Check for updates

Bounded Quantification is Undecidable

Benjamin C. Pierce
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
bcp@cs.cmu.edu

Abstract

$F_{\leq}$ is a typed λ-calculus with subtyping and bounded second-order polymorphism. First proposed by Cardelli and Wegner, it has been widely studied as a core calculus for type systems with subtyping.

Curien and Ghelli proved the partial correctness of a recursive procedure for computing minimal types of $F_{\leq}$ terms and showed that the termination of this procedure is equivalent to the termination of its major component, a procedure for checking the subtype relation between $F_{\leq}$ types. This procedure was thought to terminate on all inputs, but the discovery of a subtle bug in a purported proof of this claim recently reopened the question of the decidability of subtyping, and hence of typechecking.

This question is settled here in the negative, using a reduction from the halting problem for two-counter Turing machines to show that the subtype relation of $F_{\leq}$ is undecidable.

1 Introduction

The notion of *bounded quantification* was introduced by Cardelli and Wegner [16] in the language *Fun*. Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [11, 30], *Fun* integrated Girard-Reynolds polymorphism [25, 33] and Cardelli's first-order calculus of subtyping [7, 8].

Fun and its relatives have been studied extensively by programming language theorists and designers. Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [9]. Curien and Ghelli [20, 23] address a number of syntactic properties of $F_{\leq}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [3], Martini [29], Breazu-Tannen, Coquand, Gunter, and Scedrov [1], Cardone [17], Cardelli and Longo [13], Cardelli, Martini, Mitchell, and Scedrov [14], and Curien and Ghelli [20, 21]. $F_{\leq}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [15], Bruce [2], Cardelli [12], and Canning, Cook, Hill, Olthoff, and Mitchell [5]; an extension with intersection types [19, 34] is the subject of the present author's Ph.D. thesis [32]. Bounded quantification also plays a key role in Cardelli's programing language *Quest* [10, 13] and in the *Abel* language developed at HP Labs [4, 5, 6, 18].

The original *Fun* was simplified by Bruce and Longo [3], and again by Curien and Ghelli [20]. Curien and Ghelli's formulation, called *minimal Bounded Fun* or $F_{\leq}$ ("F sub"), is the one considered here.

Like other second-order λ-calculi, the terms of $F_{\leq}$ include variables, abstractions, applications, type abstractions, and type applications, with the refinement that each type abstraction gives a *bound* for the type variable it introduces and each type application must satisfy the constraint that the argument type is a *subtype* of the bound of the polymorphic function being applied. The well-typed terms of $F_{\leq}$ are defined by means of a collection of rules (summarized in Figure 1) for inferring statements of the form $\Gamma \vdash e \in \tau$ ("e has type τ in context Γ"). Variables, abstractions, and applications have the rules familiar from other λ-calculi (rules VAR, ABS, and APP). Type abstractions (rule TABS) declare a bound, with respect to the subtype relation, for the variable they introduce; they are checked by moving this assumption into the context and checking the body of the abstraction under the enriched set of assumptions. Type applications (rule TAPP) check that the type being passed as a parameter is indeed a subtype of the bound of the polymorphic value in the function position. Finally, like other λ-calculi with subtyping, $F_{\leq}$

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 089791-453-8/92/0001/0305 $1.50

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(VAR)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e \in \tau}{\Gamma \vdash \lambda x{:}\sigma.\ e \in \sigma{\to}\tau} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e_1 \in \sigma{\to}\tau \qquad \Gamma \vdash e_2 \in \sigma}{\Gamma \vdash e_1\, e_2 \in \tau} \qquad \text{(APP)}$$

$$\frac{\Gamma, \alpha{\le}\theta \vdash e \in \tau}{\Gamma \vdash \Lambda\alpha{\le}\theta.\ e \in \forall\alpha{\le}\theta.\ \tau} \qquad \text{(TABS)}$$

$$\frac{\Gamma \vdash e \in \forall\alpha{\le}\theta.\ \tau \qquad \Gamma \vdash \sigma \le \theta}{\Gamma \vdash e[\sigma] \in \{\sigma/\alpha\}\tau} \qquad \text{(TAPP)}$$

$$\frac{\Gamma \vdash e \in \sigma \qquad \Gamma \vdash \sigma \le \tau}{\Gamma \vdash e \in \tau} \qquad \text{(SUB)}$$

Figure 1: Typing rules

$$\Gamma \vdash \tau \le \tau \qquad \text{(REFL)}$$

$$\frac{\Gamma \vdash \tau_1 \le \tau_2 \qquad \Gamma \vdash \tau_2 \le \tau_3}{\Gamma \vdash \tau_1 \le \tau_3} \qquad \text{(TRANS)}$$

$$\Gamma \vdash \sigma \le \mathit{Top} \qquad \text{(TOP)}$$

$$\Gamma \vdash \alpha \le \Gamma(\alpha) \qquad \text{(TVAR)}$$

$$\frac{\Gamma \vdash \tau_1 \le \sigma_1 \qquad \Gamma \vdash \sigma_2 \le \tau_2}{\Gamma \vdash \sigma_1{\to}\sigma_2 \le \tau_1{\to}\tau_2} \qquad \text{(ARROW)}$$

$$\frac{\Gamma \vdash \tau_1 \le \sigma_1 \qquad \Gamma, \alpha{\le}\tau_1 \vdash \sigma_2 \le \tau_2}{\Gamma \vdash \forall\alpha{\le}\sigma_1.\ \sigma_2 \le \forall\alpha{\le}\tau_1.\ \tau_2} \qquad \text{(ALL)}$$

Figure 2: Subtyping rules

includes a rule of *subsumption*, which allows the type of a term to be promoted to any supertype (rule SUB).

The rules TAPP and SUB rely on a separately axiomatized subtype relation $\Gamma \vdash \sigma \le \tau$ ("σ is a subtype of τ under assumptions Γ"). This relation, which forms our main object of study, is summarized in Figure 2. Subtyping is both reflexive and transitive (rules REFL and TRANS). Every type is a subtype of a maximal type called *Top* (rule TOP). Type variables are subtypes of their bounds (rule TVAR). The subtype relation between arrow types is contravariant in their left-hand sides and covariant in their right-hand sides (rule ARROW). Similarly, subtyping of quantified types is contravariant in their bounds and covariant in their bodies (rule ALL).

The last rule deserves a closer look, since it is the primary cause of the difficulties we will be discussing for the rest of the paper. Intuitively, it reads as follows. A type $\tau \equiv \forall\alpha{\le}\tau_1.\ \tau_2$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of τ_1 to instances of τ_2. If τ_1 is a subtype of σ_1, then the domain of τ is smaller than that of $\sigma \equiv \forall\alpha{\le}\sigma_1.\ \sigma_2$, so τ is a weaker constraint and describes a larger collection of polymorphic values. Moreover, if, for each type θ that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $\theta \le \tau_1$), the θ-instance of σ_2 is a subtype of the θ-instance of τ_2, then τ is a "pointwise weaker" constraint and again describes a larger collection of polymorphic values.

