof-theoretic terms they both resemble a "cut rule". In [Curien Ghelli 92] any provable subtyping or typing judgement was proved to admit a single "normal form" cut-free proof, and an alternative set of rules was defined which produces all and only the "normal form" proofs of $F_{\leq}$. The operational interpretation of these "algorithmic rules" (reported in Appendix B) defines a pair of deterministic algorithms:

- a type checking (or *type assignment*) algorithm $\Gamma \vdash a:A$, which computes A from Γ and a;
- a subtype checking algorithm $\Gamma \vdash A \leq B$, which, given Γ, A and B, either is successful or fails.

Both algorithms work as follows: the input problem is compared with the conclusion of all the rules, the only matching rule is used to reduce the problem to the subproblems in the premises of the rule, and finally these subproblems are solved in the specified order, by recursively applying the same algorithm (see also Section 2.2). The algorithm terminates with success when all the subproblems match the terminal rules $AlgId\leq$, $AlgTop\leq$, $AlgVar$ and $AlgTop$; it terminates with failure when no rule matches a subproblem (e.g. $\Gamma \vdash Top \leq t$), or when an output type does not match the expected shape[3]. Note that this algorithm is deterministic (without backtracking), since for each judgement there is at most one applicable rule. This determinism was achieved by reducing the scope of transitivity, which can only be applied to type variables (*AlgTrans*), and the scope of subsumption, which can only be used within function application (*AlgApp, AlgApp2*).

The correctness and completeness of the above algorithm are proved in [Curien Ghelli 92]. Correctness means that if the algorithm answers *'A'* to a question *'$\Gamma \vdash a:?$'* then $\Gamma \vdash a:A$ is provable in the system; this can be proved easily, since the algorithm merely applies rules which are derivable within the system.

Completeness means that if $\Gamma \vdash a:A$ is provable in the system, the algorithm applied to the input Γ, a terminates, with a correct answer (actually it returns the minimum correct type). But note that in [Curien Ghelli 92] the fact that the algorithm terminates, on typable terms, is not proved by studying its computational behavior, but only indirectly, as a consequence of the fact that any provable judgement has a finite "canonical" proof, and that each step of the algorithm builds a piece of this finite proof.

Correctness and completeness of the type checking algorithm do not imply that the problem is decidable, since the algorithm may still diverge on non-typable terms. Decidability of type checking would follow immediately from decidability of subtype checking, since a rule for $\Gamma \vdash a:?$ only invokes the same algorithm applied to strict

[3]More precisely, when in rules *(AlgApp)* and *(AlgApp2)*, the minimum non variable supertype $\Gamma^*(T)$ of the type T of the applied term f does not match $A \rightarrow B$ or $\forall t \leq A.B$, respectively (see Appendix B).

subterms of a and the subtype checking algorithm. For this reason, in the rest of the paper we will only study the subtype checking algorithm, which is the hard kernel of type checking.

2.2 The subtype checking algorithm

In this section we formally describe the subtype-checking algorithm, with the help of a term rewriting relation "$\multimap$", which reduces a judgement to its antecedents in the applicable subtyping rule.

From now on we study a simplified type system without the $\rightarrow$ type constructor, since it does not add any complexity to the subtype relation: in fact, $\Gamma \vdash A \rightarrow B' \leq A' \rightarrow B$ is provable if and only if $\Gamma \vdash \forall \xi \leq A.B' \leq \forall \xi' \leq A'.B$ is provable, where ξ and ξ' are fresh type variables.

In our study of the algorithm we want to be able to follow the evolution of a variable through different "$\multimap$" rewriting steps. To this aim, when two different variables are unified by the backward application of the $(\forall \leq)$ rule:

$$(\forall \leq) \qquad \frac{\Gamma \vdash A \leq A' \quad \Gamma, t' \leq A \vdash B[t'/t] \leq B'}{\Gamma \vdash \forall t \leq A'.B \leq \forall t' \leq A.B'}$$

instead of applying the substitution $B[t'/t]$, we will record the unification of t and t' in the environment and leave B and B' as they are, by writing $\Gamma, (t=t') \leq A' \vdash B' \leq B$.

For a similar reason, we duplicate the $\leq$ relation into two relations, $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \geq A$, such that $\Gamma \vdash A \leq B \Leftrightarrow \Gamma \vdash B \geq A$. This allows the $(\forall \leq)$ rule to be rewritten as follows (recall that "$\multimap$" denotes the backward application of a rule):

$$(\forall \leq) \qquad \Gamma \vdash \forall t \leq A.B' \leq \forall t' \leq A'.B \quad \multimap \quad \{ \ \Gamma \vdash A \geq A', \ \ \Gamma, (t=t') \leq A' \vdash B' \leq B \ \}$$

In this way, the residuals A and B' of the left hand side of the comparison $\forall t \leq A.B'$ are still on the left hand side, and similarly for the right hand side. This notation is exploited, in particular in Section 4.5, to study the "$\multimap$" reduction invariant properties of each side of the comparison. Hereafter we will usually only give definitions and examples in terms of the $\Gamma \vdash A \leq B$ case; the other case is always defined symmetrically.

To summarise, the syntax of the types and judgements managed by the algorithm is:

Types	A	::=	$Top \mid t \mid \forall t \leq A.A$
Environments	Γ	::=	$() \mid \Gamma, (t=t) \leq A$
Judgements	J	::=	$\Gamma \vdash A \leq A \mid \Gamma \vdash A \geq A \mid true \mid false$

In well-formed judgements all variables are bound, and different variables have different names.

The reduction relation "$\multimap$" is defined by the following term rewriting rules, plus the symmetric rules obtained by exchanging $A \leq B$ with $B \geq A$ ($\Gamma \vdash t \approx u$, read "$\Gamma$ unifies t and u", means that either t is u, or $(u=t) \leq A$ is in Γ, or $(t=u) \leq A$ is in Γ; "$\approx$" is only defined on variables):

$$\begin{array}{lll}
\textit{(lhs-top)} & \Gamma \vdash A' \le \mathit{Top} \ \longrightarrow\!\triangleright \ \mathit{true} \\
\textit{(lhs-varId)}\ \Gamma \vdash t \approx u & \Rightarrow\ \Gamma \vdash u \le t \ \longrightarrow\!\triangleright \ \mathit{true} \\
\textit{(lhs-exp)}\ \Gamma \nvdash C \approx u,\ C \neq \mathit{Top} & \Rightarrow\ \Gamma \vdash u \le C \ \longrightarrow\!\triangleright \ \Gamma \vdash \mathit{FreshNames}(\Gamma(u)) \le C \\
\textit{(lhs-}\forall\textit{dom)} & \Gamma \vdash \forall t{\le}A.B' \le \forall t'{\le}A'.B \ \longrightarrow\!\triangleright \ \Gamma \vdash A \ge A' \\
\textit{(lhs-}\forall\textit{cod)} & \Gamma \vdash \forall t{\le}A.B' \le \forall t'{\le}A'.B \ \longrightarrow\!\triangleright \ \Gamma, (t{=}t'){\le}A' \vdash B' \le B
\end{array}$$

$$\textit{(lhs-false)}\ \textit{when nothing else applies:}\ \ \Gamma \vdash A \le B \ \longrightarrow\!\triangleright \ \mathit{false}$$

The above rules will be called "left hand side rules"; the "right hand side rules" are obtained by inverting the comparisons, like in the *(rhs-exp)* rule below (hereafter, we will omit the lhs-/rhs- prefix when it is not needed):

$$\textit{(rhs-exp)}\ \Gamma \nvdash C \approx u,\ C \neq \mathit{Top} \ \Rightarrow\ \Gamma \vdash C \ge u \ \longrightarrow\!\triangleright \ \Gamma \vdash C \ge \mathit{FreshNames}(\Gamma(u))$$

In the *(exp)* rule, $\Gamma(u)$ is the bound of u defined in Γ; $\mathit{FreshNames}(\Gamma(u))$ renames all the variables defined inside $\Gamma(u)$ with unused variable names, to preserve the invariant that different variables in a judgement have different names[4]. We will write *(exp)(A)* to denote an *(exp)* step which expands the variable to A (i.e., A is $\mathit{FreshNames}(\Gamma(u))$. After the execution of a $(\forall cod)$ step, the definitions $\forall t$ and $\forall t'$ of the variables t and t' disappear from the comparison and appear, as $(t{=}t'){\le}A'$, in the environment. For this reason, we will often say that a $(\forall cod)$ step "moves the definitions of t and t' into the environment".

The only two normal form terms of the above system are *true* and *false*. Each judgement which differs from *true* and *false* can be reduced by exactly one rule, with the only exception of judgements of the form $\Gamma \vdash \forall t{\le}A.B' \le \forall t'{\le}A'.B$, reduced by the two $\forall$ rules. $(\forall dom)$ is the only rule which inverts the direction of the comparison. When there exists one infinite reduction chain which starts from a judgement J, we say that J is a *diverging judgement.*

The subtype-checking algorithm works by maintaining a "to-do list" of subtyping judgements to be proved, which initially only contains the input judgement. At each step, one of the to-do judgements is substituted with its immediate antecedent(s), by applying the "$\longrightarrow\!\triangleright$" rewrite rules. A judgement which reduces to *true* is simply removed from the list. If a judgement in the to-do list reduces to *false*, the algorithm stops and reports a failure, meaning that the original judgement was not provable. The algorithm stops with success when the list is emptied.

This algorithm explores the set of all the "$\longrightarrow\!\triangleright$" chains which start from a judgement J; it stops either when it meets one chain which terminates with *false*, or when all chains are built, and all of them terminate with *true*. If both infinite chains and chains ending with *false* start from a judgement J, then the algorithm may either diverge or stop, depending on how it manages its to-do list. However, we will exhibit a judgement J which is diverging but does not rewrite to *false*; when applied to such a judgement, the algorithm necessarily diverges. For this reason, in the rest of the paper we will ignore the problem of the choice of the judgement to be rewritten at each step[5], but we will focus

[4] In our examples we use Greek letters for variables defined inside a bound in the environment to emphasize the fact that these names must be changed any time the bound is copied into the comparison:

$$\ldots, u \le \forall \xi{\le}t.\xi \vdash u \le A \ \longrightarrow\!\triangleright (exp)\ \ \ldots, u \le \forall \xi{\le}t.\xi \vdash \forall v{\le}t.v \le A$$

on the exploration of a single rewriting chain, and on the existence of infinite rewriting chains.

Notation: Hereafter we will use these abbreviations:

$\forall t.A$	abbreviates	$\forall t{\leq}Top.A$	
$-A$	abbreviates	$\forall t{\leq}A.Top$	where t is a fresh variable

These abbreviated terms can be reduced by the following derived rules:

$(\forall dom')$	$\Gamma \vdash -A \leq -A'$	$\longrightarrow$	$\Gamma \vdash A \geq A'$
$(\forall cod')$	$\Gamma \vdash \forall t.B' \leq \forall t'{\leq}A'.B$	$\longrightarrow$	$\Gamma, (t=t'){\leq}A' \vdash B' \leq B$

2.3 Executing judgement rewriting

The judgement rewriting process can be seen as an interleaving of scanning and substitution steps performed on the compared types. This point of view will be useful, in particular, in Sections 3 and 4.4.

