Technical Report

UCAM-CL-TR-203
ISSN 1476-2986

Number 203

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Subtyping in Ponder

(preliminary report)

Valeria C.V. de Paiva

August 1990

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

Subtyping in **Ponder** (Preliminary Report)

Valeria C. V. de Paiva

August 22, 1990

Abstract

This note starts the formal study of the type system of the functional language **Ponder**. Some of the problems of proving soundness and completeness are discussed and some preliminary results, about fragments of the type system, shown.

It consists of 6 sections. In section 1 we review briefly **Ponder**'s syntax and describe its typing system. In section 2 we consider a very restricted fragment of the language for which we can prove soundness of the type inference mechanism, but not completeness. Section 3 describes possible models of this fragment and some related work. Section 4 describes the type-inference algorithm for a larger fragment of **Ponder** and in section 5 we come up against some problematic examples. Section 6 is a summary of further work.

Introduction

Like many other terms in Computer Science subtyping means different things to different people. One of its fashionable meanings these days is related to object-oriented programming languages and the typical example in this case is to think of 'subrecords' as a subtype of 'records', [Card84].

In this note we are thinking of subtyping in a pure functional language with polymorphism. The example we have in mind, most of the time, is the relationship between a type like $\forall T.T \to T$ (polymorphic identity type) and a type $Bool \to Bool$. Whenever a function in **Ponder** needs an argument of type $\forall T.T \to T$, $Bool \to Bool$ can be used and so we say that $\forall T.T \to T$ is more general than $Bool \to Bool$, which we write $\forall T.T \to T \geq Bool \to Bool$. But we also consider the possibility of a type variable V being more general than another type variable U, $V \geq U$.

The aim of this note was to describe **Ponder**'s [cf. Fair85] typing system, show the soundness of its type inference mechanism and a kind of completeness result for it. From that we wanted to go on to discuss different semantic models for **Ponder** and their adequacy and/or shortcomings. But the problem of type inference for a language like **Ponder** is much more difficult than we had previously realised, so this note now intends to point out some of the problems and possible solutions, yet to be worked out.

1 The Typing System of Ponder

In this section we review some of the syntax and typing system of **Ponder**. The main reference for **Ponder** is Jon Fairbairn's thesis, *Design and Implementation of a Simple Typed Language based on the Lambda-Calculus* 1985, where more details can be found.

Ponder's typing system is similar to the typing system of **Fun**, Cardelli and Wegner's language and also similar to the system $\mathbf{F}_{\leq}$ of Curien-Ghelli. It is, in some sense, an extension of the polymorphic lambda-calculus of Girard and Reynolds.

The syntax of type expressions is given by

$$T ::= V \mid T \rightarrow S \mid \forall V.T \mid \mu V.T$$

where

- V is a type variable from (a given set of type variables) $TyVars$.
- $T \rightarrow S$ is the type of functions from type T to type S.
- $\forall V.T$ is for all quantification over the type expression T.
- $\mu V.T$ is the recursive type defined by the type expression T.

Raw-terms are given by

$$e ::= x \mid x : T \rightarrow e \mid e_1 e_2 \mid e : T \mid \Lambda V.e$$

where

- x is an individual variable in $Vars$,
- $(x : T \rightarrow e)$ is **Ponder** syntax for λ-abstraction,
- $(e_1 e_2)$ stands for function application,
- $(e : T)$ is a *cast expression*, that is, an expression that the programmer would like to have type T,
- $(\Lambda V.e)$ is the second-order abstraction of the expression e with respect to the type variable V.

The constants λ, $\forall$, μ and Λ bind their variable in their second argument and terms are considered up to α-conversion as usual.

For the time being we will not consider the recursive types. We call the system without recursive types **Ponder**$_-$.

We also adopt some conventions for meta-variables:

- e,e' are expressions
- T,S are type expressions
- V,U are type variables
- x,y are individual variables
- C,C' stand for sets of subtyping assumptions of the form $T \geq S$ where T and S are type expressions and at least one of them is a type variable.
- Γ stands for sets of typing assumptions of the form $(x : T)$ where x is an individual variable and T is a type expression.

Clearly $T \rightarrow S$ is the type of a first-order abstraction $(x : T \rightarrow e)$ where e has type S and $\forall V.T$ is the type of an expression of the form $\Lambda V.e$ where T is the type of e.

The typing and subtyping relations are specified via the following judgements:

$$\begin{array}{ll} C & \vdash T \geq S \quad \text{type } T \text{ is more general than } S \text{ under subtyping constraints } C \\ \Gamma, C & \vdash e : T \quad \text{expression } e \text{ has type } T \text{ in context } (\Gamma, C) \end{array}$$

A context is a pair (Γ, C) where

- 'Γ' is a list of type declarations $(x_1 : T_1, x_2 : T_2, \ldots x_m : T_m)$, where $x_1, x_2, \ldots, x_m$ are *distinct* individual variables and $T_1, \ldots T_m$ are type expressions with type variables in $\mathcal{V}$
- 'C' is a sequence of subtyping assumptions $(T_1 \geq S_1, \ldots T_k \geq S_k)$, where T_j, S_j are type expressions and either T_j or S_j is a type variable.

To characterise the allowed expressions we have rules for subtypes and typing rules for expressions.

Rules for Subtypes

$$(ass) \quad \frac{}{C, T \geq S \vdash T \geq S}$$

$$(refl) \quad \frac{}{C \vdash T \geq T}$$

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

$$(arrow) \quad \frac{C \vdash T_1 \geq T_2 \qquad C \vdash S_1 \geq S_2}{C \vdash T_2 \to S_1 \geq T_1 \to S_2}$$

$$(inst) \quad \frac{}{C \vdash \forall V.T \geq T[S/V]}$$

$$(gen) \quad \frac{C \vdash T \geq S}{C \vdash T \geq \forall V.S} \quad V \notin ftv(T), V \notin ftvC$$

$$(res) \quad \frac{}{C \vdash \forall V.T \to S \geq T \to \forall V.S} \quad V \notin ftv(T)$$

The notation $T[S/V]$ in rule $(inst)$, for instantiation, means the type expression T with the free ocurrences of the type variable V replaced by the type expression S.