Though semantically appealing, this rule creates serious problems for reasoning about the subtype relation. In a quantified type $\forall\alpha{\le}\sigma_1.\ \sigma_2$, instances of α in σ_2 are naturally thought of as being bounded by their lexically declared bound σ_1. But this connection is destroyed by the second premise of the quantifier subtyping rule: when $\forall\alpha{\le}\sigma_1.\ \sigma_2$ is compared to $\forall\alpha{\le}\tau_1.\ \tau_2$, instances of α in *both* σ_2 and τ_2 are bounded by τ_1 in the premise $\Gamma, \alpha{\le}\tau_1 \vdash \sigma_2 \le \tau_2$. As we shall see, this "re-bounding" behavior is powerful enough to allow undecidable problems to be encoded as subtyping statements.

Cardelli and Wegner's definition of *Fun* [16] used a weaker quantifier subtyping rule in which $\forall\alpha{\le}\sigma_1.\ \sigma_2$ is a subtype of $\forall\alpha{\le}\tau_1.\ \tau_2$ only when σ_1 and τ_1 are identical. (This variant can easily be shown to be decidable.) Later authors, including Cardelli, have chosen to work with the more powerful formulation given here.

Curien and Ghelli used a proof-normalization argument to show that $F_{\le}$ typechecking is *coherent* — that is, that all derivations of a statement $\Gamma \vdash e \in \tau$ have the same meaning, under certain assumptions about the semantic interpretation function. One corollary of their proof is the soundness and completeness of a natural syntax-directed procedure for computing minimal typings of $F_{\le}$ terms, with a subroutine for checking the subtype relation; the same procedure had been developed by the group at Penn and by Cardelli for use in his *Quest* typechecker [26]. The termination of Curien and Ghelli's typechecking procedure is equivalent to the termination of the subtyping algorithm. Ghelli, in his Ph.D. thesis [23], gave a proof of termination; unfortunately, this proof was later discovered — by Curien and Reynolds, independently — to contain a subtle mistake (see [31]). In fact, Ghelli soon realized that there are inputs for which the subtyping algorithm does *not* terminate [24]. Worse yet, these cases are not amenable to any simple form of cycle detection: when presented

with one of them, the algorithm would generate an infinite sequence of recursive calls with larger and larger contexts. This discovery reopened the question of the decidability of $F_{\leq}$.

The undecidability result presented here began as an attempt to formulate a more refined algorithm capable of detecting the kinds of divergence that could be induced in the simpler one. A series of partial results about decidable subsystems eventually led to the discovery of a class of input problems for which increasing the size the input by a constant factor would increase the search depth of a *succeeding* execution of the algorithm by an exponential factor. Besides dispelling previous intuitions about why the problem ought to be decidable, this construction suggested a trick for encoding natural numbers, from which it was a short step to an encoding of two-counter Turing machines.

After formally defining the $F_{\leq}$ subtype relation (Section 2), reviewing Curien and Ghelli's subtyping algorithm (Section 3), and presenting an example where the algorithm fails to terminate (Section 4), we identify a fragment of $F_{\leq}$ that forms a convenient target for the reductions to follow (Sections 5 and 6). The main result is then presented in two steps. We first define an intermediate abstraction, called *rowing machines* (Section 7); these bridge the gap between $F_{\leq}$ subtyping problems and two-counter machines by retaining the notions of bound variables and substitution from $F_{\leq}$ while introducing a computational abstraction with a finite collection of registers and an evaluation regime based on state transformation. An encoding of rowing machines as $F_{\leq}$ subtyping statements is given and proven correct, in the sense that a rowing machine R halts iff its translation $\mathcal{F}(R)$ is a derivable statement in $F_{\leq}$ (Section 8). We then review the definition of two-counter machines (Section 9) and show how a two-counter machine T may be encoded as a rowing machine $\mathcal{R}(T)$ such that T halts iff $\mathcal{R}(T)$ does (Section 10). Section 11 shows that the undecidability of subtyping implies the undecidability of typechecking. Section 12 briefly discusses the pragmatic import of our results.

Full proofs, omitted here to save space, may be found in an accompanying technical report [31]. In all cases, they proceed either by structural induction on derivations or by straightforward calculation from the definitions.

2 The Subtype Relation

We begin the detailed development of the undecidability of $F_{\leq}$ by establishing some notational conventions and defining the subtype relation formally.

2.1. Notation: We write $X \equiv Y$, where X and Y are types, contexts, statements, etc., to indicate that "X has the form Y." If Y contains free metavariables, then $X \equiv Y$ denotes pattern matching; for example "If $\tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$, then ..." means "If τ has the form $\forall\alpha{\leq}\tau_1.\ \tau_2$ for some α, τ_1, and τ_2, then ..."

2.2. Definition: The *types* of $F_{\leq}$ are defined by the following abstract grammar:

$$\tau \quad ::= \quad \alpha \mid \tau_1{\rightarrow}\tau_2 \mid \forall\alpha{\leq}\tau_1.\ \tau_2 \mid \mathit{Top}$$

2.3. Definition: *Typing contexts* in $F_{\leq}$ are lists of type variables and associated bounds,

$$\Gamma \quad ::= \quad \{\} \mid \Gamma, \alpha{\leq}\tau$$

with all variables distinct. (To deal formally with the $F_{\leq}$ typing relation, we would also need bindings of the form $x{:}\tau$.) The comma operator is used to denote both extension (Γ, $\alpha{\leq}\tau$) and concatenation (Γ_1, Γ_2) of contexts. The set of variables bound by a context Γ is written $\mathit{dom}(\Gamma)$. When $\Gamma \equiv \Gamma_1, \alpha{\leq}\tau, \Gamma_2$, we call τ the *bound* of α in Γ and write $\tau = \Gamma(\alpha)$.

2.4. Definition: A *subtyping statement* is a phrase of the form $\Gamma \vdash \sigma \leq \tau$. The portion of a statement to the right of the turnstile is called the *body*.

2.5. Definition: The set of free type variables in a type τ is written $FTV(\tau)$. A type τ is *closed* with respect to a context Γ if $FTV(\tau) \subseteq \mathit{dom}(\Gamma)$. A context Γ is closed if $\Gamma \equiv \{\}$, or $\Gamma \equiv \Gamma_1, \alpha{\leq}\tau$, with Γ_1 closed and τ closed with respect to Γ_1. A statement $\Gamma \vdash \sigma \leq \tau$ is closed if Γ is closed and σ and τ are closed with respect to Γ. In the following, we assume that all statements under discussion are closed. In particular, we allow only closed statements in instances of inference rules.

2.6. Convention: The metavariables σ, τ, θ, and ϕ range over types; α, β, and γ range over type variables; Γ ranges over contexts; J ranges over (closed) statements.

