Technical Report

UCAM-CL-TR-171
ISSN 1476-2986

Number 171

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Some types with inclusion properties in $\forall, \rightarrow, \mu$

Jon Fairbairn

June 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Some Types with Inclusion Properties in $\forall, \rightarrow, \mu$

June 1989

Jon Fairbairn
University of Cambridge Computer Laboratory,
Corn Exchange Street,
Cambridge CB2 3QG,
United Kingdom.
Telephone +44 223 334688, Telex 81240 (CAMSPL-G)
Mail address: jf@UK.AC.Cam.CL

Abstract

This paper concerns the $\forall, \rightarrow, \mu$ type system used in the non-strict functional programming language Ponder. While the type system is akin to the types of Second Order Lambda-calculus, the absence of type application makes it possible to construct types with useful inclusion relationships between them.

To illustrate this, the paper contains definitions of a natural numbers type with many definable subtypes, and of a record type with inheritance.

1. Introduction

This paper is an exploration of the type system used in the functional programming language Ponder [Fairbairn 1982]. Ponder is a very small language in the sense that it has few built in constructions. As befits such a language the type system is also small, having only three operators.

The type system resembles that of MacQueen, Sethi and Plotkin [MacQueen 1982, 1984], but has no ground types, and no type conjunction operator. Moreover there are no union types, no record types and, properly speaking, no number types (although implementations use numbers and characters from the concrete machine for efficiency).

Ponder thus bears a close resemblance to the Second Order Lambda-calculus [Reynolds 1974]. The chief difference from the SOL type system is that types are statements about expressions and have no effect on their meaning. Polymorphism is thus a property of an expression, so type abstraction is replaced by quantification, and there is no explicit type application in the language: polymorphic objects are tacitly instantiated. Thus a Ponder programme is considered to be an untyped lambda-term, the type information being added as a method of avoiding certain classes of mistake. For example the identity function on integers is considered to be the same object as the identity function on characters. This corresponds to the way that the language is given a dynamic semantics — the evaluation of a Ponder programme corresponds to the reduction of the untyped lambda term obtained by erasing all type information from the programme.

The remainder of this section describes the notation used in the paper. For present purposes it is not necessary to introduce the declaration structure of the language, and rather than use Ponder syntax for expressions, I shall use a more familiar λ-notation.

1.1. Expressions

Expressions have the following abstract syntax:

$$\text{Expression} = \begin{cases} \mathit{var} & \text{Variables} \\ \lambda \mathit{var}.\text{Expression} & \text{Function abstractions} \\ \text{Expression Expression} & \text{Applications} \\ \text{Expression}\restriction\text{Type} & \text{Cast expressions} \end{cases}$$

This is essentially the normal syntax for untyped λ-terms except for the introduction of cast expressions (and the use of arbitrary words in italics as variable names). An expression e is not considered to have a meaning unless $e : \mathsf{t}$ ("e has type t") for some type t is derivable from the rules given in the appendix. Note that $e\restriction\mathsf{t}$ is meaningless unless $e : \mathsf{t}$ is derivable.

1.2. Types

The abstract syntax of types is as follows:

$$\text{Type} = \begin{cases} \mathsf{v} & \text{Type variables} \\ \text{Type} \rightarrow \text{Type} & \text{Functions} \\ \forall \mathsf{v}.\text{Type} & \text{Quantified types} \\ \mu \mathsf{v}.\text{Type} & \text{Recursive types} \\ \mathsf{G}[\text{Type}, \ldots, \text{Type}] & \text{Generators} \end{cases}$$

Generators are just user defined parameterised type constructors, and as such add nothing to the type structure.

1.2.1. Type Variables

Type names and variables will be words written in sans-serif, eg T, t, $\mathsf{Long\text{-}name}$, ….

1.2.2. Function Types

The simplest type constructor is that of the function from one type to another, which is written using '$\rightarrow$'. Thus if $\mathsf{Parameter}$ and Result are both types, then $\mathsf{Parameter} \rightarrow \mathsf{Result}$ is the type of functions that, when applied to objects of type $\mathsf{Parameter}$ yield objects of type Result. Note that $\rightarrow$ associates to the right, so that $\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{c}$ means $\mathsf{a} \rightarrow (\mathsf{b} \rightarrow \mathsf{c})$.

1.2.3. Quantifiers

The universal quantifier, $\forall$, introduces polymorphic types. If we can derive $x : \forall \mathsf{v}.\mathsf{T}[\mathsf{v}]$, then we can derive $x : \mathsf{T}[\mathsf{t}]$ for any type t (this fact is expressed by rules $R3$ and $V5$ of the appendix).