Note that the usual rule relating quantifiers to subtyping, for instance in [CW'85]

$$\frac{C \vdash T \geq S}{C \vdash \forall V.T \geq \forall V.S} \quad V \notin ftv(C)$$

can be derived as follows:

- if V is free in T, then it is not free in $\forall V.T$ and we can apply rule (gen):

$$\dfrac{\dfrac{\dfrac{}{C \vdash \forall V.T \geq T} \qquad C \vdash T \geq S}{C \vdash \forall V.T \geq S}\,(trans)}{C \vdash \forall V.T \geq \forall V.S}\,(gen)$$

Typing Rules for Expressions

$$(var)\quad \frac{}{\Gamma, x:T \vdash x:T}$$

$$(\lambda)\quad \frac{x:T, \Gamma \vdash e:S}{\Gamma \vdash (x:T \to e):T \to S}$$

$$(ap)\quad \frac{\Gamma \vdash e:T \to S \qquad \Gamma \vdash e':T}{\Gamma \vdash ee':S}$$

$$(\Lambda)\quad \frac{V, \Gamma \vdash e:T}{\Gamma \vdash \Lambda V.e:\forall V.T}\ V \notin ftv(\Gamma)$$

$$(cast)\quad \frac{\Gamma \vdash e:T}{\Gamma \vdash (e:T):T}$$

$$(sub)\quad \frac{\Gamma \vdash e:T \qquad C \vdash T \geq S}{C, \Gamma \vdash e:S}$$

Note that the familiar rule of application of types

$$\frac{\Gamma \vdash e:\Lambda V.T}{\Gamma \vdash e_{[S]}:T(S/V)}\ (App)$$

is missing, but its role is played by the subtyping rules (*inst*), (*gen*) and (*res*) together with rule (*sub*).

Two easy examples of derivations in this system:

$$\frac{(var)\dfrac{}{x:T \vdash x:T} \qquad (ass)\dfrac{}{T \geq S \vdash T \geq S}}{x:T, T \geq S \vdash x:S}\ (sub)$$

$$\frac{\dfrac{\dfrac{}{x:T \vdash x:T}\ (var)}{\vdash (x:T \to x):T \to T}\ (\lambda)}{\vdash \Lambda T.(x:T \to x):\forall T.T \to T}\ (\Lambda)$$

This formal inference system looks deceptively harmless, and systems like **Fun** or $\mathbf{F}_{\geq}$ or $\mathbf{F}^{++}$ seem supersets of it. But there is a difference, because in all these systems, subtyping constraints have to be of the form *type variable* $\geq T$.

2 "Propositional" Ponder

The typing system described for **Ponder_** seems very reasonable and even a familiar one. But despite its similarities with well-known systems, it is not as usual as it seems.

We consider in this section **Ponder_** without quantification over types. We call this system **Ponder__** or $\mathbf{Ponder}_{prop}$ or sometimes simply **PP**.

2.1 The formal system

Types are given by the syntax:

$$T ::= V \mid T \to S$$

where V is in the set of type variables $TyVars$. Raw terms given by:

$$e ::= x \mid x : T \to e' \mid e_1 e_2 \mid e : T$$

where x is an individual variable in $Vars$ and T is a type expression.

Contexts consist of a set Γ of typing declarations of the form $(x_i : T_i)$ and a set C of subtyping constraints of the form $T_i \geq S_i$, where either T_i or S_i is a type variable.

The rules for subtyping are restricted to:

$$(ass) \quad \frac{}{C, T \geq S \vdash T \geq S}$$

$$(refl) \quad \frac{}{C \vdash T \geq T}$$

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

$$(arrow) \quad \frac{C \vdash T_1 \geq T_2 \qquad C \vdash S_1 \geq S_2}{C \vdash T_2 \to S_1 \geq T_1 \to S_2}$$

The rules for typing terms:

$$(var) \quad \frac{}{\Gamma, x : T \vdash x : T}$$

$$(\lambda) \quad \frac{x : T, \Gamma \vdash e : S}{\Gamma \vdash (x : T \to e) : T \to S}$$

$$(app) \quad \frac{\Gamma \vdash e : T \to S \qquad \Gamma \vdash e' : T}{\Gamma \vdash ee' : S}$$

$$(cast) \quad \frac{\Gamma \vdash e : T}{\mathcal{V}, \Gamma \vdash (e : T) : T}$$

$$(sub) \quad \frac{\Gamma \vdash e : T \qquad C \vdash T \geq S}{C, \Gamma \vdash e : S}$$

Say $\Gamma, C \vdash e : T$ holds iff there's a proof of it using the rules above and read it as " e has type T in the context (Γ, C)". Note that if $\Gamma, C \vdash e : T$ holds, Γ assigns types to all free variables of e and further, for any Γ, $\Gamma, C \vdash e : T$ holds iff $\Gamma', C \vdash e : T$, where Γ' is the restriction of Γ to the free variables of e.

Two examples of derivations

In the examples both T and S are type variables.

$$\dfrac{\dfrac{}{x:T\vdash x:T}\,(var) \qquad \dfrac{}{T\geq S\vdash T\geq S}\,(ass)}{x:T,T\geq S\vdash x:S}\,(sub)$$

$$\dfrac{\dfrac{}{x:T\vdash x:T}\,(var)}{\vdash (x:T\to x):T\to T}\,(\lambda)$$

We want to make some observations about the logic of this fragment. In particular we want to discuss the logic of the subtyping rules (*ass*) to (*arrow*) above.