We can represent each compared type by a tree plus a pointer which specifies which subtree is being considered (see Figure 1). Then, a $(\forall)$ step moves the pointer down the two trees, while an *(exp)* step substitutes a leaf which is a variable with a copy of its bound, renamed by *FreshNames*. For example, the following reduction sequence:

$$\vdash \forall t{\leq}(\boldsymbol{\forall z{\leq}A.z}).C \geq \forall t'{\leq}(\boldsymbol{\forall z'{\leq}A'.B'}).C' \quad \longrightarrow (\forall dom)$$
$$\vdash \forall z{\leq}A.\boldsymbol{z} \leq \forall z'{\leq}A'.\boldsymbol{B'} \quad \longrightarrow (\forall cod)$$
$$z=z'{\leq}A' \vdash \boldsymbol{z} \leq B' \quad \longrightarrow (exp)$$
$$z=z'{\leq}A' \vdash FreshNames(A') \leq B'$$

can be visualized as in Figure 1 (the dashed pointer points to the smaller side).

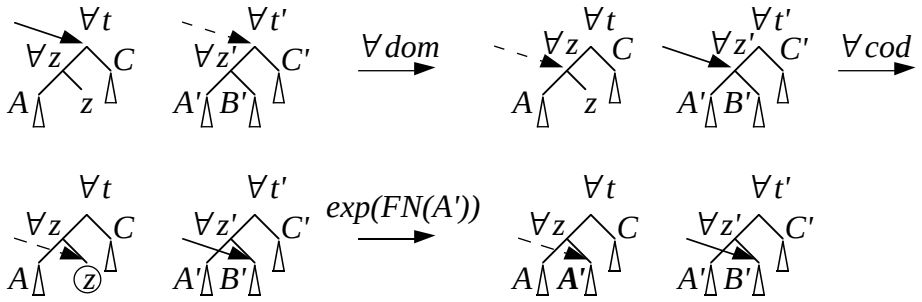

Figure 1. Subtype checking means scanning+substituting.

[5]The simplest, and most efficient, approach is to explore the rewriting chains in a depth-first way, holding the to-do list in the stack. By exploring the different chains in a breadth first way, i.e. by cycling between all judgements in the to-do list, divergence would be avoided on judgements which both diverge and rewrite to *false*. However, breadth-first exploration would not be worthwhile in practice, in view of the claim that divergent judgements do not arise in real programs.

3 A diverging judgement

Since *Fun* was defined, the algorithm in Section 2.1 was considered to be the natural one to type-check it. It was believed to be a decision procedure, and some researchers tried to prove this fact. The problem was apparently settled by the author, who produced a "proof" of termination of the algorithm. This "proof" was published in [Ghelli 90] and checked by a few people, until Pierre-Louis Curien and, independently, John Reynolds, discovered a subtle bug in it. The attempts to remove that bug finally produced a surprising result: the algorithm is not a decision procedure, and a diverging judgement can be written.

A minimal diverging judgement is:

$$v_0 \le (\forall \xi.\text{-}\forall \psi \le \xi.\text{-}\xi) \quad \vdash \quad v_0 \quad \le \quad \forall u_1 \le v_0.\text{-}v_0$$

This judgement produces no chain ending with *false* and produces only one infinite rewriting chain. The first few judgements in this chain are listed below:

$B = \forall \xi.\text{-}\forall \psi \le \xi.\text{-}\xi$

1)	$v_0 \le B$	$\vdash$	v_0	$\le$	$\forall u_1 \le v_0.\text{-}v_0$ *(lhs-exp)*
2)	$v_0 \le B$	$\vdash$	$\forall v_1.\text{-}\forall u_2 \le v_1.\text{-}v_1$	$\le$	$\forall u_1 \le v_0.\text{-}v_0$ *(lhs-∀cod')*
3)	$v_0 \le B,\ u_1 = v_1 \le v_0$	$\vdash$	$\text{-}\forall u_2 \le v_1.\text{-}v_1$	$\le$	$\text{-}v_0$ *(lhs-∀dom')*
4)	$v_0 \le B,\ u_1 = v_1 \le v_0$	$\vdash$	$\forall u_2 \le v_1.\text{-}v_1$	$\ge$	v_0 *(rhs-exp)*
5)	$v_0 \le B,\ u_1 = v_1 \le v_0$	$\vdash$	$\forall u_2 \le v_1.\text{-}v_1$	$\ge$	$\forall v_2.\text{-}\forall u_3 \le v_2.\text{-}v_2$ *(rhs-∀cod')*
6)	$v_0 \le B,\ u_1 = v_1 \le v_0,\ u_2 = v_2 \le v_1$	$\vdash$	$\text{-}v_1$	$\ge$	$\text{-}\forall u_3 \le v_2.\text{-}v_2$ *(rhs-∀dom')*
7)	$v_0 \le B,\ u_1 = v_1 \le v_0,\ u_2 = v_2 \le v_1$	$\vdash$	v_1	$\le$	$\forall u_3 \le v_2.\text{-}v_2$ *(lhs-exp)*
8)	$v_0 \le B,\ u_1 = v_1 \le v_0,\ u_2 = v_2 \le v_1$	$\vdash$	v_0	$\le$	$\forall u_3 \le v_2.\text{-}v_2$ *(lhs-exp)*

We will now try to describe informally what happens; all the ideas sketched here will be formalized in Section 4.

We say that a variable t refers to u if either u appears free in the bound of t, or if t refers to some v which in turn refers to u; e.g., in $\forall v \le u.\forall t \le v.t$, t refers to both v and u (this is made formal in Section 4.5). The typing rules enforce that no variable can refer to itself; this implies, apparently, that once a variable has been expanded in both sides of the comparison, it cannot appear in the comparison anymore. This was the main assumption supporting the idea that the algorithm should always terminate.

However, expanded variables can be reintroduced into a comparison, due to the *(∀cod)* rule which changes the bound of the variable on the smaller side. Consider Figure 2, where the first three steps of the infinite chain are depicted; in the four comparisons, the variable occurrences which refer to v_0 are underlined; the environment is depicted on top of the compared types.

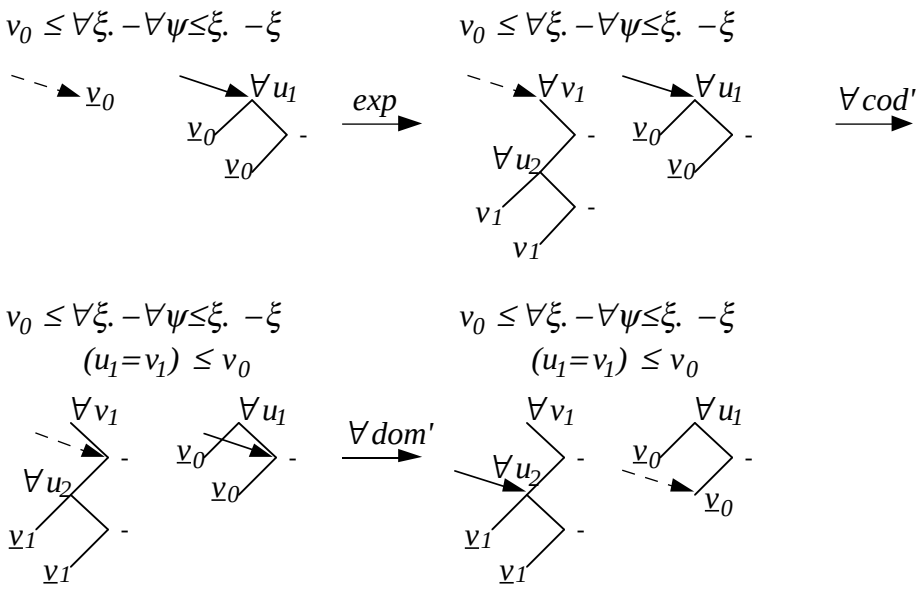

Figure 2. The first three steps of the infinite rewriting chain.

In the first step v_0 is substituted by a bound which does not refer to v_0. However, when the *(∀cod')* step unifies u_1 with v_1, the bound of v_1 becomes v_0, hence the left hand side now contains two references to v_0. The next *(∀dom')* step inverts the roles of the two sides and recreates, essentially, the initial situation, with v_0 referred by both sides. Note the different roles of the two v_0's in $\forall u_1 \le v_0.\text{-}v_0$: the first one is moved in the environment and is needed to make v_1 a reference to v_0; the second one is the one which will be expanded. Exactly the same roles will be played by the two occurrences of v_1 in $\forall v_1.\text{-}\forall u_2 \le v_1.\text{-}v_1$. This is the basis for an infinite series of expansions of the same variable.

Although this minimal diverging judgement exhibits a kind of cyclicity which seems easy to detect, the reader can verify that this pattern could be enriched in increasingly complex ways. Actually, Pierce's undecidability result implies that there is no general way to detect whether the algorithm enters an infinite loop [Pierce 93].

The rest of this paper analyses the behavior of the subtype checking algorithm. This analysis defines a set of constraints on the shape of diverging judgements, which show that all of these judgements must share a rather complex structure, and that the diverging judgement above exhibits, in some sense, the typical behavior of any diverging judgement. This "uniqueness" of the diverging judgement means that any attempt at designing a decidable variant of $F_{\le}$ can be focussed on avoiding this kind of divergence.

4 The behavior of the subtype checking algorithm

4.1 Overview

In this section we study the properties of diverging judgements, to show that they all share the basic features of our minimal case, as will be elaborated. In fact, these studies were first performed without knowing whether a diverging judgement existed or not, and their final result was the design of the judgement presented in the previous section.

We first show (Section 4.2) that in a rewriting chain nothing new is ever created: every type occurring in any judgement in the chain is equal, up to variable renaming, to some type occurring in the first judgement of the chain. We then associate a polarity with every occurrence of a type in a judgement, and show that reduction preserves polarity. These facts imply that every infinite rewriting chain eventually compares infinitely many times the same pairs of types (up to variable renaming). Hence, the complexity of the problem essentially comes from the possibility of an unlimited growth of the environment.

In Section 4.3 we prove that, in any diverging judgement, a variable exists which is expanded infinitely many times. This is a key result, and is the basis of most of the other results of Section 4. It is proved by first showing that: (a) in an infinite rewriting chain infinitely many new variables are created; (b) that new variables are always created by expanding variables with a *strictly* bigger bound.

In Section 4.4 we define a reduction invariant, called the inversion depth, defined as the maximum nesting level of bounds inside bounds. We show that a judgement may diverge only if this nesting level is at least three. This is the first result supporting our claim that "only weird judgements diverge".