For example, $\lambda x.x : \forall \mathsf{v}.\mathsf{v} \rightarrow \mathsf{v}$, and hence if $3 : \mathsf{Int}$ we have$\lambda x.x : \mathsf{Int} \rightarrow \mathsf{Int}$ gives $(\lambda x.x)\,3 : \mathsf{Int}$.

A note about binding: the scope of a variable introduced by a quantifier extends as far to the right as possible, but is limited by parentheses, so $\forall \mathsf{t}.(\mathsf{t} \rightarrow \mathsf{t}) \rightarrow \mathsf{Bool}$ means the same as $\forall \mathsf{t}.((\mathsf{t} \rightarrow \mathsf{t}) \rightarrow \mathsf{Bool})$, and takes as argument any function with the same parameter and result types, whereas $(\forall \mathsf{t}.\mathsf{t} \rightarrow \mathsf{t}) \rightarrow \mathsf{Bool}$ demands that its argument has type $\forall \mathsf{t}.\mathsf{t} \rightarrow \mathsf{t}$. For the sake of convenience, $\forall \mathsf{t}.\forall \mathsf{u} \ldots$ may be written $\forall \mathsf{t}, \mathsf{u} \ldots$

1.2.4. Recursive Types

Recursive types are introduced by means of the μ operator. Such a type satisfies the equation $\mu \mathsf{v}.\mathsf{t} = \mathsf{t}[\mu \mathsf{v}.\mathsf{t}/\mathsf{v}]$, so that $\mu \mathsf{l}.\mathsf{t} \times \mathsf{l} = \mathsf{t} \times (\mu \mathsf{l}.\mathsf{t} \times \mathsf{l}) = \mathsf{t} \times \mathsf{t} \times (\mu \mathsf{l}.\mathsf{t} \times \mathsf{l}) = \ldots$

2. Simple relationships between types

As part of the rules for deriving typings of terms, the appendix includes a definition of the relationship $\geq$ between types[†]. $\mathsf{a} \geq \mathsf{b}$ means that every object of type a is also an object of type b.

As a first illustration of the inclusion properties, we can consider the type

$$\mathsf{Bool} \triangleq \forall \mathsf{t}.\mathsf{t} \rightarrow \mathsf{t} \rightarrow \mathsf{t},$$

in which *true* $\triangleq \lambda t.\lambda f.t$ and *false* $\triangleq \lambda t.\lambda f.f$. Observe that *true* also has type $\mathsf{TrueType} \triangleq \forall \mathsf{t}, \mathsf{f}.\mathsf{t} \rightarrow \mathsf{f} \rightarrow \mathsf{t}$ and *false* has type $\mathsf{FalseType} \triangleq \forall \mathsf{t}, \mathsf{f}.\mathsf{t} \rightarrow \mathsf{f} \rightarrow \mathsf{f}$. Furthermore, one cannot derive *true* : $\mathsf{FalseType}$ or *false* : $\mathsf{TrueType}$, but we do have that $\mathsf{TrueType} \geq \mathsf{Bool}$ and $\mathsf{FalseType} \geq \mathsf{Bool}$.

† The (perhaps counter intuitive) use of $\geq$ rather than $\leq$ here is Milner's [Milner 1978]

Now an expression of the form $((\lambda b.e)\upharpoonright \mathsf{TrueType} \rightarrow \mathsf{t})$ can only be applied to objects of type $\mathsf{TrueType}$, for example *true*. This corresponds to a restriction of the applicability of the expression to a subtype of Bool.

2.1. Pairs

The natural number type that I wish to define relies on pairs, so it is useful to include a definition of pair types here.

$$\mathsf{Pair}\ [\mathsf{l},\ \mathsf{r}] \triangleq \forall \mathsf{res}.(\mathsf{l} \rightarrow \mathsf{r} \rightarrow \mathsf{res}) \rightarrow \mathsf{res}$$

In which pairs are represented as functions that may be applied to *true* or *false* to return their first or second component respectively. The pair constructing function *pair* is thus

$$\lambda l.\lambda r.\lambda u.u\ l\ r : \forall \mathsf{l}, \mathsf{r}.\ \mathsf{l} \rightarrow \mathsf{r} \rightarrow \forall \mathsf{res}.(\mathsf{l} \rightarrow \mathsf{r} \rightarrow \mathsf{res}) \rightarrow \mathsf{res}$$

and the functions to take the left and right elements of a pair are $\mathit{left} \triangleq \lambda p.p\ \mathit{true}$ and $\mathit{right} \triangleq \lambda p.p\ \mathit{false}$.