- The first thing to note is that it looks like a weak "logic of implication".
- As the set C consists of subtyping assumptions, the first rule (*assumption*) says we have contraction and weakening in this logic:

$$\dfrac{}{T\geq S,T\geq S\vdash T\geq S} \qquad \dfrac{}{T_1\geq S_1,T\geq S\vdash T\geq S}$$

- Rule 2 merely gives us some basic axioms.
- Rule 3 can be seen as a *cut-rule*.
- Rule 4 (*arrow*) can be transformed into two rules:

$$(\to_1)\quad \dfrac{C\vdash T_1\geq T_2}{C\vdash T_2\to S\geq T_1\to S}$$

$$(\to_2)\quad \dfrac{C\vdash S_1\geq S_2}{C\vdash T\to S_1\geq T\to S_2}$$

- There are other rules, which would make sense with the intuitive notion of typing as subset inclusion, where $\geq$ corresponds to $\subseteq$, for instance:

$$\dfrac{C\vdash T\geq S_1 \qquad C\vdash T\geq S_2}{C\vdash T\geq S_1\cap S_2}$$

which are not explicitly in the system. But other rules, which logically would make perfect sense, don't seem so reasonable here:

$$\dfrac{C\vdash T_1\geq S}{C\vdash T_1\cap T_2\geq S}\,(?) \qquad \dfrac{C\vdash T_2\geq S}{C\vdash T_1\cap T_2\vdash S}\,(?)$$

- For a union of types $T_1\cup T_2$, we could have:

$$\dfrac{C\vdash T_1\geq S}{C\vdash T_1\cup T_2\geq S} \qquad \dfrac{C\vdash T_2\geq S}{C\vdash T_1\cup T_2\geq S}$$

$$\dfrac{C\vdash T\geq S_1\cup S_2}{C\vdash T\geq S_1} \qquad \dfrac{C\vdash T\geq S_1\cup S_2}{C\vdash T\geq S_2}$$

Note that both Cardelli [Card84]and MacQueen,Plotkin and Sethi [MPS86] describe intersection and union of types, but they are different from the ones above.

The typing system of **Ponder** does not allow type or term constants. The rationale behind it was that one could use closed terms to code up types like integers, booleans or records [Fair89], so there would be no need for type constants.

But even if it is possible to do without these traditional type constants, if they do not introduce any complications to the system I believe they should be allowed.

One interesting question is what happens to the type inference mechanism of **Ponder** in this restricted fragment with no for all quantification of types. This is discussed in the next section.

2.2 The type inference mechanism in Ponder_{prop}

In this restricted fragment, the type inference mechanism consists of five functions, ***type-check***, *GE*, *OR*, *min* and *clos*. In this section we only discuss ***type-check***, *GE* and *clos*. The main function of the type-inference mechanism, ***type-check***, takes two arguments:

- a context (C_0, Γ)
- a Ponder expression '*e*'

and returns either *FAIL* or a type expression T and a set of constraints C with the intended meaning that (C, Γ) proves that '*e*' has type T.

A context consists of two parts. The component Γ of the context has type declarations of the form $(x_i : T_i)$ where x_i are *distinct* individual variables and T_i are type expressions. The component C of the context consists of subtyping assumptions (or constraints) of the form $T_i \geq S_i$, where T_i, S_i are type expressions. Note that this is slightly more general than contexts in the formal system in 2.1.

Definition 1 *The function* ***type-check***(C, Γ, e) *is defined by cases on the structure of the Ponder expression 'e', as follows:*

var *if the expression is a variable 'x', its type is the one given in the* Γ *part of the context, that is:*

$$\textbf{type-check}(C, \Gamma, x) = (C, T) \text{ if } (x : T) \text{ is in } \Gamma, \text{ FAIL otherwise.}$$

abs *if the expression is a* λ*-abstraction* $(x : T \rightarrow e)$*, its type is* $T \rightarrow S$*, provided the body 'e' has type* S *in the context augmented with* $\{x : T\}$*, so:*

$$\textbf{type-check}(C_0, \Gamma, x : T \rightarrow e) = (C, T \rightarrow S)$$
$$\text{if } \textbf{type-check}(C_0, \Gamma \cup \{x : T\}, e) = (C, S)$$

cast *if the expression is a cast* $(e : T)$*, it has type* T*, provided the body 'e' can be type-checked to* S *and type* S *is more general than* T*. To prove that* $S \geq T$ *we build up the set* $GE(S, T)$ *and take its transitive closure.*

$$\textbf{type-check}(C_0, \Gamma, e : T) = (clos[GE(S, T) \cup C], T)$$
$$\text{if } \textbf{type-check } (C_0, \Gamma, e) = (C, S)$$

appl *if the expression is an application* $(e_1 e_2)$*, its type is the variable type* R*, provided that '*e_1*' can be type-checked to* T_1*, '*e_2*' can be type-checked to* T_2 *and* $T_1 \geq T_2 \rightarrow R$*:*

$$\textbf{type-check}(C_0, \Gamma, e_1 e_2) = (clos[C_1 \cup C_2 \cup GE(T_1, T_2 \rightarrow R)], R)$$
if $\textbf{type-check } (C_0, \Gamma, e_1) = (C_1, T_1)$
and $\textbf{type-check } (C_0, \Gamma, e_2) = (C_2, T_2)$
and R *is a fresh type variable*

Note that the set C_0 of subtyping assumptions may start empty or not. But if clauses **cast** and **appl** are used it will increase in size, so $C_0 \subseteq C$. Also Γ may start empty or not, but whenever rule **[abs]** is used it increases.