In Section 4.5 we go back to the result that in a diverging judgement there is one variable which is expanded infinitely many times, and describe the conditions which make this possible. To this aim, we first formalize the notion of reachability informally introduced in Section 3. Then we show that, when a variable is expanded on one side, it is not reachable from that side in the resulting judgement, but it may become reachable again by obtaining a reference to that variable from the other side. To this aim, the *(∀cod)* rule must be used in a very specific way, which is described in this section. This way of using the *(∀cod)* rule is the second piece of evidence that we give for the "only-weird-judgements-diverge" claim.

Finally Section 4.6 shows that the shape of our minimal judgement is typical for diverging judgements. More precisely, we show that every occurrence in the bound of our diverging judgement derives from the need to regain a reference to an expanded variable. This implies that any diverging judgement must always contain, buried under other details, the same pieces as our minimal one, all of them playing the same roles.

4.2 Basic properties and definitions

In this section we show that in a rewriting chain essentially the same pairs of types are always compared.

We first collect some definitions (occurrence, closed form of a judgement, occurrence with respect to a judgement, polarity) which will be used in the next subsections.

Definition 4.1 (*occurrence*): An *occurrence* μ is a string of 0's and 1's, used to refer to a subterm A/μ of a type A as follows (ε is the empty sequence; "." denotes concatenation):

$A/\varepsilon = A$
$(\forall t \leq A.B)/(0.\mu) = A/\mu$
$(\forall t \leq A.B)/(1.\mu) = B/\mu$

Intuitively, μ specifies a path to be followed to extract A/μ from A: a 0 directs into

the bound and a 1 directs into the codomain; the subterm is found when the path ends.

The *valid occurrences of a type T* are all those occurrences μ such that T/μ is defined. $A[B/\mu]$ denotes the result of substituting the subterm at the occurrence μ of A with B; $A[B/t]$ (variable substitution) means substituting B at all the occurrences of t. In both cases, we will explicitly handle variable renaming.

Definition 4.2 (*closed form of a judgement*): For any judgement J

$$t'_1=t_1\leq A_1,\ldots,t'_n=t_n\leq A_n \vdash T \leq U$$

the judgement

$$\vdash \forall t'_1 \ldots \forall t'_n.T[t'_i/t_i] \leq \forall t_1\leq A_1.\ldots. \forall t_n\leq A_n.U[t_i/t'_i]$$

is called *the closed form* of J.

The compared types $\forall t'_1 \ldots \forall t'_n.T[t'_i/t_i]$ *and* $\forall t_1\leq A_1.\ldots. \forall t_n\leq A_n.U[t_i/t'_i]$ will be denoted respectively by $\forall \Gamma^-.T$ and $\forall \Gamma.U$ (where Γ is $t'_1=t_1\leq A_1,\ldots,t'_n=t_n\leq A_n$).

Fact 4.3: The closed form of a judgement J is equivalent to J, from the point of view of provability and of divergence, since it reduces to J in n $(\forall cod')$ steps, where n is the length of the environment.

The next definition extends the notion of occurrence from a type to a whole judgement, by defining the occurrence of a type in a judgement J as its occurrence on one side of the comparison of the closed form of J.

Definition 4.4 (*occurrence w.r.t. a judgement*): Given a judgement $\Gamma \vdash T \leq U$ or $\Gamma \vdash U \geq T$ (where $\Gamma = (t'_1=t_1)\leq A_1,\ldots,(t'_n=t_n)\leq A_n$) and any occurrence of a subterm in $A_1,\ldots,A_n,U$, its *occurrence w.r.t. the judgement* is its occurrence in $\forall \Gamma.U$, while for any occurrence of a subterm in T its *occurrence w.r.t. the judgement* is its occurrence in $\forall \Gamma^-.T$; in the first case we say that it occurs on the larger side of the judgement, in the second case that it occurs on the smaller side. The *valid occurrences* of a judgement are those occurrences μ such that some subterm occurs in μ w.r.t. the judgement.

Definition 4.5 (*polarity*): The polarity of an occurrence of a type in a judgement is inductively defined as follows; in the judgements:

$$(t'_1=t_1)\leq A_1,\ldots,(t'_n=t_n)\leq A_n \vdash T \leq U \qquad (t'_1=t_1)\leq A_1,\ldots,(t'_n=t_n)\leq A_n \vdash U \geq T$$

the occurrences of $A_1 \ldots A_n$ and T are *negative*, and the occurrence of U is *positive*. If the occurrence of a type $\forall t\leq A.B$ has a given polarity (positive or negative), the occurrence of B has the same polarity, while the occurrence of A has the opposite polarity.

We can now prove the first two propositions. Proposition 4.6 says that types are not created during a reduction chain, but they are just "moved around". Proposition 4.7 specifies that, when they are moved around, their polarity is preserved.

Proposition 4.6: All the new bounds inserted into the environment and all the types which are compared in a reduction chain starting from $(t'_1=t_1)\leq A_1,\ldots,(t'_n=t_n)\leq A_n \vdash T \leq U$

are similar to subterms of $A_1,\ldots, A_n, T, U$, where *T similar to U* means that T and U only differ in the names of their free and bound variables[6].

Proof: The first property is preserved by each rule: the *(∀)* rules substitute the compared types with two subterms and, in the *(∀cod)* case, add a subterm of one of the compared types to the environment; the *(exp)* rule copies into the comparison a type which is α-equal, hence similar, to a bound in the environment. The thesis follows, since being similar to a subterm is a transitive relation. □

Proposition 4.7: For each reduction sequence $\{J_i\}_{i\in I}$ (where I may be $\{0..n\}$ or ω), each type in each J_i is similar to one type appearing in J_0 with the same polarity.

Proof: This can be checked rule by rule. For example, the *(∀dom)* rule applied to a comparison $\forall t{\leq}A.B' \leq \forall t'{\leq}A'.B$ copies the bound A', which is negative since $\forall t'{\leq}A'.B$ is positive, in the environment, and all the bounds in the environment are negative by definition. The expansion rule substitutes a negative variable with a bound from the environment, negative by definition. □

Lemma 4.8: For each reduction sequence $\{J_i\}_{i\in I}$ (where I may be $\{0..n\}$ or ω), such that both a *(lhs-exp)* and a *(rhs-exp)* steps appear in the initial subsequence $\{J_i\}_{i\in\{0..m\}}$, all the new bounds inserted into the environment at any step, and all the types which are compared in a judgement J_l with $l{\geq}m$, are similar to subterms of a negative bound which appears in J_0 (i.e., this bound may either be an A_i or a bound which is a negative subterm of an A_i, of T or of U).

Proof: Any negative bound which is put in the environment by a *(∀cod)* step is similar to a negative bound in J_0 by Proposition 4.7. The rest of the proposition follows from the fact that after a *(lhs-exp)(A)* step, and before the next *(lhs-exp)* step, the left hand side of the comparison is a subterm of A, and A is similar to a bound in the environment (likewise for the right hand side). □

These propositions show that detecting rewriting divergence is only difficult because of the unlimited growth of the environment, since the comparison always regards the same (modulo similarity) pairs of types.

4.3 Variable creation in diverging judgements

The diverging sequence that we have presented always goes back to expand the same variable v_0, even though infinitely many different variables (u_i and v_i for $i\in\omega$) are created. In this section we prove that this is a feature of every diverging judgement.

Before proving this result, we have to relate variables appearing in different judgements in a precise way.

Definition 4.9 (*variable identification*): If $J \multimap J'$, and one variable in J has the same name as one variable in J', we consider them as being the same variable. If one variable t is in J' but not in J, we say that t has been *created* by the rewriting step.

New variables can be created by *(exp)(T)* steps only, and they are all and only the variables defined inside the bound T. For example, the step below creates x on the left

[6]Formally: T similar_to $U \Leftrightarrow t_1,\ldots t_n,u_1,\ldots u_n$ exist such that $\forall t_1\ldots t_n.T =_\alpha \forall u_1\ldots u_n.U$.

hand side:

$$(t=t')\leq\forall\xi.Top \vdash t \leq A. \quad \longrightarrow\triangleright (exp)(\forall x.Top)$$
$$(t=t')\leq\forall\xi.Top \vdash \forall x.Top \leq A$$

Let us now examine the evolution of the bound of a variable along a reduction chain.

Definition 4.10: A variable *t* is *properly defined* w.r.t. a judgement *J* if the bound of *t* is in a negative occurrence of *J*; otherwise *t* is *improperly defined*. The bound of a properly defined variable is its *proper bound*.

Remark 4.11: One variable *t*, along a reduction chain, evolves as follows:

- It is created inside the comparison with a given *creation bound*; it maintains that bound, with its polarity, up to the step where its definition $(\forall t)$ occupies occurrence ε on one side of the comparison.
- If the next $(\forall)$ step is $(\forall dom)$, the variable simply disappears from the judgement. If the next $(\forall)$ step is $(\forall cod)$, the variable is unified with one variable *t'* from the other side, its definition is moved from the comparison into the environment, and:
  i) If *t* was improperly defined, i.e. if *t* was defined at occurrence ε on the smaller side of the comparison, then, after the $(\forall cod)$ step, *t* changes its bound, acquiring the negative bound of *t'*, and becomes properly defined.
  ii) If *t* was already properly defined, i.e. if *t* was defined at occurrence ε on the larger side of the comparison, then *t* changes neither its bound nor the polarity of its bound.

  In both cases, in the next steps *t*, which is now defined in the environment, will remain properly defined, and its bound will no longer change.

  Hence, in a fixed rewriting chain, every variable has exactly one creation bound and at most one proper bound, which may either be its creation bound or may be acquired after being unified to a properly defined variable.

By the previous remark, the following notion of *creation bound-depth* and *proper bound-depth* is well defined, and every variable in a given chain has exactly one creation bound-depth and has either one proper bound-depth or no proper bound-depth at all.

Definition 4.12: The *depth* of a type is the length of its longest valid occurrence. In a rewriting chain of judgements, the *creation bound-depth* of a variable is the depth of its creation bound, while the *proper bound-depth* is the depth of its proper bound.

Proposition 4.13: In an infinite reduction there are an infinite number of *(lhs-exp)* steps, an infinite number of *(rhs-exp)* steps, an infinite number of *(lhs-*$\forall$*)* steps (where a $(\forall)$ step is either a $(\forall cod)$ or a $(\forall dom)$) and an infinite number of *(rhs-*$\forall$*)* steps. In an infinite reduction, infinitely many different variables are created on both sides.

Proof: There can be no infinite sequence of consecutive $(\forall)$ steps since each of them strictly decreases the dimension of the types compared. A sequence of consecutive *(exp)* steps always has the form: $\Gamma\vdash t_1\leq A \longrightarrow\triangleright \ldots \longrightarrow\triangleright \Gamma\vdash t_n\leq A \longrightarrow\triangleright \Gamma\vdash B\leq A$ with $t_n\leq B'$, $t_{n-1}\leq t_n,\ldots, t_1\leq t_2$ contained in the environment; when the environment has length *n*, at most *n* consecutive *(exp)* steps are possible. Hence any infinite chain is

formed by an infinite interleaving of finite groups of *(∀)* and *(exp)* steps.