It is nicer to write $\mathsf{Pair}\ [\mathsf{a}, \mathsf{b}]$ as $\mathsf{a} \times \mathsf{b}$, with $\mathsf{a} \times \mathsf{b} \times \mathsf{c}$ meaning $\mathsf{a} \times (\mathsf{b} \times \mathsf{c})$

2.2. Infinite Lists

Another type necessary for the natural number type is infinite lists, given by

$$\mathsf{InfList}[\mathsf{t}] \triangleq \mu \mathsf{v}.\mathsf{t} \times \mathsf{v}$$

Observe that $\mu\mathsf{v}.\mathsf{t}\times\mathsf{v} = \mathsf{t}\times(\mu\mathsf{v}.\mathsf{t}\times\mathsf{v}) = \mathsf{t}\times(\mathsf{t}\times(\ldots))$ and that $\forall\mathsf{v}.\mathsf{t}\times(\mathsf{t}\times\mathsf{v}) \geq \mathsf{t}\times(\mathsf{t}\times\mathsf{InfList}[\mathsf{t}])$.

3. Natural numbers

While one could use Church numerals having the type $\forall \mathsf{t}.(\mathsf{t} \rightarrow \mathsf{t}) \rightarrow \mathsf{t} \rightarrow \mathsf{t}$, this type does not divide conveniently into subtypes. In this section I will present a natural numbers type that divides into a wide collection of subtypes. The natural number n will be represented by the n^{th} projection from infinite lists:

$$\mathsf{Nat} \triangleq \forall \mathsf{t}.\mathsf{InfList}[\mathsf{t}] \rightarrow \mathsf{t}$$

We shall write $\underline{n}$ for this representation of the natural number n. Informally, $\underline{n}$: $(e_0, (e_1, \ldots (e_n, (e_{n+1}, \ldots)))) \mapsto e_n$. So we have that $\underline{0} \triangleq \lambda l.\mathit{left}\ l$ and $\underline{1} \triangleq \lambda l.\mathit{left}\ (\mathit{right}\ l)$. But now observe that $\underline{0} : \forall \mathsf{a}, \mathsf{b}.(\mathsf{a}\times\mathsf{b}) \rightarrow \mathsf{a}$, which $\geq \mathsf{Nat}$. Similarly $\underline{1} : \forall \mathsf{a}, \mathsf{b}, \mathsf{c}.(\mathsf{a}\times\mathsf{b}\times\mathsf{c}) \rightarrow \mathsf{b}$, and this $\geq \mathsf{Nat}$. In general $\underline{n} : \forall \mathsf{t}_0, \ldots \mathsf{t}_n, \mathsf{u}.(\mathsf{t}_0 \times \ldots \times \mathsf{t}_n \times \mathsf{u}) \rightarrow \mathsf{t}_n$. We shall refer to this type for each $\underline{n}$ as Single_n, and for every n, $\mathsf{Single}_n \geq \mathsf{Nat}$. Furthermore, both $\underline{0}$ and $\underline{1}$ have the type $\mathsf{ZeroOne} \triangleq \forall \mathsf{b}, \mathsf{c}.(\mathsf{b} \times \mathsf{b} \times \mathsf{c}) \rightarrow \mathsf{b}$. So $\mathsf{ZeroOne}$ is a subtype of Nat containing $\underline{0}$ and $\underline{1}$, but not $\underline{2}$, because $\mathsf{Single}_2 \not\geq \mathsf{ZeroOne}$.

The successor function is given by $succ \triangleq \lambda n.\lambda l.n(right\ l) : \forall \mathsf{a}, \mathsf{b}, \mathsf{c}.(\mathsf{a} \to \mathsf{b}) \to (\mathsf{c} \times \mathsf{a}) \to \mathsf{b}$. Notice that $succ : \mathsf{Single}_n \to \mathsf{Single}_{n+1}$. It is also worth looking at the predecessor function defined by $pred \triangleq \lambda n.\lambda l.n\,(\perp, l) : \forall \mathsf{a}, \mathsf{b}, \mathsf{c}.(\mathsf{a} \times \mathsf{b} \to \mathsf{c}) \to \mathsf{b} \to \mathsf{c}$, where $\perp$ is generated by, for example, the fixpoint of the identity.