Next to define the function GE, remember that type expressions are either type variables or arrow types.

Definition 2 *Given type expressions T and S the function $GE(T,S)$ returns a set of subtyping assumptions C. The function $GE(T,S)$ is defined by cases on the structure of T and S as follows:*

- *If T is a type variable V*
  - *C1. If S is any type expression, $GE(V,S) = \{V \geq S\}$.*
- *If S is a type variable U,*
  - *C2. If T is any type expression, $GE(T,U) = \{T \geq U\}$.*
- *If T is an arrow type $T_1 \rightarrow T_2$*
  - *C3. If S is an arrow type $S_1 \rightarrow S_2$, $GE(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = GE(S_1,T_1) \cup GE(T_2,S_2)$.*

Definition 3 *The function clos (for transitive closure under GE) takes a set of subtyping assumptions C and returns clos C, the least set such that:*

- $C \subseteq clos\ C$
- $\{T \geq V, V \geq S\} \subseteq clos\ C \Rightarrow GE(T,S) \subseteq clos\ C$

If we apply *clos* to the set

$$C = \{A \geq A \rightarrow A, A \rightarrow A \geq A\}$$

as *clos* acts when the "middle" is a type variable, it only adds $GE(A \rightarrow A, A \rightarrow A) = \{A \geq A\}$ to the set C.

Note that the algorithm always halt, as the three functions only decrease the sizes of the expressions they're dealing with.

Examples of the mechanism

1. Suppose I, B, V denote type variables (you might think of I as the type of *Integers*, B as *Booleans* and V as a generic type variable). Suppose also that f and x have been declared with types $(I \rightarrow I) \rightarrow B$ and $V \rightarrow V$, respectively, in Γ and C_0 is empty.

   If we apply the function ***type-check*** to try to infer a type for fx in the context (Γ, C_0) we have:

$$\begin{array}{lcl} \textit{type-check}(\Gamma, \emptyset, fx) & = & (C, R) \\ \textit{type-check}(\Gamma, \emptyset, f) & = & (\emptyset, (I \rightarrow I) \rightarrow B) \\ \textit{type-check}(\Gamma, \emptyset, x) & = & (\emptyset, V \rightarrow V) \end{array}$$

   where

$$C = clos(GE((I \rightarrow I) \rightarrow B, V \rightarrow V \rightarrow R))$$

   The mechanism will call the GE with parameters $(I \rightarrow I) \rightarrow B$ and $(V \rightarrow V) \rightarrow R$:

$$\begin{array}{lcl} GE((I \rightarrow I) \rightarrow B, (V \rightarrow V) \rightarrow R) & = & GE(V \rightarrow V, I \rightarrow I) \cup GE(B,R) \\ & = & GE(I,V) \cup GE(V,I) \cup GE(B,R) \\ & = & \{I \geq V, V \geq I, B \geq R\} \end{array}$$

The function *clos* only add to the set C the subtyping assumptions $\{I \geq I, V \geq V\}$, which are, in some sense, already there.

The intuition here is that the set of subtyping constraints

$$C = \{I \geq V, V \geq I, B \geq R, I \geq I, V \geq V\}$$

'proves' $(I \to I) \to B \geq (V \to V) \to R$. A derivation in the system $\mathbf{Ponder}_{prop}$ is:

$$\dfrac{\dfrac{\dfrac{}{C \vdash I \geq V}\,(ass) \quad \dfrac{}{C \vdash V \geq I}\,(ass)}{C \vdash V \to V \geq I \to I}\,(arrow) \quad \dfrac{}{C \vdash B \geq R}\,(ass)}{C \vdash (I \to I) \to B \geq (V \to V) \to R}\,(arrow)$$

Now to obtain one of the possible derivations of $fx : R$ we simply plugg in the derivation above, call it π, in the following:

$$\dfrac{\dfrac{}{\Gamma \vdash f : (I \to I) \to B} \quad \dfrac{\pi}{C \vdash (I \to I) \to B \geq (V \to V) \to R} \qquad \dfrac{}{\Gamma \vdash x : V \to V}}{\Gamma, C \vdash fx : R}$$

Note that if we swap arguments in Γ, that is if

$$\Gamma = \{f : (V \to V) \to B, x : I \to I\}$$

we obtain the same result:

$$\textbf{\textit{type-check}}(\Gamma, \emptyset, fx) = (C, R)$$

where C is as above.

Now for a more complicated example, where the function ***type-check*** calls itself:

2. Suppose T, A and B denote type variables, a, b and f individual variables, the initial subtyping (part of a context) C_0 is empty and

$$\Gamma = \{f : T \to (T \to T), a : A, b : B\}$$

To infer a type for fab in the context above, we apply $\textbf{\textit{type-check}}(\Gamma, \emptyset, fab)$. If it succeeds, we have:

$$\begin{array}{lcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fab) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, fa) & = & (C_1, R_1) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, b) & = & (C_2, R_2) \end{array}$$

where:

- R is a new type variable
- $C = clos[C_1 \cup C_2 \cup GE(R_1, R_2 \to R)]$
- (C_1, R_1) is the result of $\textbf{\textit{type-check}}(\Gamma, \emptyset, fa)$ and
- (C_2, R_2) is the result of $\textbf{\textit{type-check}}(\Gamma, \emptyset, b)$.

But ***type-check***$(\Gamma, \emptyset, b) = (\emptyset, B)$ and ***type-check***$(\Gamma, \emptyset, fa) = (C_1, R_1)$ where R_1 is a type variable and we have to calculate $C_1 = clos[GE(T \to (T \to T), A \to R_1)]$.

$$\begin{aligned} GE(T \to (T \to T), A \to R_1) &= GE(A, T) \cup GE(T \to T, R_1) \\ &= \{A \geq T, T \to T \geq R_1\} \end{aligned}$$

Thus $C_1 = \{A \geq T, T \to T \geq R_1\}$, as *clos* does nothing here.