2.7. Definition: $F_{\leq}$ is the least three-place relation closed under the subtyping rules in Figure 2.

2.8. Convention: Types, contexts, and statements that differ only in the names of bound variables are considered to be identical. (In a statement $\Gamma_1, \alpha{\leq}\theta, \Gamma_2 \vdash \sigma \leq \tau$, the variable α is bound in Γ_2, σ, and τ.)

2.9. Definition: The capture-avoiding substitution of σ for α in τ is written $\{\sigma/\alpha\}\tau$. Substitution is extended pointwise to contexts: $\{\sigma/\alpha\}\Gamma$.

2.10. Definition: The *positive* and *negative occurrences* in a statement $\Gamma \vdash \sigma \leq \tau$ are defined as follows. The type σ and the bounds in Γ are negative occurrences; τ is a positive occurrence. If $\tau_1{\rightarrow}\tau_2$ is a positive (resp. negative) occurrence, then τ_1 is a negative (positive) occurrence and τ_2 is a positive (negative) occurrence. If $\forall\alpha{\leq}\tau_1.\ \tau_2$ is a positive (resp. negative) occurrence, then τ_1 is a negative (positive) occurrence and τ_2 is a positive (negative) occurrence.

2.11. Fact: The rules defining $F_{\leq}$ preserve the signs of occurrences: wherever a metavariable τ appears in a premise of one of the rules, it has the same sign as the corresponding occurrence of τ in the conclusion.

2.12. Definition: In the examples below, it will be convenient to rely on a few abbreviations:

$$\begin{array}{lcl} \forall\alpha.\ \tau & \stackrel{\text{def}}{=} & \forall\alpha{\leq}\mathit{Top}.\ \tau \\ \forall\alpha_1{\leq}\phi_1..\alpha_n{\leq}\phi_n.\ \tau & \stackrel{\text{def}}{=} & \forall\alpha_1{\leq}\phi_1.\ ..\ \forall\alpha_n{\leq}\phi_n.\ \tau \\ \neg\tau & \stackrel{\text{def}}{=} & \forall\alpha{\leq}\tau.\ \alpha \end{array}$$

The salient property of the last of these is that it allows the right- and left-hand sides of subtyping statements to be swapped:

2.13. Fact: $\Gamma \vdash \neg\sigma \leq \neg\tau$ is derivable iff $\Gamma \vdash \tau \leq \sigma$ is.

3 A Subtyping Algorithm

The rules defining $F_{\leq}$ do not constitute an algorithm for checking the subtype relation, since they are not syntax-directed. In particular, the rule TRANS cannot effectively be applied backwards, since this would involve "guessing" an appropriate intermediate type τ_2. Curien and Ghelli (as well as Cardelli and others) use the following reformulation:

3.1. Definition: $F^N_{\leq}$ (N for normal form) is the least relation closed under the following rules:

$$\Gamma \vdash \sigma \leq \mathit{Top} \qquad \text{(NTop)}$$

$$\Gamma \vdash \alpha \leq \alpha \qquad \text{(NRefl)}$$

$$\frac{\Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau} \qquad \text{(NVar)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1{\to}\sigma_2 \leq \tau_1{\to}\tau_2} \qquad \text{(NArrow)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(NAll)}$$

The reflexivity rule here is restricted to type variables. Transitivity is eliminated, except for instances of the following form, which are "hidden" in instances of the new rule NVAR:

$$\frac{\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau}$$

3.2. Lemma: [Curien and Ghelli] The relations $F_{\leq}$ and $F^N_{\leq}$ coincide: $\Gamma \vdash \sigma \leq \tau$ is derivable in $F_{\leq}$ iff it is derivable in $F^N_{\leq}$.

3.3. Definition: The rules defining $F^N_{\leq}$ may be read as an algorithm (i.e., a recursively defined procedure, not necessarily always terminating) for checking the subtype relation. We write $F^N_{\leq}$ to refer either to the algorithm or to the inference system, depending on context.

The algorithm $F^N_{\leq}$ may be thought of as incrementally attempting to build a normal form derivation of a statement J, starting from the root and recursively building subderivations for the premises. By Lemma 3.2, if there is any derivation whatsoever of a statement J, there is one in normal form; the algorithm is guaranteed to recapitulate this derivation and halt in finite time.

4 Nontermination of the Algorithm

Ghelli recently dispelled the widely held belief that the algorithm $F^N_{\leq}$ terminates on all inputs, by discovering the following example.

4.1. Example: Let $\theta \equiv \forall\alpha.\ \neg(\forall\beta{\leq}\alpha.\ \neg\beta)$. Then executing the algorithm $F^N_{\leq}$ on the input problem

$$\alpha_0{\leq}\theta \vdash \alpha_0 \leq (\forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1)$$

leads to the following infinite sequence of recursive calls:

$$\begin{array}{lcl} \alpha_0{\leq}\theta & \vdash & \alpha_0 \\ & \leq & \forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1 \\ \alpha_0{\leq}\theta & \vdash & \forall\alpha_1.\ \neg(\forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2) \\ & \leq & \forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1 \\ \alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash & \neg(\forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2) \\ & \leq & \neg\alpha_1 \\ \alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash & \alpha_1 \\ & \leq & \forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2 \\ \alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash & \alpha_0 \\ & \leq & \forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2 \end{array}$$

etc.

(The α-conversion steps necessary to maintain the well-formedness of the context when new variables are added are performed tacitly here, choosing new names so as to clarify the pattern of infinite regress.)

5 A Deterministic Fragment

The pattern of recursion in Ghelli's example is an instance of a more general scheme — one so general, in fact, that it can be used to encode termination problems for two-counter Turing machines. We now turn to demonstrating this fact.

5.1. Fact: The rules defining $F^N_{\leq}$ preserve the signs of occurrences: wherever a metavariable τ appears in a premise of one of the rules, it has the same sign as the corresponding occurrence of τ in the conclusion.

In what follows, it will be convenient to work with a fragment of $F^N_{\leq}$ with somewhat simpler behavior: we drop the $\to$ type constructor and its subtyping rule; we introduce a negation operator explicitly into the syntax and include a rule for comparing negated expressions; we drop the left-hand premise from the rule for

comparing quantifiers, requiring instead that when two quantified types are compared, the bound of the one on the left must be *Top*; and we consider only statements where no variable occurs positively, allowing us to drop the NREFL rule. Since the $F^N_{\leq}$ rules preserve positive and negative occurrences, we may redefine the set of types so that positive and negative types (i.e. those that appear in positive and negative positions) are separate syntactic categories. At the same time, we simplify each category appropriately.