Clearly any finite subset $\mathbf{S}$ of the natural numbers can be represented as a type $\mathsf{T}_\mathbf{S} \triangleq \forall \mathsf{t}_0, \ldots, \mathsf{t}_n, \mathsf{u}, \mathsf{r}.(\mathsf{V}_0 \times \ldots \times (\mathsf{V}_n \times \mathsf{u})) \to \mathsf{r}$, where n is the largest number in $\mathbf{S}$, and V_i is t_i if $i \notin \mathbf{S}$ and r if $i \in \mathbf{S}$. Again $\mathsf{T}_\mathbf{S} \geq \mathsf{Nat}$, so if $n \in \mathbf{S}$, then $\underline{n} : \mathsf{T}_\mathbf{S}$. Certain other subsets can be represented, for example the set of even natural numbers corresponds to the type $\forall \mathsf{a}, \mathsf{b}.(\mu \mathsf{t}.\mathsf{a} \times (\mathsf{b} \times \mathsf{t})) \to \mathsf{a}$.

4. Records

Some languages (such as Cardelli's Amber) have record types with 'multiple inheritance' [Cardelli 1985]. A record type is written $\mathsf{Rec}\{f_1 : \mathsf{t}_1, \ldots, f_n : \mathsf{t}_n\}$, which stands for a record type with fields named $f_1 \ldots f_n$ of types $\mathsf{t}_1 \ldots \mathsf{t}_n$ respectively. A value of such a type is written $\{f_1 = e_1, \ldots, f_n = e_n\}$ with selection operations $f_i\,\mathbf{Of}\,\{f_1 = e_1, \ldots, f_i = e_i, \ldots, f_n = e_n\} = e_i : \mathsf{t}_i$. The order in which the fields are presented is immaterial, so for example $\{snoo = 1, izzy = 2\} = \{izzy = 2, snoo = 1\}$.

Inheritance just means that we have

$$\mathbf{Rec}\{f_1 : \mathsf{t}_1, \ldots, f_n : \mathsf{t}_n, g_1 : \mathsf{u}_1, \ldots, g_m : \mathsf{u}_m\} \geq \mathbf{Rec}\{f_1 : \mathsf{t}_1, \ldots, f_n : \mathsf{t}_n\}.$$

We can simulate this behaviour in $\forall, \to, \mu$ by means of records with fields numbered by the natural numbers of the previous section.

4.1. Existentially quantified types

In order to model the inheritance properties correctly it is necessary to model a type that corresponds to forgetting all the type information about an object. If the type system included an existential quantifier, one might expect that for any object x, $x : \exists \mathsf{t}.\mathsf{t}$, so that for any type t, $\mathsf{t} \geq \exists \mathsf{t}.\mathsf{t}$. While it would be possible to include an existential quantifier with this property, it would not be desirable, since it would have the effect of hiding type errors. Nor is it necessary, since it can be modelled in the usual way, with $\exists \mathsf{t}.\mathsf{T}$ replaced by $\forall \mathsf{r}.(\forall \mathsf{t}.\mathsf{T} \to \mathsf{r}) \to \mathsf{r}$, which I will write as $\Sigma \mathsf{t}.\mathsf{T}$. Now for $\exists \mathsf{t}.\mathsf{t}$ we can use $\Sigma \mathsf{t}.\mathsf{t} = \forall \mathsf{r}.(\forall \mathsf{t}.\mathsf{t} \to \mathsf{r}) \to \mathsf{r}$, and although this is not related to every type, each type T can be transformed into $\Sigma \mathsf{t}.\mathsf{T}$ (even if t is not free in T) and $\Sigma \mathsf{t}.\mathsf{T} \geq \Sigma \mathsf{t}.\mathsf{t}$. An object of type u can be turned into an object of type $\Sigma \mathsf{t}.\mathsf{u}$ by application of $sigma \triangleq (\lambda x.\lambda f.f\ x \upharpoonright \forall \mathsf{u}.\mathsf{u} \to \Sigma \mathsf{t}.\mathsf{u})$.