Note that $C_1 \vdash T \to (T \to T) \geq A \to R_1$ and $\Gamma, C_1 \vdash fa : R_1$. One possible derivation would be:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : T \to (T \to T)} \quad \dfrac{\dfrac{}{C_1 \vdash A \geq T} \quad \dfrac{}{C_1 \vdash T \to T \geq R_1}}{C_1 \vdash T \to (T \to T) \geq A \to R_1}}{\Gamma, C_1 \vdash f : A \to R_1} \quad \dfrac{}{\Gamma \vdash a : A}}{\Gamma, C_1 \vdash fa : R_1}$$

Another possible derivation would be:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : T \to (T \to T)} \quad \dfrac{\dfrac{}{\Gamma \vdash a : A} \quad \dfrac{}{C_1 \vdash A \geq T}}{\Gamma, C_1 \vdash a : T}}{\Gamma, C_1 \vdash fa : T \to T} \quad \dfrac{}{C_1 \vdash T \to T \geq R_1}}{\Gamma, C_1 \vdash fa : R_1}$$

Note that the proof-theory of the system starts showing its ugly head here. Both derivations assume the same hypotheses and have the same number of rules, but the first is in a special form, where subtyping assumptions are used first and that is the one the type inference mechanism "provides". It's an interesting question whether the two proofs are equivalent or not...

Also if one defines semantics in terms of the inference rules used in the derivation of a typing $\Gamma, C \vdash e : T$ some kind of coherence result is needed to show that two different derivations of the same typing give you the same meaning.

Back to the main derivation, $C = clos[C_1 \cup C_2 \cup C_3]$, where $C_2 = \emptyset$ and $C_3 = \{R_1 \geq B \to R\}$. Thus $C_1 \cup C_2 \cup C_3 = \{A \geq T, T \to T \geq R_1, R_1 \geq B \to R\}$ and, this time *clos* does something,

$$C = \{A \geq T, T \to T \geq R_1, R_1 \geq B \to R, T \to T \geq B \to R, B \geq T, T \geq R\}$$

A derivation of $\Gamma, C \vdash fab : R$ can be obtained using either of the derivations π above in:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, C_1 \vdash fa : R_1} \quad \dfrac{}{C_3 \vdash R_1 \geq B \to R}}{\Gamma, C_1, C_3 \vdash fa : B \to R} \quad \dfrac{}{\Gamma \vdash b : B}}{\Gamma, C_1, C_2 \vdash fab : R}$$

Note that both C_1 and C_3 are *good* contexts, which only contain subtyping assumptions of the form $var \geq T_i$ or $T_i \geq var$, but $clos(C_1 \cup C_3)$ is not. Also it is clear that, at least in this simple example, one could "simplify" *clos* C to only the 'essential' assumptions

$$\{A \geq T, B \geq T, T \geq R\}$$

A last thing to note is that we are saying here that for any type T which is less general than both A and B and more general than R, we can prove that *fab* will have type R. The problem is there might not be a type T which is less general than both A and B... That is the problem addressed by functions OR and *min*, see section 2.2.4

2.3 Soundness of the type inference mechanism

In this very restricted fragment of Ponder it is easy to see that the type inference mechanism only infers types which can be deduced using the formal system. To show that we need a lemma proving that the function GE does what its intuitive meaning says it has to.

Lemma 1 *Given two type expressions T and S, the function $GE(T,S)$ returns a set C of subtyping constraints such that $C \vdash T \geq S$.*

Proof: By structural induction. As before, T and S can be either variables or arrow types. We check each one of the possibilities.

- if T is a type variable V
  - case 1: If S is any type expression, by definition, $GE(V,S) = \{V \geq S\}$. The singleton set $\{V \geq S\}$ clearly proves $V \geq S$, using rule assumption (*ass*):

$$\frac{}{V \geq S \vdash V \geq S}\ (ass)$$

- If S is a type variable U,
  - case 2: If T is any type expression, the result is as trivial as case 1, since by definition $GE(T,U) = \{T \geq U\}$.
- if T is an arrow type $T_1 \rightarrow T_2$
  - case 3: If S is an arrow type $S_1 \rightarrow S_2$ then $GE(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = GE(S_1,T_1) \cup GE(T_2,S_2)$. By induction hypothesis there exists C_1, and a derivation π_1 such that $C_1 \vdash S_1 \geq T_1$. Also there exists C_2 and a derivation π_2 such that $C_2 \vdash T_2 \geq S_2$. Then rule *(arrow)*, plus weakening, guarantees that $C_1, C_2 \vdash T_1 \rightarrow T_2 \geq S_1 \rightarrow S_2$.

$$\dfrac{\dfrac{\dfrac{\pi_1}{C_1 \vdash S_1 \geq T_1}}{C_1, C_2 \vdash S_1 \geq T_1}\ (w) \qquad \dfrac{\dfrac{\pi_2}{C_2 \vdash T_2 \geq S_2}}{C_1, C_2 \vdash T_2 \geq S_2}\ (w)}{C_1, C_2 \vdash T_1 \rightarrow T_2 \geq S_1 \rightarrow S_2}\ (arrow)$$

Now using the lemma we want to prove soundness of the type inference mechanism with respect to the typing rules. The proof is by induction on the structure of the expression e. The basis corresponds to e being a variable.

It is clear from the definition of the function ***type-check*** that the functions GE and *clos*, hence lemma 1, are needed in the cases of e a casted expression or e an application.