5.2. Definition: The sets of *positive types* τ^+ and *negative types* τ^- are defined by the following abstract grammar:

$$\begin{array}{lcl} \tau^+ & ::= & \mathit{Top} \mid \neg\tau^- \mid \forall\alpha{\leq}\tau^-.\ \tau^+ \\ \tau^- & ::= & \alpha \mid \neg\tau^+ \mid \forall\alpha.\ \tau^- \end{array}$$

A *negative context* Γ^- is one whose bounds are all negative types.

5.3. Definition: $F^P_{\leq}$ (P for polarized) is the least relation closed under the following rules:

$$\Gamma^- \vdash \tau^- \leq \mathit{Top} \qquad \text{(PTOP)}$$

$$\frac{\Gamma^- \vdash \Gamma^-(\alpha) \leq \tau^+}{\Gamma^- \vdash \alpha \leq \tau^+} \qquad \text{(PVAR)}$$

$$\frac{\Gamma^-, \alpha{\leq}\phi^- \vdash \sigma^- \leq \tau^+}{\Gamma^- \vdash \forall\alpha.\ \sigma^- \leq \forall\alpha{\leq}\phi^-.\ \tau^+} \qquad \text{(PALL)}$$

$$\frac{\Gamma^- \vdash \tau^- \leq \sigma^+}{\Gamma^- \vdash \neg\sigma^+ \leq \neg\tau^-} \qquad \text{(PNEG)}$$

$F^P_{\leq}$ is almost the system we need, but it still lacks one important property: $F_{\leq}$ is not a conservative extension of $F^P_{\leq}$. For example, the non-derivable $F^P_{\leq}$ statement

$$\vdash \neg\mathit{Top} \leq \forall\alpha{\leq}\mathit{Top}.\ \alpha$$

corresponds, under the abbreviation for $\neg$, to the derivable $F_{\leq}$ statement

$$\vdash \forall\alpha{\leq}\mathit{Top}.\ \alpha \leq \forall\alpha{\leq}\mathit{Top}.\ \alpha.$$

To achieve conservativity, we restrict the form of $F^P_{\leq}$ statements even further so that negated types can never be compared with quantified types.

5.4. Definition: Let n be a fixed nonnegative number. The sets of *n-positive* and *n-negative types* are defined by the following abstract grammar:

$$\begin{array}{l} \tau^+ ::= \mathit{Top} \mid \forall\alpha_0{\leq}\tau_0^-\ ..\ \alpha_n{\leq}\tau_n^-.\ \neg\tau^- \\ \tau^- ::= \alpha \mid \forall\alpha_0\ ..\ \alpha_n.\ \neg\tau^+ \end{array}$$

We stipulate, moreover, that an n-positive type $\forall\alpha_0{\leq}\tau_0^-\ ..\ \alpha_n{\leq}\tau_n^-.\ \neg\tau^-$ is closed only if no α_i appears free in any τ_i.

An *n-negative context* is one whose bounds are all n-negative types.

5.5. Convention: To reduce clutter, we drop the superscripts + and − and leave n implicit in what follows.

5.6. Definition: $F^D_{\leq}$ (D for deterministic) is the least relation closed under the following rules:

$$\Gamma \vdash \tau \leq \mathit{Top} \qquad \text{(DTOP)}$$

$$\frac{\Gamma \vdash \Gamma(\alpha) \leq \forall\alpha_0{\leq}\tau_0\ ..\ \alpha_n{\leq}\tau_n.\ \neg\tau}{\Gamma \vdash \alpha \leq \forall\alpha_0{\leq}\tau_0\ ..\ \alpha_n{\leq}\tau_n.\ \neg\tau} \qquad \text{(DVAR)}$$

$$\frac{\Gamma, \alpha_0{\leq}\tau_0\ ..\ \alpha_n{\leq}\tau_n \vdash \tau \leq \sigma}{\Gamma \vdash \forall\alpha_0\ ..\ \alpha_n.\ \neg\sigma \leq \forall\alpha_0{\leq}\tau_0\ ..\ \alpha_n{\leq}\tau_n.\ \neg\tau} \qquad \text{(DALLNEG)}$$

Using the earlier abbreviations for negation, multiple quantification, and unbounded quantification, we may read every $F^D_{\leq}$ statement as an $F^N_{\leq}$ statement. Under this interpretation, the two subtype relations coincide for statements in their common domain:

5.7. Lemma: $F^N_{\leq}$ is a conservative extension of $F^D_{\leq}$: if J is an $F^D_{\leq}$ statement, then J is derivable in $F^D_{\leq}$ iff it is derivable in $F^N_{\leq}$.

These simplifications justify a useful change of perspective. Since the only rule in $F^N_{\leq}$ with two premises has been replaced by a rule with one premise, derivations in this fragment are linear (each node has at most one subderivation). The syntax-directed construction of such a derivation may be viewed as a deterministic state transformation process, where the subtyping statement being verified is the current state and the single premise that must be recursively verified (if any) is the next state. In other words, a subtyping statement is thought of as an instantaneous description of a kind of automaton.

From now on we use terminology that makes the intuition of "subtyping as state transformation" more explicit. Analogous terminology and notation will be used to describe the execution behavior of the other calculi introduced below.

5.8. Definition: The *one-step elaboration* function $\mathcal{E}$ for $F^D_{\leq}$-statements is the partial mapping defined by:

$$\mathcal{E}(J) = \begin{cases} J' & \text{if } J \text{ is the conclusion of an instance of DVAR or DALLNEG and } J' \text{ is the corresponding premise} \\ \text{undef.} & \text{if } J \text{ is an instance of DTOP.} \end{cases}$$

J' is an *immediate subproblem* of J in $F^D_{\leq}$, written $J \longrightarrow_D J'$, if $J' = \mathcal{E}(J)$. J' is a *subproblem* of J in $F^D_{\leq}$, written $J \xrightarrow{*}_D J'$, if either $J \equiv J'$ or $J \longrightarrow_D J_1$ and $J_1 \xrightarrow{*}_D J'$. The *elaboration* of a statement J is the sequence of subproblems encountered by the subtyping algorithm given J as input.

6 Eager Substitution

To make a smooth transition between the subtyping statements of $F_{\leq}$ and the rowing machine abstraction to be introduced in Section 7, we need one more variation in the definition of subtyping, where, instead of maintaining a context with the bounds of free variables, the quantifier rule immediately substitutes the bounds into the body of the statement.

6.1. Definition: The simultaneous, capture-avoiding substitution of ϕ_0 through ϕ_n, respectively, for α_0 through α_n in τ, is written $\{\phi_0/\alpha_0 \,..\, \phi_n/\alpha_n\}\,\tau$.