4.2. Numbered records

Since there is only a countable collection of names for fields, we can assume that there is a translation between fieldnames and numerals, and consider only types of the form $\mathbf{Rec}\{\underline{n}_1 : \mathsf{t}_{\underline{n}_1}, \ldots, \underline{n}_m : \mathsf{t}_{\underline{n}_m}\}$, and regard this as a shorthand for the type generated by $\mathsf{F}_1 \times \ldots \times \mathsf{F}_{max} \times \mathsf{InfList}[\Sigma \mathsf{t}.\mathsf{t}]$, where max is the largest element of $\{\underline{n}_1 \ldots \underline{n}_m\}$, $\mathsf{F}_i = \Sigma \mathsf{s}.\mathsf{t}_i$

if $i \in \{\underline{n}_1 \ldots \underline{n}_m\}$ and $\mathsf{F}_i = \Sigma\mathsf{s}.\mathsf{s}$ otherwise. Correspondingly, $\{\underline{n}_1 = e_1, \ldots \underline{n}_m = e_m\}$ is represented by $(F_1, \ldots F_{max}, \perp)$, where $F_i = \mathit{sigma}\ e_i$ if $i \in \{\underline{n}_1 \ldots \underline{n}_m\}$ and $\perp\lceil\Sigma\mathsf{t}.\mathsf{t}$ otherwise.

This gives us the required properties, since an absent field gives an element of type $\Sigma\mathsf{t}.\mathsf{t}$, and the comparison of record types comes from the pointwise comparison of the fields.

5. Conclusions

The advantage of natural numbers defined as above is that one need only provide one collection of constant symbols to represent constants of all subsets of the natural numbers. A similar arrangement can be made so that natural number constants are also integers. Although the use of a unary representation is impractical, it is possible to arrange similar relationships between numbers represented as lists of booleans. Here the subsets that can be taken correspond to limiting the length of the list — which fits nicely with limited word lengths on computers.

The formulation of record types gives some insight into what can be done with records. For example, field names are first class objects (they are just natural numbers).

Appendix 1: Typing Rules

This appendix describes the rules for typing expressions and relating types. Both the systems require the following ground rules:

1. Basic inference rules

Rules $B1$ and $B2$ apply to both the relationship between types and typing of terms, with Γ being a set of assumptions of the form ϕ where ϕ is either $\mathsf{T}_1 \geq \mathsf{T}_2$ or $e : \mathsf{T}$.

Assumption

$$\Gamma, \phi \vdash \phi \qquad B1$$

Weakening

$$\frac{\Gamma \vdash \phi_1}{\Gamma, \phi_2 \vdash \phi_1} \qquad B2$$

2. The Relation of generality between types

The relation $\mathsf{T}_1 \geq \mathsf{T}_2$ is intended to mean that any object of type T_1 may validly be used in any situation where an object of type T_2 may validly be used. So $\mathsf{T}_1 \geq \mathsf{T}_2 \& x : \mathsf{T}_1 \Rightarrow x : \mathsf{T}_2$, which fact is expressed in rule *V5*.

Rules $R1$ to $R8$ below define the relation $\geq$. V_n are type variables, T_n are arbitrary types (possibly with free variables), Γ stands for a set of assumptions each of which is of the form $\mathsf{T}_1 \geq \mathsf{T}_2$ and $\geq$ is as above.

Reflexivity

$$\Gamma \vdash \mathsf{T} \geq \mathsf{T} \qquad R1$$

Transitivity

$$\frac{\Gamma \vdash \mathsf{T}_1 \geq \mathsf{T}_2, \quad \Gamma \vdash \mathsf{T}_2 \geq \mathsf{T}_3}{\Gamma \vdash \mathsf{T}_1 \geq \mathsf{T}_3} \qquad R2$$

Instantiation

$$\Gamma \vdash \forall \mathsf{V}.\mathsf{T}_1 \geq \mathsf{T}_1[\mathsf{T}_2/\mathsf{V}] \qquad R3$$

(Expressions of the form $\mathsf{T}_1[\mathsf{T}_2/\mathsf{V}]$ mean " T_1 with every free occurrence of V replaced by T_2, with bound variables in T_1 renamed in such a way as to avoid variable capture.")