Theorem 1 *Given a context (C_0, Γ) and an expression in Ponder 'e' such that*

$$\textbf{\textit{type-check}}(C_0, \Gamma, e) = (C, T)$$

there exists a derivation π according to the rules such that π has as its last formula $C, \Gamma \vdash e : T$

BASIS of INDUCTION: If ϵ is a variable, $\textbf{\textit{type-check}}(C_0, \Gamma, x) = (C, T)$, implies, by definition of the function ***type-check*** that the assertion $(x : T)$ is in the context Γ and $C_0 = C$. The derivation π in this case is simply

$$\frac{}{\Gamma, C \vdash x : T}\ (var)$$

Now for the induction step, we have 3 cases:

- If e is a λ-abstraction $(x : T \to e)$, and $\textbf{\textit{type-check}}(C_0, \Gamma, x : T \to e) = (C, T \to S)$, then, by definition, $\textbf{\textit{type-check}}(C_0, \Gamma \cup \{x : T\}, e) = (C, S)$.

  By induction hypothesis there exists a derivation π whose last formulae is $\Gamma, x : T, C \vdash e : S$ and we can use rule (λ) to get a derivation with last formula $\Gamma, C \vdash (x : T \to e) : T \to S$.

$$\frac{\dfrac{\pi}{\Gamma, x : T, C \vdash e : S}}{\Gamma, C \vdash (x : T \to e) : T \to S}\ (\lambda)$$

- If e is a casted expression $(e : T)$ and $\textbf{\textit{type-check}}(C_0, \Gamma, (e : T)) = (C, T)$, then there exists a set of constraints C_1 and a type S such that
  - $\textbf{\textit{type-check}}(C_0, \Gamma, e) = (C_1, S)$ and
  - $C = \mathit{clos}(C_1 \cup GE(S, T))$.

  By induction there exists a derivation π_1 whose last formula is $C_1, \Gamma \vdash e : S$.

  Using the lemma, the function $GE(S, T)$ produces a set of subtyping assumptions C_2 such that $C_2 \vdash S \geq T$, via derivation π_2.

  Using weakening and rule (sub) we derive $\Gamma, C_1, C_2 \vdash e : T$. The set $C = \mathit{clos}[C_1 \cup C_2]$ so it contains $C_1 \cup C_2$, we have $\Gamma, C \vdash e : T$ and using rule $(cast)$ the result.

$$\frac{\dfrac{\dfrac{\pi_1}{C_1, \Gamma \vdash e : S} \qquad \dfrac{\pi_2}{C_2 \vdash S \geq T}}{C_1, C_2, \Gamma \vdash e : T}\ (sub)}{C, \Gamma \vdash (e : T) : T}\ (cast)$$

- If e is an application $e_1 e_2$ and $\textbf{\textit{type-check}}(C_0, \Gamma, e_1 e_2) = (C, R)$ then,
  - $\textbf{\textit{type-check}}(C_0, \Gamma, e_1) = (C_1, T_1)$ and
  - $\textbf{\textit{type-check}}(C_0, \Gamma, e_2) = (C_2, T_2)$ and
  - $C = \mathit{clos}(C_1 \cup C_2 \cup GE(T_1, T_2 \to R))$ and
  - R is a fresh type variable.

  By induction there is a derivation π_1 with last formula $C_1, \Gamma \vdash e_1 : T_1$ and also a derivation π_2 which proves $C_2, \Gamma \vdash e_2 : T_2$.

  By lemma $GE(T_1, T_2 \to R)$ produces a set C_3 such that $C_3 \vdash T_1 \geq (T_2 \to R)$, using derivation π_3. Using rules (sub) and (app) (plus weakening) we conclude $C_1, C_2, C_3, \Gamma \vdash e_1 e_2 : R$.

$$\frac{\dfrac{\dfrac{\dfrac{\pi_1}{C_1, \Gamma \vdash e_1 : T_1} \qquad \dfrac{\pi_3}{C_3 \vdash T_1 \geq (T_2 \to R)}}{C_3, C_1, \Gamma \vdash e_1 : T_2 \to R} \qquad \dfrac{\pi_2}{C_2, \Gamma \vdash e_2 : T_2}}{C_1, C_2, C_3, \Gamma \vdash e_1 e_2 : R}}{C, \Gamma \vdash e_1 e_2 : R}$$

2.4 The 'existential' character of the algorithm

To show soundness of the type inference mechanism for **PP** only the three functions ***type-check***, GE and *clos* are necessary. But functions OR and *min* have a different role to play.

The function OR is similar, in a way, to function GE:

Definition 4 *Given type expressions T and S the function $OR(T,S)$ returns a set of subtyping constraints C. The function OR is defined by cases on the structure of T and S as follows:*

- *If V is a type variable and S any type expression $OR(V,S) = GE(V,S) \cup GE(S,V)$.*
- *If T is an arrow expression $T_1 \to T_2$ and S a type variable V,*

$$OR(T_1 \to T_2, V) = GE(T_1 \to T_2, V) \cup GE(V, T_1 \to T_2)$$

- *If T and S are arrow types $T_1 \to T_2$ and $S_1 \to S_2$, $OR(T_1 \to T_2, S_1 \to S_2) = OR(T_2, S_2)$.*

Function *min* is similar to function *clos*.

Definition 5 *The function min takes a set C and closes it under minimal types, so min C is the least set such that*

- $C \subseteq min\ C$
- $\{T \geq V, S \geq V\} \subseteq min\ C \Rightarrow OR(T,S) \subseteq min\ C$

The intuition behind OR and *min* is that if the algorithm produces, for a given context (Γ, C_0) and an expression e, a set of constraints C of the form

$$C = \{A \geq T, B \geq T\}$$

then even if there exists a derivation of $\Gamma, C \vdash e : S$ there might not exist a type T which is at the same time less general than A and B. To make sure that such a T exists the algorithm calls *min* and OR. These functions interact with the other functions by adding new type contraints. Thus if we apply *min* to the set of subtyping constraints obtained by *clos* C, it might call OR and the set of contraints can only increase.