6.2. Definition: $F^F_{\leq}$ (F for flattened) is the least relation closed under the following rules:

$$\vdash \tau \leq \mathit{Top} \qquad \text{(FTop)}$$

$$\frac{\vdash \{\phi_0/\alpha_0 \,..\, \phi_n/\alpha_n\}\,\tau \leq \{\phi_0/\alpha_0 \,..\, \phi_n/\alpha_n\}\,\sigma}{\vdash \forall\alpha_0 \,..\, \alpha_n.\ \neg\sigma \leq \forall\alpha_0{\leq}\phi_0 \,..\, \alpha_n{\leq}\phi_n.\ \neg\tau} \qquad \text{(FAllNeg)}$$

6.3. Remark: Of course, an analogous reformulation of full $F_{\leq}$ would not be correct. For example, in the non-derivable statement

$$\vdash (\forall\alpha{\leq}\mathit{Top}.\ \mathit{Top}) \leq (\forall\alpha{\leq}\mathit{Top}.\ \alpha)$$

substituting *Top* for α in the bodies of the quantifiers yields the derivable statement $\vdash \mathit{Top} \leq \mathit{Top}$. But having restricted our attention to statements where variables appear only negatively, we are guaranteed that the only position where the elaboration of a statement can cause a variable to appear by itself in the body of a subproblem is on the left-hand side, where it will immediately be replaced by its bound. We are therefore safe in making the substitution eagerly.

6.4. Lemma: $F^D_{\leq}$ is a conservative extension of $F^F_{\leq}$.

7 Rowing Machines

The reduction from two-counter Turing machines to $F_{\leq}$ subtyping statements is easiest to understand in terms of an intermediate abstraction called a rowing machine that makes more stylized use of bound variables. A rowing machine is a tuple of *registers*

$$\langle \rho_1 \,..\, \rho_n \rangle,$$

where the contents of each register is a *row*. By convention, the first register is the machine's *program counter* (or *PC*). To move to the next state, the *PC* is used as a template to construct the new contents of each of the registers from the current contents of all of the registers (including the *PC*).

7.1. Definition: The set of *rows* (of width n) is defined by the following abstract grammar:

$$\begin{array}{lcll} \rho & ::= & \alpha_m & \text{for } 1 \leq m \leq n \\ & | & [\alpha_1..\alpha_n]\langle\rho_1..\rho_n\rangle & \\ & | & \text{HALT} & \end{array}$$

The variables $\alpha_1..\alpha_n$ in $[\alpha_1..\alpha_n]\langle\rho_1..\rho_n\rangle$ are binding occurrences whose scope is the rows ρ_1 through ρ_n. We regard rows that differ only in the names of bound variables as identical.

7.2. Definition: A *rowing machine* (of width n) is a tuple $\langle\rho_1..\rho_n\rangle$, where each ρ_i is a row of width n with no free variables.

7.3. Definition: The *one-step elaboration* function $\mathcal{E}$ for rowing machines of width n is the partial mapping

$$\mathcal{E}(\langle\rho_1..\rho_n\rangle) = \begin{cases} \langle\{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\,\rho_{11} \\ \quad ..\ \{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\,\rho_{1n}\rangle \\ \qquad\qquad \text{if } \rho_1 = [\alpha_1..\alpha_n]\langle\rho_{11}..\rho_{1n}\rangle \\ \text{undefined} \\ \qquad\qquad \text{if } \rho_1 = \text{HALT}. \end{cases}$$

(Since rowing machines consist only of closed rows, we need not define the evaluation function for the case where the *PC* is a variable. Also, since all the ρ_n are closed, the substitution is trivially capture-avoiding.)

7.4. Notational conventions: When the symbol "—" appears as the ith component of a compound row $[\alpha_1..\alpha_n]\langle\rho_1..\rho_n\rangle$, it stands for the variable α_i.

To avoid a proliferation of variable names in the examples and definitions below, we sometimes use numerical indices (like deBruijn indices [22]) rather than names for variables: the "variable" #n refers to the n^{th} bound variable of the row in which it appears; ##n refers to the n^{th} bound variable of the row enclosing the one in which it appears; and so on. For example, the row $[\alpha_1..\alpha_3]\langle\alpha_1, [\beta_1..\beta_3]\langle\alpha_1, \beta_1, \beta_3\rangle, \alpha_1\rangle$ would be abbreviated $\langle$—, $\langle$##1, #1, —$\rangle$, #1$\rangle$.

7.5. Definition: A rowing machine R *halts* if there is a machine R' such that $R \longrightarrow_R^{\star} R'$ and the *PC* of R' is the instruction HALT.

7.6. Example: The machine

$$\langle \text{LOOP}, \text{A}, \text{B} \rangle,$$

where

$$\begin{array}{lcl} \text{LOOP} & \equiv & \langle —, \#3, \#2\rangle \\ \text{A} & \equiv & \text{an arbitrary row} \\ \text{B} & \equiv & \text{an arbitrary row} \end{array}$$

executes an infinite loop where the contents of the second and third register are exchanged at successive steps:

$$\begin{array}{ll} & \langle \text{LOOP}, \text{A}, \text{B}\rangle \\ \longrightarrow_R & \langle \text{LOOP}, \text{B}, \text{A}\rangle \\ \longrightarrow_R & \langle \text{LOOP}, \text{A}, \text{B}\rangle \\ \longrightarrow_R & \ldots \end{array}$$

7.7. Example: The row

$$\text{BRI} \equiv \langle \#2, — \rangle$$

encodes an *indirect branch* to the contents of register 2 at the moment when BRI is executed. The machine

$$\langle \text{BRI}, \langle \text{BRI}, \langle \text{BRI}, \text{HALT}\rangle\rangle\rangle$$

elaborates as follows:

$$\begin{aligned}
& \langle \text{BRI}, \langle \text{BRI}, \langle \text{BRI}, \text{HALT}\rangle\rangle\rangle \\
\longrightarrow_R \quad & \langle\langle \text{BRI}, \langle \text{BRI}, \text{HALT}\rangle\rangle, \langle \text{BRI}, \langle \text{BRI}, \text{HALT}\rangle\rangle\rangle \\
\longrightarrow_R \quad & \langle \text{BRI}, \langle \text{BRI}, \text{HALT}\rangle\rangle \\
\longrightarrow_R \quad & \langle\langle \text{BRI}, \text{HALT}\rangle, \langle \text{BRI}, \text{HALT}\rangle\rangle \\
\longrightarrow_R \quad & \langle \text{BRI}, \text{HALT}\rangle \\
\longrightarrow_R \quad & \langle \text{HALT}, \text{HALT}\rangle.
\end{aligned}$$

8 Encoding Rowing Machines as Subtyping Problems

We now show how a rowing machine R can be encoded as a subtyping problem $\mathcal{F}(R)$ such that R halts iff $\mathcal{F}(R)$ is derivable in $F^F_{\leq}$.