Both an infinite number of *(lhs-exp)* and of *(rhs-exp)* steps must be performed in any infinite chain, since any *(∀)* step strictly reduces the size of both compared types. An infinite number of *(∀)* steps are performed on each side, since any sequence of *(lhs-exp)* steps is terminated by a *(lhs-∀)* step, and similarly on the right hand side.

In any infinite reduction chain, the last expansion of any sequence of expansion steps like the one exemplified above always copies a bound B with shape $\forall t \leq T.U$, since the next step is a *(∀)* step. Hence, an infinite number of variables are created on both sides of the comparison. □

Lemma 4.14: A variable with creation bound-depth n is created by expanding a variable whose proper bound-depth is at least $n+1$.

Proof: A variable t with a bound B is created by expanding a variable u whose proper bound A contains a subterm similar to $\forall t \leq B$; hence, if the depth of B is n, then the depth of A is at least $n+1$. □

Lemma 4.15: In any infinite reduction, if k variables have a creation bound satisfying a property Q, then at most $2k$ variables have a proper bound satisfying Q.

Proof: Intuitively, any creation bound may become the proper bound of at most two variables. More formally, let:

$Cre = \{t \mid A$ is the creation bound of t and $Q(A)\}$
$Pro = \{t \mid A$ is the proper bound of t and $Q(A)\}$
$C\&P = \{t \mid t \in Cre$ and the creation bound of t is proper$\}$
$UniC\&P = \{t \mid t$ is unified by a *(∀cod)* step to one variable u in $C\&P\}$

By Remark 4.11, a variable t is in Pro iff either it has been created with a proper bound satisfying Q ($t \in C\&P$) or it has been unified with a variable in such a situation ($t \in UniC\&P$). Moreover, every variable in $C\&P$ is unified to at most one variable in $UniC\&P$, hence $\#UniC\&P \leq \#C\&P$ (where $\#S$ is the cardinality of a set S). To sum up:

$$\#Pro = \#C\&P + \#UniC\&P \leq \#C\&P + \#C\&P \leq \#Cre + \#Cre = 2k.$$

Proposition 4.16: In any infinite reduction there is one variable which is expanded an infinite number of times.

Proof: By Proposition 4.13, in an infinite reduction sequence, an infinite number of different variables are created. Let n be the maximum i such that an infinite number of different variables of creation bound-depth i are created. n exists, since by Proposition 4.13 an infinite number of variables are actually created, and by Proposition 4.6 there is an upper limit to the bound-depths of all these variables. By the definition of n, there is only a finite number k of different variables with creation bound-depth greater than n. By Lemma 4.15, at most $2k$ variables may have a proper bound-depth greater than n. Since an infinite number of variables with creation bound-depth n are created, then, by Lemma 4.14, the $2k$ (or less) variables with proper bound-depth greater than n are (collectively) expanded an infinite

number of times to create these infinitely many variables, which means that at least one of the $2k$ variables is expanded an infinite number of times. □

Proposition 4.16 states that there is one variable which is expanded an infinite number of times, and Proposition 4.13 states that an infinite number of variables are created, but up to this point there is no reason to believe that this infinite number of different variables are used (i.e. appear in their scope), that their definition is moved into the environment by the *(∀cod)* rule, and that they are expanded, as happens in our diverging judgement. In Section 4.5 we will show that this is always the case.

4.4 The inversion depth of a diverging judgement

We have seen that, in every diverging judgement, there is one variable which reappears (an infinite number of times) on one side of the comparison after it has been expanded on that side. In this and in the next subsection we study how a variable may reappear. In this section we show that a minimum "inversion depth" is needed for its bound; in the next section we focus on a specific way of using the *(∀cod)* rule.

Definition 4.17 (*inversion depth, odd/even occurrences*): The *inversion depth* of an occurrence ν is the number of 0's in it; an occurrence is odd/even if its inversion depth is odd/even. The *inversion depth* of a type is the maximum *inversion depth* of all of its valid occurrences[7]. The *inversion depth* of a judgement $\Gamma \vdash A \leq B$ is the maximum inversion depth of all the valid occurrences of the judgement (Definition 4.4), i.e. the maximum between the inversion depths of $\forall \Gamma^{-}.A$ and $\forall \Gamma.B$.

Inversion refers to the fact that if we follow the path encoded by an occurrence ν along a type, each 0 in ν corresponds to a polarity inversion. The inversion depth of a judgement is a measure of its complexity, and it never increases during reduction.

Proposition 4.18: Rewriting does not increase the inversion depth of a judgement.

Proof: Suppose that n is the inversion depth of the judgement. The maximum inversion depth of a bound in the environment is then, at most, $n-1$, hence an *(exp)* step puts into the comparison a type whose depth is at most $n-1$. A *(∀cod)* rule applied to a comparison of depth n puts in the environment a bound of maximum depth $n-1$, which cannot make the inversion depth of the judgement bigger than n. Finally *(∀dom)* just decreases the inversion depth of the types compared. □

The following lemma shows that a bound with inversion depth 2 is needed to change the direction of the comparison twice.

Lemma 4.19: If a sequence of rewriting steps contains *(lhs-exp)(A)*, *(rhs-exp)(B)*, and *(lhs-exp)(C)* (in this order, but possibly separated by other steps), and if *(lhs-exp)(A)* is the last *(lhs-exp)* step before *(rhs-exp)(B)*, then the inversion depth of A is at least 2.

Proof: Let the path between two *(exp)* steps be the occurrence representing the movements made by the pointer along the compared types (Section 2.3); formally, let it be the sequence which contains one 0 (resp. one 1) for each *(∀dom)* (resp.

[7] This notion is similar to the *rank* of functional types.

(∀cod)) step performed after the first expansion and before the second one. Observe that:

a) The path between a right hand side *(exp)* and a left hand side *(exp)*, or vice versa, is always odd (i.e. it contains an odd number of 0's).

b) If *(lhs-exp)(T')* is the first left expansion which follows *(lhs-exp)(T)*, if μ is the path between *(lhs-exp)(T)* and *(lhs-exp)(T')*, then T/μ is the variable substituted by *T'*, hence μ is a valid occurrence of *T*.

Let *(lhs-exp)(C')* be the first left expansion which follows *(rhs-exp)(B)*. By (a), the path μ between *(lhs-exp)(A)* and *(lhs-exp)(C')* has an inversion depth of at least 2, since it is the concatenation of the two odd paths from *(lhs-exp)(A)* to *(rhs-exp)(B)* and from *(rhs-exp)(B)* to *(lhs-exp)(C')*; by (b) μ is a valid occurrence of *A*; hence the inversion depth of *A* is at least 2. □

Propositions 4.13, 4.18 and Lemma 4.19 together force a lower bound on the inversion depth of a diverging judgement.

Proposition 4.20: The inversion depth of a diverging judgement is at least 3.

Proof: By Proposition 4.13, any infinite chain starting from the diverging judgement contains three expansion steps satisfying the conditions of Lemma 4.19. When the first step is executed, the type *A* of Lemma 4.19 is a renamed copy of a bound in the environment; since the inversion depth of this bound is at least 2, the inversion depth of the whole judgement is at least 3. By Proposition 4.18 (depth never increases), the inversion depth of the original judgement is also at least 3. □

Proposition 4.20 gives an elementary characterization of a subset of the subtyping judgements which is decidable and expressive: types with an inversion depth strictly greater than two are, in practical use, rare[8].

The reader can check that the inversion depth of our diverging judgement is three; hence our judgement is minimal with respect to that parameter.

4.5 Regaining references to expanded variables

The key feature of diverging judgements is the existence of a variable which, after being expanded, appears back on the same side to be expanded once again, actually infinitely many times again (Proposition 4.16). We show here that this would not be possible without the unification performed by the *(∀cod)* rule, and that this unification must be exploited in quite a special way to reach this effect.

To this aim, we first define when a variable is reachable from another one, from a specific side of the comparison, or from the whole comparison. We show that this definition captures the idea of reachability, i.e. that only if a variable is reachable from the comparison, may it be expanded in some future step. Then we show that, when a variable *z* is expanded on one side of the comparison, then no references to that variable remain on that side, which implies that a reference to *z* must be reobtained in order to expand *z* once again. We finally show how *(∀cod)* must be exploited to regain that

[8]Types with a high inversion depth arise when $F_{\leq}$ is used to encode, for example, products or existential types (see [Cardelli Longo 92] [Ghelli 90] [Cardelli Martini Mitchell Scedrov 91]). However, if products and existentials are regarded as primitive type constructors, they do not add anything to the whole inversion depth of a judgement.

reference. This result is used to show, in the next section, the minimality of our judgement.

We first define the reachability relation.

Definition 4.21 (*negative free*): A variable t is *negative free* in an occurrence of a type T w.r.t. J, if a negative (w.r.t. J) free occurrence of t is inside the occurrence of type T.[9]

When the occurrence of T is understood, we just say that t is negative free in T. For example, we say that, in $t \leq Top \vdash Top \rightarrow t \leq t \rightarrow Top$, t is negative free in both $Top \rightarrow t$ and $t \rightarrow Top$.

Definition 4.22 (*reachability*): With respect to a fixed judgement J, the variable u is *immediately reachable from* a properly defined t, written $t\ R_J\ u$, iff u is negative free in the *proper* bound of t. The strict reachability relation R_J^+ is the transitive closure of immediate reachability R; $t\ R_J^*\ u$ means $t=u$ *or* $t\ R_J^+\ u$. If $t\ R_J^*\ u$ we say that t is a *reference* to u.

We are only interested in negative variables and in proper bounds, since only negative variables can be expanded, and can only be substituted by proper bounds.

We now extend the notion of reachability to the comparison of a judgement. If Γ is $(t'_1=t_1) \leq A_1 \ldots (t'_n=t_n) \leq A_n$, let $def(\Gamma)$ be the set of the variables defined in Γ, i.e. the set $\{t'_1,t_1,\ldots,t'_n,t_n\}$. For $J = \Gamma \vdash T_1 \leq T_2$ and $X \subseteq def(\Gamma)$, $ReachVars_X(J,T_i)$ contains those variables in X which are reachable from side T_i of the comparison, i.e. from some negative free variable of T_i, and $ReachVars_X(J)$ contains those variables in X which are reachable from either T_1 or T_2.