In the first example above, where

$$C = \{I \geq V, V \geq I, B \geq R, I \geq I, V \geq V\}$$

OR and *min* do nothing.

In the second example above, where

$$C = \{A \geq T, T \to T \geq R_1, R_1 \geq B \to R, T \to T \geq B \to R, B \geq T, T \geq R\}$$

the function *min* C will call $OR(A,B)$, which implies that *both* $A \geq B$ and $B \geq A$ are going to be added to the set.

Note that more subtyping constraints restrict the number of possible typings for expressions in given contexts, but soundness is not affected. The termination of the algorithm is not affected either, as both OR and *min* decrease the size of the expressions they deal with.

2.5 Some kind of Completeness ?

Note that there is no chance of completeness for the type inference mechanism, at least in the usual sense. For instance if $\Gamma = \{x : T\}$ and $C = \{T \geq S\}$, the derivation

$$\frac{\dfrac{}{\Gamma \vdash x : T} \qquad \dfrac{}{C \vdash T \geq S}}{\Gamma, C \vdash x : S}\ (sub)$$

is perfectly valid, but the type inference mechanism will never return type S for the variable x.

But, at least in this simple case, it's possible to define the set of ***possible typings of 'x' in the context*** (Γ, C), call it $PT_{(\Gamma,C)}(x)$. In the example above $PT_{(\Gamma,C)}(x) = \{T, S\}$.

We could try to define (inductively) sets of possible typings of an expresion 'e' in a context (Γ, C). In our example 1 where

$$C = \{I \geq V, V \geq I, B \geq R, I \geq I, V \geq V\}$$

and $\Gamma = \{x : V \to V, f : (I \to I) \to B\}$ there is a finite number of possible typings for x, f and fx. For instance $PT_{(\Gamma,C)}(x) = \{V \to V, I \to I, I \to V, V \to I\}$

But that does not work in general. If $\Gamma = \{x : S\}$ and $C = \{S \geq S \to S, S \to S \geq S\}$ where S is a type variable, it's easy to see that $PT_{(\Gamma,C)}(x) = \{S, S \to S, (S \to S) \to (S \to S), \ldots\}$ an infinite set!!!

It seems that one would like to rule out these loops, without ruling out, as for instance Mitchell does in [M84], arrow types in the set of constraints.

Note that Lemma 1 proves 'soundness' of the function GE for **PP**, but there is no 'completeness', as the derivation

$$\frac{\dfrac{}{C \vdash T_1 \geq T_2} \qquad \dfrac{}{C \vdash T_2 \geq T_3}}{C \vdash T_1 \geq T_3}$$

is valid if for instance $C = \{T_1 \geq T_2, T_2 \geq T_3\}$ but the function GE applied to T_1 and T_3 will only produce the subtyping constraint $\{T_1 \geq T_3\}$.

3 Semantics of Ponder$_{prop}$

The system we are calling **Ponder**$_{prop}$ could well be called simple typed lambda-calculus with subtyping. It has been discussed, or rather, supersets of it have been discussed, in the literature mainly in Cardelli's *A Semantics of Multiple Inheritance*, in Bruce and Longo's *A Modest Model of Records, Inheritance and Bounded Quantification* as well as in Mitchell's *Coercion and Type Inference*. These three papers propose three different models for the calculi they are discussing, and all of them restrict to models of **Ponder**$_{prop}$.

In this section we discuss their approaches, similarities and differences to the one taken in Ponder, as well as semantic models for the system **Ponder**$_{prop}$, called from now on **PP**.

3.1 The Lambda Model

Mitchell's paper *Coercion and Type Inference* presents a system of inference rules that seems very similar to **PP**. We call his system **M**. In our notation, the syntax for type expressions of **M** is:

$$TExp = K \mid V \mid T \to T'$$

and raw terms are given by:

$$Exp = x \mid \lambda x.e \mid ee'$$

Contexts are (Γ, C), where Γ is a set of typing declarations of the form $(x : T)$ and C consists of *coercions* of the form $T \geq S$ where T and S are *atomic* types, that is either type constants or type variables.

The four main rules of inference of **M** are:

$$(var) \quad \frac{}{C, \Gamma, x : T \vdash x : T}$$

$$(\lambda) \quad \frac{C, x : T, \Gamma \vdash e : S}{C, \Gamma \vdash (\lambda x.e) : T \to S}$$

$$(app) \quad \frac{C, \Gamma \vdash e : T \to S \qquad C, \Gamma \vdash e' : T}{C, \Gamma \vdash ee' : S}$$

$$(coerce) \quad \frac{C, \Gamma \vdash e : T \qquad C \vdash T \geq S}{C, \Gamma \vdash e : S}$$

It also has two subsidiary rules which read:

$$(arrow) \quad \text{From } T_2 \geq T_1 \text{ and } S_1 \geq S_2 \text{ derive } T_1 \to S_1 \geq T_2 \to S_2$$

$$(trans) \quad \text{From } T_1 \geq T_2 \text{ and } T_2 \geq T_3 \text{ derive } T_1 \geq T_3$$

The main differences between **M** and **PP** are:

1. system **M** does not allow subtyping constraints which refer to arrow types;
2. in system **PP** we have to declare the type of the variable we are abstracting over, so the system is closer to the typed lambda-calculus;
3. **PP** does not allow type constants, which **M** does;
4. system **M** has no casting of expressions.