The idea of the translation is that a rowing machine $R = \langle \rho_1 .. \rho_n \rangle$ becomes a subtyping statement such that

- if $\rho_1 = \text{HALT}$, the elaboration of $\mathcal{F}(R)$ halts (by reaching a subproblem where *Top* appears on the right-hand side);
- if $\rho_1 = [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n}\rangle$, the elaboration of $\mathcal{F}(R)$ reaches a subproblem that encodes the rowing machine

$$\langle \{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\, \rho_{11} .. \{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\, \rho_{1n}\rangle.$$

In more detail, if $R = \langle [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n}\rangle .. \rho_n\rangle$, then $\mathcal{F}(R)$ is essentially the following:

$$\begin{aligned}
\vdash \quad & \forall \gamma_1..\gamma_n. \\
& \quad \neg(\forall \gamma'_1 \leq \gamma_1 .. \gamma'_n \leq \gamma_n. \\
& \quad\quad \neg \ldots) \\
\leq \quad & \forall \gamma_1 \leq \mathcal{F}(\rho_1) .. \gamma_n \leq \mathcal{F}(\rho_n). \\
& \quad \neg(\forall \alpha_1..\alpha_n. \\
& \quad\quad \neg(\forall \alpha'_1 \leq \mathcal{F}(\rho_{11}) .. \alpha'_n \leq \mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11}))).
\end{aligned}$$

The elaboration of this statement proceeds as follows:

1. The current contents of the registers $\rho_1..\rho_n$ are temporarily saved by matching the quantifiers on the right with the ones on the left; this has the effect of substituting the bounds $\mathcal{F}(\rho_1) .. \mathcal{F}(\rho_n)$ for free occurrences of the variables $\gamma_1..\gamma_n$ on the left-hand side.

   The right- and left-hand sides are also swapped (by the $\neg$ constructor on both sides), so that what now appears on the left is a sequence of variable bindings for the free variables $\alpha_1..\alpha_n$ of ρ_1.
2. The saved contents of the original registers now appear on the right-hand side. When these are matched with the quantifiers on the left, the result is that the old values of the registers are substituted for the variables $\alpha_1 .. \alpha_n$ in the body of the left-hand side.

   Swapping right- and left-hand sides again yields a statement of the same form as the original, where the appropriate instances of $\mathcal{F}(\rho_{11}) .. \mathcal{F}(\rho_{1n})$ appear as the bounds of the outer quantifiers on the right:

$$\begin{aligned}
\vdash \ldots \leq\ & (\forall \gamma_1 \leq \{\mathcal{F}(\rho_1)/\alpha_1 .. \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}) .. \\
& \quad \gamma_n \leq \{\mathcal{F}(\rho_1)/\alpha_1 .. \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{1n}). \\
& \quad\quad \ldots \{\mathcal{F}(\rho_1)/\alpha_1 .. \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}) \ldots
\end{aligned}$$

To be able to get back to a statement of the same form as the original, one piece of additional mechanism is required: besides the n variables used to store the old state of the registers, a variable γ_0 is used to hold the original value of the entire left-hand side of $\mathcal{F}(R)$. This variable is used at the end of a cycle to set up the left hand side of the statement encoding the next state of the rowing machine.

8.1. Definition: Let ρ be a row of width n. The $F^F_{\leq}$-*translation* of ρ, written $\mathcal{F}(\rho)$, is the n-negative type

$$\begin{aligned}
& \mathcal{F}(\alpha_i) && = \alpha_i \\
& \mathcal{F}(\text{HALT}) && = \forall \gamma_0, \alpha_1 .. \alpha_n.\ \neg \mathit{Top} \\
& \mathcal{F}([\alpha_1..\alpha_n]\langle \rho_1..\rho_n\rangle) && = \\
& \quad \forall \gamma_0, \alpha_1 .. \alpha_n. \\
& \quad\quad \neg\Big(\forall \gamma'_0 \leq \gamma_0, \alpha'_1 \leq \mathcal{F}(\rho_1) .. \alpha'_n \leq \mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1)\Big)
\end{aligned}$$

where γ_0, γ'_0, and α'_1 through α'_n are fresh variables.

8.2. Fact: The free variables of $\mathcal{F}(R)$ coincide with those of R.

8.3. Definition: Let $R = \langle \rho_1..\rho_n\rangle$ be a rowing machine. The $F^F_{\leq}$-*translation* of R, written $\mathcal{F}(R)$, is the $F^F_{\leq}$ statement

$$\vdash \sigma \leq \forall \gamma_0 \leq \sigma, \gamma_1 \leq \mathcal{F}(\rho_1) .. \gamma_n \leq \mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1),$$

where

$$\sigma \equiv \forall \gamma_0, \gamma_1..\gamma_n.\ \neg(\forall \gamma'_0 \leq \gamma_0, \gamma'_1 \leq \gamma_1 .. \gamma'_n \leq \gamma_n.\ \neg\gamma_0).$$

8.4. Lemma: If $R \longrightarrow_R R'$, then $\mathcal{F}(R) \xrightarrow{*}_F \mathcal{F}(R')$.

Proof: By the definition of the elaboration function for rowing machines, $R \equiv \langle \rho_1..\rho_n\rangle$, where

$$\rho_1 \equiv [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n}\rangle,$$

and

$$R' \equiv \langle \{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\, \rho_{11} .. \{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\, \rho_{1n}\rangle.$$

Calculate as follows:

$$\begin{aligned}
& \mathcal{F}(R) \\
\equiv \quad & \vdash \ \sigma \\
& \leq \ \forall \gamma_0 \leq \sigma, \gamma_1 \leq \mathcal{F}(\rho_1) .. \gamma_n \leq \mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1)
\end{aligned}$$