Generalisation

$$\frac{\Gamma \vdash \mathsf{T}_1 \geq \mathsf{T}_2}{\Gamma \vdash \mathsf{T}_1 \geq \forall \mathsf{V}.\mathsf{T}_2} \qquad \mathsf{V} \text{ not free in } \mathsf{T}_1 \text{ or } \Gamma \qquad R4$$

Function

$$\frac{\Gamma \vdash \mathsf{T}_3 \geq \mathsf{T}_1, \quad \Gamma \vdash \mathsf{T}_2 \geq \mathsf{T}_4}{\Gamma \vdash \mathsf{T}_1 \to \mathsf{T}_2 \geq \mathsf{T}_3 \to \mathsf{T}_4} \qquad R5$$

Note contrapositivity on the left of $\to$

Result

$$\Gamma \vdash (\forall \mathsf{V}.\mathsf{T}_1 \to \mathsf{T}_2) \ \geq \ \mathsf{T}_1 \to \forall \mathsf{V}.\mathsf{T}_2 \qquad \mathsf{V} \text{ not free in } \mathsf{T}_1 \qquad R6$$

Recursion

$$\frac{\Gamma, (\mu \mathsf{v}.\mathsf{T}) \geq \mathsf{T}_1 \vdash \mathsf{T}[\mu \mathsf{v}.\mathsf{T}/\mathsf{v}] \geq \mathsf{T}_1}{\Gamma \vdash (\mu \mathsf{v}.\mathsf{T}) \geq \mathsf{T}_1} \qquad \mathsf{T} \neq \mathsf{v} \qquad R7a$$

$$\frac{\Gamma, \mathsf{T}_1 \geq (\mu \mathsf{v}.\mathsf{T}) \vdash \mathsf{T}_1 \geq \mathsf{T}[\mu \mathsf{v}.\mathsf{T}/\mathsf{v}]}{\Gamma \vdash \mathsf{T}_1 \geq (\mu \mathsf{v}.\mathsf{T})} \qquad \mathsf{T} \neq \mathsf{v} \qquad R7b$$

3. Rules for Type-Validity of Expressions

This section presents the rules to which valid Ponder programmes must conform. In general a programme will consist of a ‘casted’ expression, the type of which is determined by the environment in which the programme is intended to run.

An expression p is type-valid if a statement of the form $p : \mathsf{T}$ for some T may be proved within the following rules. Note that although all untyped lambda terms may be given the type $\mu\mathsf{t}.\mathsf{t} \to \mathsf{t}$ the presence of cast expressions $e\lceil\mathsf{t}$ means that not all typings are valid.

Γ is a set of assumptions as before but may also include assumptions of the form $v : \mathsf{T}$.

Application

$$\frac{\Gamma \vdash e_1 : (\mathsf{T}_1 \to \mathsf{T}_2) \quad \Gamma \vdash e_2 : \mathsf{T}_1}{\Gamma \vdash e_1 \ e_2 : \mathsf{T}_2} \qquad V1$$

Function

$$\frac{\Gamma, v : \mathsf{T}_1 \vdash e : \mathsf{T}_2}{\Gamma_1 \vdash (\lambda v.e : \mathsf{T}_1 \to \mathsf{T}_2)} \qquad \text{where } \Gamma = \Gamma_1 - \{v : \mathsf{T} \mid \mathsf{T} \text{ is a type}\} \qquad V2$$

Cast

$$\frac{\Gamma \vdash e : \mathsf{T}_1}{\Gamma \vdash (e\lceil \mathsf{T}_1) : \mathsf{T}_1} \qquad V3$$

Generalisation

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash e : \forall \mathsf{V}.\mathsf{T}} \qquad \mathsf{V} \text{ not free in } \Gamma \qquad V4$$

Restriction

$$\frac{\Gamma \vdash e : \mathsf{T}_1 \quad \Gamma \vdash \mathsf{T}_1 \geq \mathsf{T}_2}{\Gamma \vdash e : \mathsf{T}_2} \qquad V5$$

Bibliography

[Cardelli 1985]:
Luca Cardelli
A Semantics of Multiple Inheritance
Semantics of Datatypes, Lecture Notes in Computer Science 173 1985

[Fairbairn 1982]:
Jon Fairbairn
Ponder and its type system
Technical Report 31, Cambridge University Computer Laboratory 1982

[MacQueen 1982]:
D.B. MacQueen, Ravi Sethi,
A Semantic Model of Types for Applicative Languages,
Symposium on Lisp and Functional Programming 1982

[MacQueen 1984]:
D. B. MacQueen, R. Sethi, G. Plotkin,
An Ideal Model for Recursive Polymorphic Types,
Eleventh Annual ACM Symposium on Principles of Programming Languages 1984

[Milner 1978]:
R. Milner
A Theory of Type Polymorphism in Programming
Journal of Computer and System Sciences Volume 17 No.3, December 1978

[Reynolds 1974]:
JC Reynolds
Towards a Theory of Type Structure
Proceedings Coloque sur la Programmation, Springer Lecture Notes in Computer Science 19 1974