Definition 4.23: Let $J = \Gamma \vdash T_1 \leq T_2$:

$$ReachVars_X(J,T_1) =_{\text{def}} \{t \in X \mid \exists \nu \text{ even, } u \text{ free in } T_1,\ T_1/\nu = u \text{ s.t. } u\ R_J^*\ t\}$$
$$ReachVars_X(J,T_2) =_{\text{def}} \{t \in X \mid \exists \nu \text{ odd, } u \text{ free in } T_2,\ T_2/\nu = u \text{ s.t. } u\ R_J^*\ t\}$$
$$ReachVars_X(J) =_{\text{def}} ReachVars_X(J,T_1) \cup ReachVars_X(J,T2_1)$$

The name *reachability* given to the above relation is justified by the fact that no variable which is unreachable from J may be expanded in some judgement deriving from J.

Theorem 4.24: Consider a judgement $\Gamma \vdash T \leq U$ and a reduction chain $\{J_i\}_{i \in I}$ starting from it. The sequence $ReachVars_{def(\Gamma)}(\{J_i\})_{i \in I}$ is non-increasing.

Proof: Consider a $(\forall cod)$ step:

$$J_i = \Gamma, \Gamma' \vdash \forall t \leq A.B' \leq \forall t' \leq A'.B \quad \triangleright \quad J_{i+1} = \Gamma, \Gamma', t=t' \leq A' \vdash B' \leq B$$

We show that $u \in ReachVars_{def(\Gamma)}(J_{i+1})$ implies that $u \in ReachVars_{def(\Gamma)}(J_i)$. By definition, there exists w negative free in B' or in B such that $w\ R_{J_{i+1}}^*\ u$. The following cases arise:

a) $w=t'$ (or t) and $w=u$: this is impossible: u is different from t (and $u \neq t'$) since $u \in def(\Gamma)$ but $t'(t) \notin def(\Gamma)$.

[9] Formally, an occurrence ν is inside an occurrence μ when $\nu \equiv \mu.\mu'$. "Inside" refers to the fact that the path $\mu.\mu'$ leads inside the type which is reached by the path μ.

b) $w=t'$ (or t) and $w\ R_{J_{i+1}}{}^{+}\ u$: since $w=t'\ R_{J_{i+1}}{}^{+}\ u$ then there exists t'' free at an even occurrence in A' such that $t''\ R^*\ u$. Since A' is a subterm of $\forall t'\leq A'.B$, and A' is negative in J_i, $u\in ReachVars_{def(\Gamma)}(J_i)$; the same holds if $w=t$.

c) $w\neq t',t$ and $w\ R_{J_{i+1}}{}^{*}\ u$: since $w\neq t$ and $w\neq t'$, if w is negative free in either B' or B then w is negative free in $\forall t\leq A.B'$ or in $\forall t'\leq A'.B$, i.e. it is negative free on one side of the comparison of the judgement J_i, hence $u\in ReachVars_{def(\Gamma)}(J_i)$.

A similar but simpler proof can be performed for the *(∀dom)* case.

Consider now an expansion step:

$$(t'_1=t_1)\leq A_1\ldots(t'_n=t_n)\leq A_n \vdash t_i \leq B \quad \triangleright \quad \Gamma \vdash FreshNames(A_i)\leq B$$

The negative free variables of $FreshNames(A_i)$ are the negative free variables of A_i, which were already reachable through t_i, while B is not affected by the step. □

Corollary 4.25: If $t\notin ReachVars_{\{t\}}(J)$, then there exists no J' deriving from J such that an expansion step expanding t can be applied to J'.

Proof: An expansion step expanding t can be applied to J' only if one side of the comparison of J' consists of a negative free t; in this case $t\in ReachVars_{\{t\}}(J')$, hence, if J rewrites to J', $t\in ReachVars_{\{t\}}(J)$.

The next fact to prove is that, when a variable t is expanded, in the resulting judgement no reference to t remains on that side of the judgement, i.e. that the strict reachability relation is acyclic. We will actually prove a stronger property, upward well-foundedness of the reachability relation.

Lemma 4.26: The $R_J{}^{+}$ strict reachability relation on variables is upward well-founded, i.e. there is no infinite chain $\{t_i\}_{i\in\omega}$ such that, for any $i\in\omega$, $t_i\ R_J{}^{+}\ t_{i+1}$. In particular, for no t we may have $t\ R_J{}^{+}\ t$.

Proof: If $t\ R_J{}^{+}\ u$ then t is defined in the scope of u, and this is an acyclic relation. Formally, if the definition at occurrence π_t (w.r.t. J) of one variable t is in the scope of another variable u defined at occurrence π_u, then $\pi_t=\pi_u.1.\mu$ for some μ. Hence, if $|\pi|$ is the length of an occurrence, then $|\pi_u|<|\pi_t|$. $t\ R_J\ u$ implies that u is free in the bound of t, hence that the definition of t is in the scope of u, hence that $|\pi_u|<|\pi_t|$. Since $<$ is downward well-founded on integers, then $R_J{}^{+}$ is upward well-founded on variables. □

Corollary 4.27: In any infinite reduction chain $\{J_i = \Gamma_i \vdash A_i \diamond_i B_i\}_{i\in\omega}$ (where $\diamond_i$ is either $\leq$ or $\geq$) a variable t and an infinite set $I\subseteq\omega$ exist such that, for one side of the comparison (say the left hand side), for all i's in I, $t\notin ReachVars_{\{t\}}(J_i,A_i)$ and $t\in ReachVars_{\{t\}}(J_{i+1},A_{i+1})$.

Proof: By Proposition 4.16, in an infinite reduction chain one variable exists, say t, which is expanded infinitely many times, hence it is expanded infinitely many times at least on one side of the comparison, say the left hand side. Let $L=\{l \mid J_l$ reduces to J_{l+1} by expanding t on the left hand side$\}$; both l and $\{l+1 \mid l\in L\}$ are infinite, and, for all l's in L:

- $t\in ReachVars_{\{t\}}(J_l,A_l)$, since, for $l\in L$, $A_l=t$.

- $t \notin ReachVars_{\{t\}}(J_{l+1},A_{l+1})$: for $l \in L$, A_{l+1} is a renamed version of the bound of t; hence, $A_{l+1}\ R_{J_{l+1}}{}^*\ t$ would imply $t\ R_{J_{l+1}}{}^+\ t$, which is forbidden by Lemma 4.26.

For any pair of consecutive integers l and l' in L, $t \notin ReachVars_{\{t\}}(J_{l+1},A_{l+1})$ and $t \in ReachVars_{\{t\}}(J_{l'},A_{l'})$; hence, for any l, a k_l exists, with $l+1 \le k_l < l'$, such that $t \notin ReachVars_{\{t\}}(J_{k_l},A_{k_l})$ and $t \in ReachVars_{\{t\}}(J_{k_{l+1}},A_{k_{l+1}})$. The set formed by all these k_l's is an infinite set which satisfies the theorem hypothesis. □

We have formalized the intuition that in a diverging judgement there is a variable whose reference is lost and then regained, infinitely many times, by one side of the comparison. We can finally study the "fine structure" needed to regain that lost reference.

Proposition 4.28: If $J \multimap J'$ and $u \in ReachVars_X(J',A')\text{-}ReachVars_X(J,A)$, where A and A' are the left hand sides of the comparisons of J and J', then:

a) $J = \Gamma \vdash \forall t \le T.U' \le \forall t' \le T'.U$ (for some t,t',T,U)
$J' = \Gamma, t=t' \le T' \vdash U' \le U$
and J is transformed in J' by a *(lhs-∀cod)* step.

b) t is negative free in U' and u is reachable in J from a free negative variable of T'.

Proof: Consider a *(∀cod)* step:

$$J=\Gamma \vdash \forall t \le T.U' \le \forall t' \le T'.U \quad \multimap \quad J'=\Gamma, t=t' \le T' \vdash U' \le U$$

Suppose that $u \in ReachVars_X(J',U')\text{-}ReachVars_X(J,\forall t \le T.U')$. By definition, there exists w negative free in U' such that $w\ R_{J'}{}^*\ u$. The following cases arise:

a) $w \neq t$ and $w\ R_{J'}{}^*\ u$: this is impossible: since $w \neq t$, if w is negative free in U' then w is negative free in $\forall t \le T.U'$, and then $u \in ReachVars_X(J,\forall t \le T.U')$.

b) $w=t$ and $w=u$: this is impossible, since $u \in def(\Gamma)$ but $t \notin def(\Gamma)$.

c) $w=t$ and $w\ R_{J'}{}^+\ u$: this means that $t(=w)$ is negative free in U', and there is a free negative variable z in T' (the bound of t in J') such that $z\ R_{J'}{}^*\ u$, q.e.d..

We omit the simple proof of the fact that the set $ReachVars_X(J,A)$, where A is the left hand side of the comparison, cannot grow in the *(∀dom)*, *(exp)* and *(rhs-∀cod)* cases. □

The proposition above states that the only way of gaining a reference to one variable u on one side of the comparison is to unify a variable (say t) improperly (negatively) defined on that side to a variable, properly defined on the other side, t', whose bound T' refers to u. Furthermore, the variable t must appear in an even occurrence of its scope U'. We can now complete Proposition 4.13.

Proposition 4.29: In any infinite reduction an infinite number of different variables must be created, appear in their scope, have their definition moved into the environment by a *(∀cod)* step, and be expanded.

Proof: With respect to a fixed rewriting chain $C=\{J_i\}_{i \in \omega}$, we say that t is usefully-reachable from u (w.r.t. to a judgement J_i) iff $u\ R_{J_i}{}^*\ t$ and, furthermore, u is expanded in some step of C. Reasoning as in Corollary 4.27, we prove that there exist one variable t and an infinite set I such that t is not usefully-reachable from

one side, say the left hand side, of any judgement in $\{J_i\}_{i\in I}$ but is usefully-reachable from the left hand side of the judgements in $\{J_{i+1}\}_{i\in I}$. Since useful reachability implies reachability, by Proposition 4.28, for any $i\in I$, there exists one different variable t_i which appears in its scope U'_i and whose definition is moved into the environment in the rewriting step transforming J_i in J_{i+1}. t_i is also expanded in some step, by definition of useful reachability. □

To summarize, we have shown that every diverging chain uses infinitely many different variables to be able to expand one single variable infinitely many times (Propositions 4.16 and 4.29); Proposition 4.28 specifies the fine structure needed to exploit a new variable to prepare a new expansion of an already expanded variable.

4.6 The minimality of our diverging judgement

We can finally show the minimality of our judgement. More precisely, we show that, for any diverging judgement J, each occurrence of a type operator ($\forall$, t or "-") in the bound $B=\forall\xi.\text{-}\forall\psi\leq\xi.\text{-}\xi$ of the minimal diverging judgement corresponds to an occurrence of the same type operator in a bound of J, both occurrences playing the same role w.r.t. divergence.

Theorem 4.30: Any diverging judgement contains two bounds with the following structure:

$$(1)\qquad A' \;=\; E_1[\forall t.E_2[t]]$$
$$(2)\qquad B' \;=\; O[\forall t'\leq E_3[t''].U]$$

Where the $E_i[]$ are even contexts, i.e. types with a hole at an even occurrence, and $O[]$ is an odd context.