$$
\begin{array}{lll}
\equiv & \vdash & \forall\gamma_0, \gamma_1..\gamma_n.\ \neg(\forall\gamma_0'\leq\gamma_0, \gamma_1'\leq\gamma_1\ ..\ \gamma_n'\leq\gamma_n.\ \neg\gamma_0) \\
& \leq & \forall\gamma_0\leq\sigma, \gamma_1\leq\mathcal{F}(\rho_1)\ ..\ \gamma_n\leq\mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1) \\
\longrightarrow_F & \vdash & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\gamma_1\ ..\ \mathcal{F}(\rho_n)/\gamma_n\}\,\mathcal{F}(\rho_1) \\
& \leq & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\gamma_1\ ..\ \mathcal{F}(\rho_n)/\gamma_n\} \\
& & \quad (\forall\gamma_0'\leq\gamma_0, \gamma_1'\leq\gamma_1\ ..\ \gamma_n'\leq\gamma_n.\ \neg\gamma_0) \\
\equiv & \vdash & \mathcal{F}(\rho_1) \\
& \leq & \forall\gamma_0'\leq\sigma, \gamma_1'\leq\mathcal{F}(\rho_1)\ ..\ \gamma_n'\leq\mathcal{F}(\rho_n).\ \neg\sigma \\
\equiv & \vdash & \forall\gamma_0, \alpha_1..\alpha_n. \\
& & \quad \neg(\forall\gamma_0'\leq\gamma_0, \alpha_1'\leq\mathcal{F}(\rho_{11})\ ..\ \alpha_n'\leq\mathcal{F}(\rho_{1n}). \\
& & \quad\quad \neg\mathcal{F}(\rho_{11})) \\
& \leq & \forall\gamma_0'\leq\sigma, \gamma_1'\leq\mathcal{F}(\rho_1)\ ..\ \gamma_n'\leq\mathcal{F}(\rho_n).\ \neg\sigma \\
\equiv & \vdash & \forall\gamma_0, \alpha_1..\alpha_n. \\
& & \quad \neg(\forall\gamma_0'\leq\gamma_0, \alpha_1'\leq\mathcal{F}(\rho_{11})\ ..\ \alpha_n'\leq\mathcal{F}(\rho_{1n}). \\
& & \quad\quad \neg\mathcal{F}(\rho_{11})) \\
& \leq & \forall\gamma_0\leq\sigma, \alpha_1\leq\mathcal{F}(\rho_1)\ ..\ \alpha_n\leq\mathcal{F}(\rho_n).\ \neg\sigma \\
\longrightarrow_F & \vdash & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\sigma \\
& \leq & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\} \\
& & \quad (\forall\gamma_0'\leq\gamma_0, \alpha_1'\leq\mathcal{F}(\rho_{11})\ ..\ \alpha_n'\leq\mathcal{F}(\rho_{1n}). \\
& & \quad\quad \neg\mathcal{F}(\rho_{11})) \\
\equiv & \vdash & \sigma \\
& \leq & \forall\,\gamma_0'\leq\sigma, \\
& & \quad \alpha_1'\leq(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11}))\ .. \\
& & \quad \alpha_n'\leq(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{1n})). \\
& & \quad\quad \neg(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \\
\equiv & \vdash & \sigma \\
& \leq & \forall\,\gamma_0\leq\sigma, \\
& & \quad \gamma_1\leq(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11}))\ .. \\
& & \quad \gamma_n\leq(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{1n})). \\
& & \quad\quad \neg(\{\mathcal{F}(\rho_1)/\alpha_1\ ..\ \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \\
\equiv & & \mathcal{F}(R'). \qquad \square
\end{array}
$$

8.5. Lemma: If $R \equiv \langle \text{HALT}, \rho_2..\rho_n\rangle$, then $\mathcal{F}(R)$ is derivable in $F_{\leq}^{F}$.

Proof: Similar. $\square$

8.6. Corollary: The rowing machine R halts iff $\mathcal{F}(R)$ is derivable in $F_{\leq}^{F}$.

9 Two-counter Machines

This section reviews the definition of two-counter Turing machines; see, e.g., Hopcroft and Ullman [27] for more details.

9.1. Definition: A *two-counter machine* is a tuple $\langle PC, A, B, I_1..I_w\rangle$, where A and B are nonnegative numbers and PC and I_1 through I_w are instructions of one of the forms:

INCA⇒m
INCB⇒m
TSTA⇒m/n
TSTB⇒m/n
HALT.

9.2. Definition: The *elaboration function* $\mathcal{E}$ for two-counter machines is the partial function mapping $T = \langle PC, A, B, I_1..I_w\rangle$ to

$$
\mathcal{E}(T) = \begin{cases}
\langle I_m, A{+}1, B, I_1..I_w\rangle & \text{if } PC \equiv \text{INCA}{\Rightarrow}m \\
\langle I_m, A, B{+}1, I_1..I_w\rangle & \text{if } PC \equiv \text{INCB}{\Rightarrow}m \\
\langle I_m, A, B, I_1..I_w\rangle & \text{if } PC \equiv \text{TSTA}{\Rightarrow}m/n \text{ and } A = 0 \\
\langle I_n, A{-}1, B, I_1..I_w\rangle & \text{if } PC \equiv \text{TSTA}{\Rightarrow}m/n \text{ and } A > 0 \\
\langle I_m, A, B, I_1..I_w\rangle & \text{if } PC \equiv \text{TSTB}{\Rightarrow}m/n \text{ and } B = 0 \\
\langle I_n, A, B{-}1, I_1..I_w\rangle & \text{if } PC \equiv \text{TSTB}{\Rightarrow}m/n \text{ and } B > 0 \\
\text{undefined} & \text{if } PC \equiv \text{HALT}.
\end{cases}
$$

9.3. Definition: A two-counter machine T *halts* if $T \xrightarrow{*}_T T'$ for some machine $T' \equiv \langle \text{HALT}, A', B', I_1..I_w\rangle$.

9.4. Fact: The halting problem for two-counter machines is undecidable.

Proof sketch: Hopcroft and Ullman [27, pp. 171–173] show that a similar formulation of two-counter machines is Turing-equivalent. (Their two-counter machines have test instructions that do not change the contents of the register being tested and separate decrement instructions. It is easy to check that this formulation and the one used here are inter-encodable.) $\square$

10 Encoding Two-counter Machines as Rowing Machines

We can now finish the proof of the undecidability of $F_{\leq}$ subtyping by showing that any two-counter machine T can be encoded as a rowing machine $\mathcal{R}(T)$ such that T halts iff $\mathcal{R}(T)$ does.

The main trick of the encoding lies in the representation of natural numbers as rows. Each number n is encoded as a *program* (i.e., a row) that, when executed, branches indirectly through one of two registers whose contents have been set beforehand to appropriate destinations for the zero and nonzero cases of a test; in other words, n encapsulates the behavior of the test instruction on a register containing n. The increment operation simply builds a new program of this sort from an existing one. The new program saves a pointer to the present contents of the register in a local variable so that it can restore the old value (i.e., one less than its own value) before executing the branch.

The encoding $\mathcal{R}(T)$ of a two-counter machine $T \equiv \langle PC, A, B, I_1..I_w\rangle$ comprises the following registers:

#1	$\mathcal{R}^w(PC)$
#2	$\mathcal{R}^w_A(A)$
#3	$\mathcal{R}^w_B(B)$
#4	address register for zero branches
#5	address register for nonzero branches
#6	$\mathcal{R}^w(I_1)$
...	
#6+w−1	$\mathcal{R}^w(I_w)$.

We use four translation functions for the various components: $\mathcal{R}(T)$ is the encoding of a the two-counter machine T as a rowing machine of width $w+5$; $\mathcal{R}^w(I)$ is the encoding of a two-counter instruction I as a row of width $w+5$; $\mathcal{R}^w_A(n)$ is the encoding of the natural number n, when it appears as the contents of register A, as a row of width $w+5$; $\mathcal{R}^w_B(n)$ is the encoding of the natural number n, when it appears as the contents of register B, as a row of width $w+5$.