Proof: Consider a diverging chain $\{J_i=\Gamma_i\vdash A_i\leq B_i\}_{i\in\omega}$ starting from J. By Corollary 4.27, a variable u and an infinite set $I\subseteq\omega$ exist such that, for one side of the comparison (say the left hand side), $\forall\, i\in I\ u\notin ReachVars_{\{t\}}(J_i,A_i)$ and $u\in ReachVars_{\{t\}}(J_{i+1},A_{i+1})$. Let us choose a $j\in I$ such that both a *(rhs-exp)* and a *(lhs-exp)* come before the step $J_j \longrightarrow J_{j+1}$ (this is always possible by Proposition 4.13). In this way, by Lemma 4.8, we are sure that A_j and B_j are similar to two subterms of two negative bounds A' and B' appearing in J. By Proposition 4.28, A_j and B_j are two types $\forall t\leq T.U'$ and $\forall t'\leq T'.U$ such that:

(1) $\forall t\leq T.U'$ occurs negatively in J_j. Hence, it is similar to a subterm which occurs in an even occurrence of the bound A'. Moreover, t appears negatively in U', hence A' can be written as $E_1[\forall t.E_2[t]]$, where the $E_i[]$ are even contexts.

(2) $\forall t'\leq T'.U$ occurs positively in J_j. Hence, it is similar to a subterm which occurs in an odd occurrence of the bound B'. Moreover, some variable t'' appears free at a negative occurrence of T', hence B' can be written as $O[\forall t'\leq E_3[t''].U]$, where $O[]$ is an odd context and $E_3[]$ is an even context. □

We can now show that, for any diverging judgement J, each occurrence of a type operator in the bound $B=\forall\xi.\text{-}\forall\psi\leq\xi.\text{-}\xi$ corresponds to an occurrence of the same operator in some bound of J (where "-" may be substituted by any operator which inverts polarity). Here "corresponds" means that the two operators play the same role in

the *(∀cod)* step which is applied infinitely many times, according to Proposition 4.28, in order to have the variable *t* of Corollary 4.27 reappear infinitely many times in the set of variables which are reachable from one side of the comparison.

Consider the graphical representation of *B* in Figure 3.

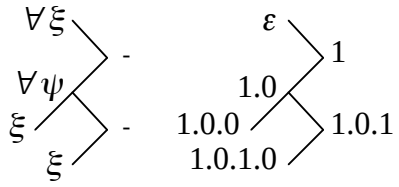

Figure 3. The subterms occurring in the bound *B*.

The quantifier ∀ξ occurring at ε is used to introduce an improperly defined variable and the occurrence of the variable ξ at 1.0.1.0 is the one which will be expanded later on; hence they correspond to the ∀*t* and *t* which must occur in even contexts in the bound *A′* according to Theorem 4.30. In the same way, occurrences 1.0 (∀ψ) and 1.0.0 (ξ) correspond to the ∀*t′* and *t″* required by condition (2) of Theorem 4.30. The "-" appearing at occurrence 1 constitutes the odd context *O[]* required by condition (2) and, finally, the "-" at occurrence 1.0.1 completes the even context surrounding the variable ξ at 1.0.1.0 as required by condition (1). Hence, any diverging judgement *J* contains the whole structure of the bound *B*, possibly split in two bounds. One may now also show, by Proposition 4.28, that not only is bound *B* minimal, but also the right hand side of the comparison in the judgement $v{\le}B \vdash v \le \forall u{\le}v.v$ is the minimal one needed to start a diverging chain.

5 Divergence and undecidability of Fbq

The type *Top* was initially defined in *Fun* to deal with both bounded and unbounded quantification with a single ∀ construct, by representing an unbounded quantification $\forall t.T$ as $\forall t{\le}Top.T$. Alternatively, following [Bruce Longo 90], we may define two different ∀ quantifiers, bounded and unbounded, thus avoiding the type *Top*. The resulting type system has been called *Fbq* by Luca Cardelli (F + Bounded Quantification), and has three different *(∀)* subtyping rules, one to compare two bounded quantifications, one to compare two unbounded quantifications, and one to perform the mixed comparison (unbounded ≤ bounded). The rules are just three different instances of the $F_{\le}$ rules; for example, the mixed comparison rule is as follows (the algorithmic subtyping rules of *Fbq* are in Appendix C):

(u-b∀≤)
$$\frac{\Gamma, t'{\le}A \vdash B'[t'/t] \le B}{\Gamma \vdash \forall t.B' \le \forall t'{\le}A.B}$$

Fbq can be seen as a sublanguage of $F_{\le}$, since the unbounded quantification of *Fbq* can be read as *Top*-bounded quantification in $F_{\le}$; $F_{\le}$ is conservative over *Fbq*, in the sense that any provable $F_{\le}$ judgement which can be written inside *Fbq* can also be proved inside *Fbq* [Ghelli 90]. The standard type-checking algorithm for *Fbq* can easily be defined by dropping the *Top* rules from our reduction system, by introducing into the environment syntax the *t=t′ type* statement to substitute *t=t′≤Top*, and by adding two

reduction rules for the unbounded-unbounded and unbounded-bounded comparisons:

$$\begin{array}{lllll} \textit{(Fbq-u}\leq\textit{b)} & \Gamma \vdash \forall t.B' \leq \forall t'\leq A'.B & \longrightarrow & \Gamma, t=t'\leq A' & \vdash B' \leq B \\ \textit{(Fbq-u}\leq\textit{u)} & \Gamma \vdash \forall t.B' \leq \forall t'.B & \longrightarrow & \Gamma, t=t' \textit{ type} & \vdash B' \leq B \end{array}$$

It is then easy to see that the diverging judgement in Section 3 also diverges in this reduction system.

We can generalize this fact by proving that type checking *Fbq* is as difficult as type checking $F_{\leq}$; more precisely, each problem may be transformed into the other one in linear time. In one direction, since $F_{\leq}$ is a consistent extension of *Fbq*, then any $F_{\leq}$ type checker can be used to type check any *Fbq* judgement. We now illustrate the other transformation, which reduces the subtype checking problem of $F_{\leq}$ to subtype checking for *Fbq*.

To this aim we define a deep-double-negation mapping $[\![\_]\!]$ which transforms $F_{\leq}$ judgements into *Fbq* judgements and preserves provability. We first define negation of *T*, written *-T*, as $\forall t\leq T.t$; note that this is different from the previous definition $\forall t\leq T.Top$.

Definition 5.1: The mapping - (*surface negation*[10]) of $F_{\leq}$ and *Fbq* types on, respectively, $F_{\leq}$ and *Fbq* types is defined as:

$$\begin{array}{lll} A\neq Top \implies & -A & = \forall t\leq A.t \\ & -Top & = \forall t.t \end{array}$$

where *t* is a fresh variable.

The mapping $[\![\_]\!]$ (*deep double negation*) of $F_{\leq}$ types and judgements on *Fbq* types and judgements negates each occurrence in the judgement twice, the only exception being bounds.

Definition 5.2: The mapping $\lceil\_\rceil$ (*internal double negation*) from $F_{\leq}$ minus $\{Top\}$ on *Fbq* is defined as:

$$\begin{array}{llll} \lceil t\rceil: & & \lceil t\rceil & = t \\ \lceil\forall\rceil: & A,B\neq Top: & \lceil \forall t\leq A.B\rceil & = \forall t\leq \lceil A\rceil.\text{--}\lceil B\rceil \\ & A\neq Top: & \lceil \forall t\leq A.Top\rceil & = \forall t\leq \lceil A\rceil.\text{--}Top \\ & B\neq Top: & \lceil \forall t\leq Top.B\rceil & = \forall t.\text{--}\lceil B\rceil \\ & & \lceil \forall t\leq Top.Top\rceil & = \forall t.\text{--}Top \end{array}$$

The mapping $[\![\_]\!]$ from $F_{\leq}$ to *Fbq* is defined as:

$$\begin{array}{lll} A\neq Top: & [\![A]\!] & = \text{--}\lceil A\rceil \\ & [\![Top]\!] & = \text{--}Top \end{array}$$

These mappings are extended to judgements in the natural way:

- $\lceil\Gamma\rceil$: $t=t'\leq Top$ becomes $t=t'$ *type*, while $t=t'\leq A$ $(A\neq Top)$ becomes $t=t'\leq\lceil A\rceil$
- $[\![\Gamma\vdash A\leq B]\!] = \lceil\Gamma\rceil \vdash [\![A]\!] \leq [\![B]\!]$ (note that Γ is *internally* double negated)
- $\lceil\Gamma\vdash A\leq B\rceil = \lceil\Gamma\rceil \vdash \lceil A\rceil \leq \lceil B\rceil$ (well defined only if $A\neq Top$ and $B\neq Top$)

We want to prove that any $F_{\leq}$ subtyping judgement *J* is provable if and only if $[\![J]\!]$ is provable in *Fbq*. We first need some lemmas.

[10]No (conscious) logical intuition is hidden behind the name *negation*.

Lemma 5.3: $\Gamma \vdash -A \leq -B$ reduces to $\Gamma \vdash A \geq B$ and to *true*.

Lemma 5.4: If $A \neq B$ and A and B are different from *Top*, then $\Gamma \vdash [\![A]\!] \leq [\![B]\!]$ reduces to $\Gamma \vdash (A] \leq (B]$ and to *true*.

Lemma 5.5: $\Gamma \vdash [\![A]\!] \leq [\![Top]\!]$ is provable.

Proof: If $A \neq Top$, $\Gamma \vdash [\![A]\!] \leq [\![Top]\!]$ is equal to

$\Gamma \vdash --(A] \leq --Top$

which reduces as follows (we ignore some trivial successful branches):

$\Gamma \vdash --(A] \leq --Top$ $\quad\longrightarrow\triangleright$
$\Gamma \vdash -(A] \geq -Top$ $\quad=$
$\Gamma \vdash \forall u \leq (A].u \geq \forall u'.u'$ $\quad\longrightarrow\triangleright$
$\Gamma, u=u' \leq (A] \vdash u \geq u'$ $\quad\longrightarrow\triangleright$
true. □

Lemma 5.6: $\Gamma \vdash [\![Top]\!] \leq [\![A]\!]$ is not provable if $A \neq Top$.

Proof: If $A \neq Top$, then $\Gamma \vdash [\![Top]\!] \leq [\![A]\!]$ is equal to

$\Gamma \vdash --Top \leq --(A]$

which reduces as follows (we ignore some trivial successful branches):

$\Gamma \vdash -Top \geq -(A]$ $\quad=$
$\Gamma \vdash \forall u'.u' \geq \forall u \leq (A].u$ $\quad\longrightarrow\triangleright$
false. □

Theorem 5.7: For any judgement J, J is provable in $F_{\leq}$ if and only if $[\![J]\!]$ is provable in *Fbq*.

Proof: We prove that $\vdash_{\leq} J \Rightarrow \vdash_{bq} [\![J]\!]$ by induction on the length of the longest reduction chain starting from J (this maximum exists since $\vdash_{\leq} J$), and prove $\vdash_{\leq} J \Leftarrow \vdash_{bq} [\![J]\!]$ by induction on the length of the longest reduction chain starting from $[\![J]\!]$. We work by case analysis on the shape of the types compared in J; we only report the interesting cases $t \leq A$ and $\forall t \leq A.B' \leq \forall t \leq A'.B$. For each of these cases we simply show that J reduces to a set of non trivially provable judgements $J_1,\ldots,J_n$ iff $[\![J]\!]$ reduces to a set of non trivially provable judgements $J'_1,\ldots,J'_n$ such that J'_i is either $(J_i]$, or $[\![J_i]\!]$. Then $\vdash_{\leq} J \Leftrightarrow \vdash_{\leq} J_1,\ldots,J_n \Leftrightarrow$by induction $\vdash_{bq} J'_1,\ldots,J'_n \Leftrightarrow \vdash_{bq} [\![J]\!]$.

We present the reduction chains for some interesting cases with no further comment.

$J = \Gamma, t=t' \leq A, \Gamma' \vdash t \leq C$ *with* $\not\vdash C \approx t$, $C \neq Top$, $A \neq Top$.

$J = \quad \Gamma, t=t' \leq A, \Gamma' \vdash t \leq C \quad\longrightarrow\triangleright (exp) \quad \Gamma \vdash A \leq C$

$[\![J]\!] = (\Gamma], t=t' \leq (A], (\Gamma'] \vdash --t \leq --(C] \quad\longrightarrow\triangleright *(\forall dom')$
$(\Gamma], t=t' \leq (A], (\Gamma'] \vdash t \leq (C] \quad\longrightarrow\triangleright (exp)$
$(\Gamma], t=t' \leq (A], (\Gamma'] \vdash (A] \leq (C]$

$J = \Gamma,\ t=t' \le A,\ \Gamma' \vdash\ t \le C$ with $\not\vdash C \approx t,\ C \neq Top,\ A = Top$: in this case we have to prove that neither J nor $[\![J]\!]$ are provable (proof omitted).

$J = \Gamma \vdash\ \forall t \le A.B' \le \forall t' \le A'.B$ with $A \neq Top$ and $A' \neq Top$

$$
\begin{array}{lll}
J = & \Gamma \vdash\ \forall t \le A.B' \le \forall t' \le A'.B & \multimap \\
 & (\forall dom)\ \Gamma \vdash A \ge A' & \text{and } (\forall cod)\ \Gamma,\ t=t' \le A' \vdash B' \le B
\end{array}
$$

$$
\begin{array}{lll}
[\![J]\!] = & \lceil \Gamma \rceil \vdash\ [\![\forall t \le A.B']\!] & \le\ [\![\forall t' \le A'.B]\!] \quad = \\
 & \lceil \Gamma \rceil \vdash\ -\!-\forall t \le \lceil A \rceil.[\![B']\!] & \le\ -\!-\forall t' \le \lceil A' \rceil.[\![B]\!] \quad \multimap^{*} (\forall dom') \\
 & \lceil \Gamma \rceil \vdash\ \forall t \le \lceil A \rceil.[\![B']\!] & \le\ \forall t' \le \lceil A' \rceil.[\![B]\!] \quad \multimap \\
 & (\forall dom)\ \lceil \Gamma \rceil \vdash \lceil A \rceil \ge \lceil A' \rceil & \text{and } (\forall dom)\ \lceil \Gamma \rceil,\ t=t' \le \lceil A' \rceil \vdash [\![B']\!] \le [\![B]\!]
\end{array}
$$

$J = \Gamma\ \vdash\ \forall t \le Top.B' \le \forall t' \le A'.B$ *with* $A' \neq Top$

$$
\begin{array}{lll}
J = & \Gamma \vdash\ \forall t \le Top.B' \le \forall t' \le A'.B & \multimap (\forall\ cod) \\
 & \Gamma,\ t=t' \le A' \vdash B' \le B &
\end{array}
$$

$$
\begin{array}{lll}
[\![J]\!] = & \lceil \Gamma \rceil \vdash\ [\![\forall t \le Top.B']\!] & \le\ [\![\forall t' \le A'.B]\!] \quad = \\
 & \lceil \Gamma \rceil \vdash\ -\!-\forall t.[\![B']\!] & \le\ -\!-\forall t' \le \lceil A' \rceil.[\![B]\!] \quad \multimap^{*} (\forall dom') \\
 & \lceil \Gamma \rceil \vdash\ \forall t.[\![B']\!] & \le\ \forall t' \le \lceil A' \rceil.[\![B]\!] \quad \multimap (\forall cod') \\
 & \lceil \Gamma \rceil,\ t=t' \le \lceil A' \rceil \vdash [\![B']\!] \le [\![B]\!]. \quad \square &
\end{array}
$$

Theorem 5.7 completes the proof that subtype checking of $F_{\le}$ can be reduced to subtype checking of Fbq and vice-versa. A proof of this fact based on a more complex translation, also translating $F_{\le}$ terms to Fbq terms, was previously suggested by Luca Cardelli (personal communication). Note that $F_{\le}$ and Fbq subtype checking are also equivalent from the point of view of complexity, since the mapping $[\![J]\!]$ can be executed in linear time and increases the size of J only by a constant factor.

It may be interesting to know that, in [Katiyar Sankar 92], Fbq was proved to become decidable as soon as the mixed bound comparison *(Fbq-u≤b)* is forbidden.

6 Conclusions

We have shown that the standard type-checking algorithm of system $F_{\le}$ is only a semidecision procedure, by presenting a subtype judgement which makes it diverge. The divergence result was very surprising for the author, who shared the common belief that the standard algorithm was a decision procedure. Whilst the paper was being written, the author communicated this judgement to Benjamin Pierce, who used it to encode two-register Turing machines as $F_{\le}$ subtyping judgements, proving that the problem is undecidable, in sharp contrast with the common belief that type-checking $F_{\le}$ is "easy" [Pierce 93].

We have given a set of results about the nature of judgements which make the algorithm diverge. These results can be used to prove decidability or undecidability for variations of $F_{\le}$, to design decidable versions of the system, and to characterize interesting decidable subsystems of $F_{\le}$. These results have been used to support the claim that undecidability of $F_{\le}$ may not be a problem of practical concern, since divergence of type checking is limited to judgements with a very peculiar and unnatural structure [Curien Ghelli 93,

Ghelli Pierce 92]. Similarly, they have been used to support the claim that the non-conservativity of recursive types w.r.t. $F_{\leq}$ subtyping is not a practical problem, since this non-conservativity is limited to diverging judgements [Ghelli 93a].

Acknowledgments

I gratefully thank Luca Cardelli for many insights into the problem of decidability of type checking for the language $F_{\leq}$. The work on type-checking algorithms for the Galileo and Fibonacci object-oriented database programming languages, in the project led at Pisa University by Antonio Albano, provided the motivations for this study. Special thanks to Pierre-Louis Curien who provided a great deal of help and suggestions during many phases of this work, and to the anonymous referee who provided many useful and constructive comments.

References

[Bruce 91] K. B. Bruce, "The equivalence of two semantic definitions for inheritance in object-oriented languages", in *Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, 1991.

[Bruce 92] K. B. Bruce, "*A paradigmatic object-oriented language: Design, static typing and semantics*", Technical Report CS-92-01, Williams College, 1992.

[Bruce 93] K. B. Bruce, "Safe type checking in a statically typed object-oriented programming language", in *POPL '93*, 1993.

[Bruce Longo 90] K. B. Bruce and G. Longo, "A Modest Model of Records, Inheritance and Bounded Quantification", *Information & Computation*, 87(1/2), 196-240, 1990.

[Canning et al 89] P. Canning, W. Cook, W. Hill, J.C. Mitchell, and W. Olthoff, "F-bounded quantification for object-oriented programming", in *Functional Programming and Computer Architecture*, 273-280, 1989.

[Canning Hill Olthoff 88] P. Canning, W. Hill, and W. Olthoff, "*A kernel language for object oriented programming*", Technical Report STL-88-21, HP Labs, 1988.

[Cardelli Longo 92] L. Cardelli and G. Longo, "A semantic basis for Quest", *Journal of Functional Programming*, 1 (4), 417-458, 1992.

[Cardelli Martini Mitchell Scedrov 91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov, "An extension of system F with subtyping", *Intl. Conference on Theoretical Aspects of Computer Software*, Sendai, Japan, LNCS 526, 1991. To appear in Information & Computation.

[Cardelli Mitchell 91] L. Cardelli and J.C. Mitchell, "Operations on Records", *Mathematical Structures in Computer Science*, 1 (1), 3-48, 1991.

[Cardelli Wegner 85] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism", *ACM Computing Surveys*, 17 (4), 1985.

[Castagna Pierce 94] G. Castagna and B. Pierce, "*Decidable Bounded Quantification*", in *POPL '94*, 1994.

[Cook 89] W. Cook, "*A Denotational Semantics of Inheritance*", PhD Thesis, Brown University, 1989.

[Curien Ghelli 92] P.-L. Curien and G. Ghelli, "Coherence of Subsumption in $F_{\leq}$, Minimum Typing and Type Checking", *Mathematical Structures in Computer Science*, 2(1), 1992.

[Curien Ghelli 93] P.-L. Curien and G. Ghelli, "Confluence and decidability of $\beta\eta top_{\leq}$ reduction in $F_{\leq}$", *Information & Computation*, to appear.

[Danforth Tomlinson 88] S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Computing Surveys*, 20 (1), 29-72, 1988.

[DeBruijn 72] N.G. De Bruijn, "Lambda Calculus Notation Without Nameless Dummies, a Tool for Automatic Formula Manipulation", *Indag. Math.*, 34, 381-392, 1972.

[Ghelli 90] G. Ghelli, "*Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*", PhD Thesis, TD-6/90, Dipartimento di Informatica dell'Università di Pisa, Italy, 1990.

[Ghelli 91] G. Ghelli, "Modelling features of object-oriented languages in second order functional languages with subtypes", in *Foundations of Object-Oriented Languages*, J.W. de Bakker, W.P. de Roever, G.Rozenberg (Eds.), LNCS 489, 311-340, 1991.

[Ghelli 93a] G. Ghelli, "Recursive types are not conservative over $F_{\leq}$", *Intl. Conf. on Typed Lambda Calculus and Applications* (TLCA), Utrecht, The Netherlands, 1993.

[Ghelli 93b] G. Ghelli, "*Divergence of $F_{\leq}$ type checking*", Technical Report 5/93, University of Pisa, Dipartimento di Informatica, March 1993.