10.1. Definition: The *row-encoding* (for w instructions) of a natural number n in register A, written $\mathcal{R}^w_A(n)$, is defined as follows:

$$\mathcal{R}^w_A(0) = \langle \#4, —, —, \text{HALT}, \text{HALT}, \underbrace{— .. —}_{w \text{ times}} \rangle$$

$$\mathcal{R}^w_A(n{+}1) = \langle \#5, \mathcal{R}^w_A(n), —, \text{HALT}, \text{HALT}, \underbrace{— .. —}_{w \text{ times}} \rangle.$$

The row-encoding (for w instructions) of a natural number n in register B, written $\mathcal{R}^w_B(n)$, is defined as follows:

$$\mathcal{R}^w_B(0) = \langle \#4, —, —, \text{HALT}, \text{HALT}, \underbrace{— .. —}_{w \text{ times}} \rangle$$

$$\mathcal{R}^w_B(n{+}1) = \langle \#5, —, \mathcal{R}^w_B(n), \text{HALT}, \text{HALT}, \underbrace{— .. —}_{w \text{ times}} \rangle.$$

10.2. Definition: The *row-encoding* (for w instructions) of an instruction I, written $\mathcal{R}^w(I)$, is defined as follows:

$\mathcal{R}^w(\text{INCA}{\Rightarrow}m)$
$= \langle \#m{+}5, \langle \#5, \#\#2, —, \text{HALT}, \text{HALT}, — .. — \rangle, —,$
$\text{HALT}, \text{HALT}, — .. — \rangle$
$\mathcal{R}^w(\text{INCB}{\Rightarrow}m)$
$= \langle \#m{+}5, —, \langle \#5, —, \#\#3, \text{HALT}, \text{HALT}, — .. — \rangle,$
$\text{HALT}, \text{HALT}, — .. — \rangle$
$\mathcal{R}^w(\text{TSTA}{\Rightarrow}m/n)$
$= \langle \#2, —, —, \#m{+}5, \#n{+}5, — .. — \rangle$
$\mathcal{R}^w(\text{TSTB}{\Rightarrow}m/n)$
$= \langle \#3, —, —, \#m{+}5, \#n{+}5, — .. — \rangle$
$\mathcal{R}^w(\text{HALT})$
$= \langle \text{HALT}, —, —, \text{HALT}, \text{HALT}, — .. — \rangle.$

10.3. Definition: Let $T \equiv \langle PC, A, B, I_1..I_w \rangle$ be a two-counter machine. The *row-encoding* of T, written $\mathcal{R}(T)$, is the rowing machine of width $w{+}5$ defined as follows:

$$\mathcal{R}(T) = \langle \mathcal{R}^w(PC), \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \text{HALT}, \text{HALT}, \mathcal{R}^w(I_1) .. \mathcal{R}^w(I_w) \rangle$$

10.4. Lemma: If $T \longrightarrow_T T'$, then $\mathcal{R}(T) \xrightarrow{*}_R \mathcal{R}(T')$.

Proof: Straightforward. □

10.5. Lemma: If $T = \langle \text{HALT}, A, B, I_1..I_w \rangle$, then $\mathcal{R}(T)$ halts.

Proof: Immediate. □

10.6. Corollary: T halts iff $\mathcal{R}(T)$ does.

10.7. Theorem: The $F_{\leq}$ subtyping relation is undecidable.

Proof: Assume, for a contradiction, that we had a total-recursive procedure for testing the derivability of subtyping statements in $F_{\leq}$. Then to decide whether a two-counter machine T halts, we could use this procedure to test whether $\mathcal{F}(\mathcal{R}(T))$ is derivable, since T halts iff $\mathcal{R}(T)$ halts (by Corollary 10.6), iff $\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^F_{\leq}$ (by Corollary 8.6), iff $\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^D_{\leq}$ (by Lemma 6.4) iff $\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^N_{\leq}$ (by Lemma 5.7), iff $\mathcal{F}(\mathcal{R}(T))$ is derivable in $F_{\leq}$ (by Lemma 3.2). □

11 Typechecking

From the undecidability of $F_{\leq}$ subtyping, the undecidability of typechecking follows immediately: we need only show how to write down a term that is well typed iff a given subtyping statement $\vdash \sigma \leq \tau$ is derivable. One such term is $\lambda f{:}\tau{\rightarrow}\mathit{Top}.\ \lambda a{:}\sigma.\ f\ a$.

12 Conclusions

The undecidability of $F_{\leq}$ will perhaps surprise many of those who have studied, extended, and applied it since its introduction in 1985. But it may turn out that language designs and implementations based on $F_{\leq}$ will not be greatly affected by this discovery, since the algorithm has been used for several years now without any sign of misbehavior in any situation arising in practice. Indeed, constructing even the simplest nonterminating example requires a contortion that is difficult to imagine anyone performing by accident. Moreover, a number of useful fragments of $F_{\leq}$ are easily shown to be decidable. For example:

- The prenex fragment, where all quantifiers appear at the outside and quantifiers are instantiated only at monotypes.
- A predicative fragment where types are stratified into universes and the bound of a quantified type lives in a lower universe than the quantified type itself.
- Cardelli and Wegner's original formulation where the bounds of two quantified types must be identical in order for one to be a subtype of the other.

Acknowledgements

I am grateful for productive discussions with John Reynolds, Robert Harper, Luca Cardelli, Giorgio Ghelli, Daniel Sleator, and Tim Freeman.

References

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

[2] 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. To appear.

[3] Kim Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 38–50, 1988.

[4] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Object Oriented Programing: Systems, Languages, and Applications (Conference Proceedings)*, pages 457–467, 1989.

[5] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

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

[7] 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.

[8] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[9] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[10] Luca Cardelli. Typeful programming. Research Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.

[11] Luca Cardelli, 1991. Personal Communication.

[12] Luca Cardelli. Extensible records in a pure calculus of subtyping. To appear, 1991.

[13] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest: (Extended abstract). In *ACM Conference on Lisp and Functional Programming*, pages 30–43, Nice, France, June 1990. Extended version available as DEC SRC Research Report 55, Feb. 1990.

[14] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Ito and Meyer [28], pages 750–770.

[15] Luca Cardelli and John Mitchell. Operations on records (summary). In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, Tulane University, New Orleans, March 1989. Springer Verlag. To appear in Mathematical Structures in Computer Science; also available as DEC Systems Research Center Research Report #48, August, 1989.

[16] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[17] Felice Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

[18] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990.

[19] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560, New York, 1980. Academic Press.

[20] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 1991. To appear.

[21] Pierre-Louis Curien and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In Ito and Meyer [28], pages 731–749.

[22] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[23] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[24] Giorgio Ghelli, 1991. Personal Communication.

[25] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[26] Carl Gunter, 1990. Personal Communication.

[27] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[28] T. Ito and A. R. Meyer, editors. *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science. Springer-Verlag, September 1991.

[29] Simone Martini. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

[30] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[31] Benjamin C. Pierce. Bounded quantification is undecidable. Technical Report CMU-CS-91-161, Carnegie Mellon University, July 1991.

[32] Benjamin C. Pierce. Programming with intersection types and bounded polymorphism. Ph.D. thesis (in progress), 1991.

[33] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[34] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.