[Ghelli Pierce 92] G. Ghelli and B. Pierce, "*Bounded Existentials and Minimal Typing*", manuscript, June 1992, available from the authors.

[Girard 72] J.Y. Girard, "*Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*", Thèse de Doctorat d'Etat, Paris, 1972.

[Gunter Mitchell 93] C. Gunter and J.C. Mitchell, "*Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*", The MIT Press, 1993, to appear.

[Hofmann Pierce 94] M. Hofmann and B. Pierce, "*A unifying type-theoretic framework for objects*", in *Symposium on Theoretical Aspects of Computer Science*, 1994.

[Katiyar Sankar 92] D. Katiyar and S. Sankar, "Completely Bounded Quantification is Decidable", in *ACM SIGPLAN Workshop on ML and its Applications*, June, 1992.

[Mitchell 90] J.C. Mitchell, "Towards a typed foundation for method specialization and inheritance", in *POPL '90*, 1990.

[Mitchell et al 91] J.C. Mitchell, S. Meldal, and N. Madhav, "An extension of standard ML modules with subtyping and inheritance", in *POPL '91*, 1991.

[Pierce 93] B. Pierce, "Bounded Quantification is Undecidable", *Information & Computation*, to appear. Also in [Gunter Mitchell 93].

[Pierce Turner 93] B. Pierce and D.N. Turner, "Object-Oriented Programming Without Recursive Types", in *POPL '93*, 1993.

[Reynolds 74] J.C. Reynolds, "Towards a theory of type structure", in *Paris Colloquium in Programming*, LNCS 19, 408-425, 1974.

Appendix A: The System $F_{\leq}$.

Syntax: $A ::= t \mid Top \mid A \rightarrow A \mid \forall t \leq A.\ A$

$a ::= x \mid top \mid \lambda x{:}A.\ a \mid a(a) \mid \Lambda t \leq A.\ a \mid a\{A\}$

Environments (sequences whose individual components have the form $x{:}A$ or $t \leq A$):

($\varnothing$env) $() \quad env$

(≤env) $\dfrac{\Gamma \ env \quad \Gamma \vdash A \ type \quad t \notin \Gamma}{\Gamma,\ t \leq A \quad env}$[11]

(: env) $\dfrac{\Gamma \ env \quad \Gamma \vdash A \ type \quad x \notin \Gamma}{\Gamma, x{:}A \quad env}$

Types:

(VarForm) $\dfrac{\Gamma, t \leq A, \Gamma' \ env}{\Gamma, t \leq A, \Gamma' \vdash t \quad type}$

(TopForm) $\dfrac{\Gamma \ env}{\Gamma \vdash Top \quad type}$

(→Form) $\dfrac{\Gamma \vdash A \ type \quad \Gamma \vdash B \ type}{\Gamma \vdash A \rightarrow B \quad type}$

(∀Form) $\dfrac{\Gamma, t \leq A \vdash B \ type \quad t \notin \Gamma}{\Gamma \vdash \forall t \leq A.\ B \quad type}$

Subtypes:

(Var≤) $\dfrac{\Gamma, t \leq A, \Gamma' \ env}{\Gamma, t \leq A, \Gamma' \vdash t \leq A}$

(Top≤) $\dfrac{\Gamma \vdash A \ type}{\Gamma \vdash A \leq Top}$

(→≤) $\dfrac{\Gamma \vdash A \leq A' \quad \Gamma \vdash B \leq B'}{\Gamma \vdash A' \rightarrow B \leq A \rightarrow B'}$

(∀≤) $\dfrac{\Gamma \vdash A \leq A' \quad \Gamma, t' \leq A \vdash B[t'/t] \leq B' \quad t \notin \Gamma}{\Gamma \vdash \forall t \leq A'.B \leq \forall t' \leq A.B'}$

(Id≤) $\dfrac{\Gamma \vdash t \ type}{\Gamma \vdash t \leq t}$

(Trans≤) $\dfrac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C}$

Expressions:

(Var) $\dfrac{\Gamma, x{:}A, \Gamma' \ env}{\Gamma, x{:}A, \Gamma' \vdash x{:}\ A}$

(Top) $\dfrac{\Gamma \ env}{\Gamma \vdash top{:}\ Top}$

(→ Intro) $\dfrac{\Gamma, x{:}A \vdash b{:}\ B \quad x \notin \Gamma}{\Gamma \vdash \lambda x{:}A.b{:}\ A \rightarrow B}$

(→ Elim) $\dfrac{\Gamma \vdash f{:}\ A \rightarrow B \quad \Gamma \vdash a{:}\ A}{\Gamma \vdash f(a){:}\ B}$

(∀ Intro) $\dfrac{\Gamma, t \leq A \vdash b[t/t']{:}\ B \quad t \notin \Gamma}{\Gamma \vdash \Lambda t' \leq A.b{:}\ \forall t \leq A.B}$

(∀ Elim) $\dfrac{\Gamma \vdash f{:}\ \forall t \leq A.\ B \quad \Gamma \vdash A' \leq A}{\Gamma \vdash f\{A'\}{:}\ B[t \leftarrow A']}$

(Subsump) $\dfrac{\Gamma \vdash a{:}\ A \quad \Gamma \vdash A \leq B}{\Gamma \vdash a{:}\ B}$

[11] $t \notin \Gamma$ means that t is not bounded by any element $t \leq A$ of Γ; similarly for $x \notin \Gamma$.

Appendix B: The algorithmic rules and the rewriting rules

Environments as functions from variables to types

$\Gamma(t)$ *(the bound of t in* Γ*)*: $(\Gamma, t{\leq}A, \Gamma')(t) =_{def} A$

$\Gamma^*(T)$ *(the minimum non variable supertype of T in* Γ*)*:

$$\Gamma^*(T) =_{def} \begin{cases} T & \text{if } T \text{ is not a type variable} \\ \Gamma^*(U) & \text{if } T{=}t \text{ and } \Gamma(t){=}U \end{cases}$$

Expressions:

(AlgVar) $\dfrac{\Gamma, x{:}A, \Gamma' \ env}{\Gamma, x{:}A, \Gamma' \vdash x{:}\ A}$ *(AlgTop)* $\dfrac{\Gamma \ env}{\Gamma \vdash top{:}\ Top}$

(AlgAbs) $\dfrac{\Gamma, x{:}\ A \vdash b{:}\ B \quad x{\notin}\Gamma}{\Gamma \vdash \lambda x{:}A.b{:}\ A{\to}B}$ *(AlgAbs2)* $\dfrac{\Gamma, t \leq A \vdash b{:}\ B \quad t{\notin}\Gamma}{\Gamma \vdash \Lambda t{\leq}A.b{:}\ \forall t{\leq}A.B}$

(AlgApp) $\dfrac{\Gamma \vdash f{:}\ T \quad \Gamma^*(T) = A{\to}B \quad \Gamma \vdash a{:}\ A' \quad \Gamma \vdash A'{\leq}A}{\Gamma \vdash f(a){:}\ B}$

(AlgApp2) $\dfrac{\Gamma \vdash f{:}\ T \quad \Gamma^*(T) = \forall t{\leq}A.B \quad \Gamma \vdash A' \leq A}{\Gamma \vdash f\{A'\}{:}\ B[t{\leftarrow}A']}$

Subtypes:

(AlgId≤) $\dfrac{\Gamma \vdash t \ type}{\Gamma \vdash t{\leq}t}$ *(AlgTop≤)* $\dfrac{A{\neq}Top \quad \Gamma \vdash A \ type}{\Gamma \vdash A{\leq}Top}$

(AlgTrans≤) $\dfrac{A{\neq}t, A{\neq}Top \quad \Gamma \vdash \Gamma(t){\leq}A}{\Gamma \vdash t{\leq}A}$

(Alg→≤) $\dfrac{\Gamma \vdash A'{\leq}A \quad \Gamma \vdash B'{\leq}B}{\Gamma \vdash A{\to}B' \leq A'{\to}B}$ *(Alg∀≤)* $\dfrac{\Gamma \vdash A'{\leq}A \quad \Gamma, t'{\leq}A' \vdash B'[t'/t]{\leq}B \quad t{\notin}\Gamma}{\Gamma \vdash \forall t{\leq}A.B' \leq \forall t'{\leq}A'.B}$

Rewriting rules for the ≤ case (for the ≥ case, exchange ≤ with ≥ and vice versa):

(top)			$\Gamma \vdash A'{\leq}Top$	$\longrightarrow$	*true*
(varId)	$\Gamma{\vdash}t{\approx}u$	$\Rightarrow$	$\Gamma \vdash u \leq t$	$\longrightarrow$	*true*
(exp)	$\Gamma{\nvdash}C{\approx}u, C{\neq}Top$	$\Rightarrow$	$\Gamma \vdash u \leq C$	$\longrightarrow$	$\Gamma \vdash FreshNames(\Gamma(u)) \leq C$
(∀dom)			$\Gamma \vdash \forall t{\leq}A.B' \leq \forall t'{\leq}A'.B$	$\longrightarrow$	$\Gamma \vdash A \geq A'$
(∀cod)			$\Gamma \vdash \forall t{\leq}A.B' \leq \forall t'{\leq}A'.B$	$\longrightarrow$	$\Gamma, (t{=}t'){\leq}A' \vdash B' \leq B$
(false)	*nothing else applies*	$\Rightarrow$	$\Gamma \vdash A \leq B$	$\longrightarrow$	*false*

The ≈ relation is the minimal relation such that: $\Gamma,(t{=}u){\leq}A,\Gamma' \vdash t{\approx}u, u{\approx}t, t{\approx}t, u{\approx}u$.

Appendix C: Algorithmic subtype rules for Fbq

(Id≤), (Trans≤), (→≤) plus:

(b-b∀≤) $\dfrac{\Gamma \vdash A'{\leq}A \quad \Gamma, t'{\leq}A' \vdash B'[t'/t]{\leq}B \quad t{\notin}\Gamma}{\Gamma \vdash \forall t{\leq}A.B' \leq \forall t'{\leq}A'.B}$ *(u-u∀≤)* $\dfrac{\Gamma, t' \ type \vdash B'[t'/t]{\leq}B \quad t{\notin}\Gamma}{\Gamma \vdash \forall t.B' \leq \forall t'.B}$

(u-b∀≤) $\dfrac{\Gamma, t'{\leq}A \vdash B'[t'/t] \leq B \quad t{\notin}\Gamma}{\Gamma \vdash \forall t.B' \leq \forall t'{\leq}A.B}$

Rewriting rules: as above, minus *(top)*, plus:

(Fbq-u≤b)	$\Gamma \vdash \forall t.B' \leq \forall t'{\leq}A'.B$	$\longrightarrow$	$\Gamma, t{=}t'{\leq}A'$	$\vdash B' \leq B$
(Fbq-u≤u)	$\Gamma \vdash \forall t.B' \leq \forall t'.B$	$\longrightarrow$	$\Gamma, t{=}t' \ type$	$\vdash B' \leq B$