Mitchell's subtyping relation is stricter than ours, as he can prove the following lemma:

Lemma 2 *In system* **M**

$$C \vdash T_1 \to T_2 \geq S_1 \to S_2 \qquad \textit{iff} \qquad C \vdash S_1 \geq T_1 \quad \textit{and} \quad C \vdash T_2 \geq S_2$$

In the system **PP** we know that if $C \vdash S_1 \geq T_1$ and $C \vdash T_2 \geq S_2$ then $C \vdash T_1 \to T_2 \geq S_1 \to S_2$, using rule (*arrow*), but the converse is not necessarily true. If

$$C \vdash T_1 \to T_2 \geq S_1 \to S_2$$

that could mean that $C \vdash T_2 \geq S_2$ and $C \vdash S_1 \geq T_1$, but it can also be the case that there exists an X such that $C \vdash T_1 \to T_2 \geq X$ and $C \vdash X \geq S_1 \to S_2$. Note that it is the same problem discussed in section 2.2.4.

To give semantics to system **M** Mitchell starts with a model of the untyped lambda-calculus, thought of as a triple $(D, Fun, Graph)$. To quote his definition:

Definition 6 *A lambda model $(D, Fun, Graph)$ is a set D together with mappings $Fun: D \to [D \to D]$ and $Graph: [D \to D] \to D$ such that:*

1. $Fun \cdot Graph = 1_{[D \to D]}$
2. *all functions in D that are definable by lambda-expressions can be "compiled" into elements of D using $Graph$.*

In other words, take D a set which solves the following domain equation

$$D \cong [D \to D]$$

and where the isomorphisms are called Fun and $Graph$.

Adapting the ideas of *Type Inference and Coercion* to system **PP** we forget about type constants K and add one main rule of inference

$$\frac{C, \Gamma \vdash e : T}{C, \Gamma \vdash (e : T) : T}$$

and a subsidiary rule

$$\frac{}{C \vdash T \geq T} \quad (refl)$$

Note that the rule

$$\frac{}{C, T \geq S \vdash T \geq S} \quad (ass)$$

is implicit in **M**.

Types are interpreted as arbitrary sets of elements of lambda-models. A type environment η for a model $(D, Fun, Graph)$ is a mapping from type variables to subsets of D. The meaning of a type expression T in a type environment η is defined inductively by:

$$\begin{array}{lcl} [[V]]\eta & = & \eta(V) \\ [[T \to T']]\eta & = & \{d \mid \forall d_1 \in [[T]]\eta, Fun(d)(d_1) \in [[T']]\eta\} \end{array}$$

Coercions or subtyping constraints are interpreted as subset inclusion. Thus, say a model D and a type environment η satisfy a set of coercions $C = \{T_1 \geq S_1, \ldots, T_k \geq S_k\}$ if

$$[[T_i]]\eta \subseteq [[S_i]]\eta \qquad \text{for all} \qquad T_i \geq S_i \quad \text{in } C$$

Given a lambda model $(D, Fun, Graph)$ and an environment ρ mapping individual variables to elements of D, the meaning of an expression e of **PP** is defined inductively by:

$$\begin{array}{lcl} [[x]]\rho & = & \rho(x) \\ [[x : T \to e]]\rho & = & Graph(\lambda d.[[e]]\rho(d/x)) \\ [[e : T]]\rho & = & [[e]]\rho \\ [[e_1 e_2]]\rho & = & Fun([[e_1]]\rho)([[e_2]]\rho) \end{array}$$

A model D, a type environment η and an environment ρ satisfy type assigment Γ if whenever $(x : T)$ is in Γ, $\rho(x)$ is in $[[T]]\eta$.

A model D, a type environment η and an environment ρ satisfy a typing $(e{:}T)$ if $[[e]]\rho \in [[T]]\eta$.

Informally the statement

$$\Gamma, C \models_\lambda e{:}T$$

means that if types satisfy the subtyping constraints C and if variables have the types assigned by Γ then the expression e has type T.

Definition 7 *Say that the assertion*

$$\Gamma, C \models_\lambda e : T$$

is satisfied in the lambda model D iff for every environment ρ and every type environment η which satisfy C and Γ also satisfy $e : T$.

We have a semantic soundness result.

Theorem 2 *If $\Gamma, C \vdash e : T$ then $\Gamma, C \models_\lambda e : T$.*

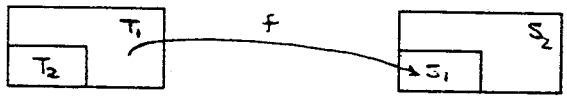
To prove it, we need an obvious lemma:

Lemma 3 *The subtyping subsidiary rules (refl), (arrow) and (trans) are sound in this model.*

Rule (*trans*) because of the transitivity of subset containment:

$$[[T_1]]\eta \subseteq [[T_2]]\eta \text{ and } [[T_2]]\eta \subseteq [[T_3]]\eta \text{ imply } [[T_1]]\eta \subseteq [[T_3]]\eta$$

Rule (*arrow*) because if you have a function $f: T_1 \to S_1$ in the picture below, f can be restricted to take elements of T_2 and if the image is in S_1, it is automatically in S_2.



$$[[T_2]]\eta \subseteq [[T_1]]\eta, [[S_1]]\eta \subseteq [[S_2]]\eta \Rightarrow [[T_1 \to S_1]]\eta \subseteq [[T_2 \to S_2]]\eta$$

Prove the theorem by induction on the proof $\Gamma, C \vdash e : T$. In the basis case the proof is an assumption

$$\frac{}{\Gamma, C \vdash x : T}$$

x is a type variable and $(x : T)$ is in Γ. The result is trivial, by definition of (η, ρ) satisfying C and Γ.

If the proof has more than one rule and x is a variable, then $(x : S)$ is in Γ and there is a derivation π consisting only of rules (*arrow*) and (*trans*) such that $C \vdash S \geq T$ and

$$\frac{\dfrac{}{\Gamma \vdash x : S} \qquad \dfrac{\overset{\pi}{}}{C \vdash S \geq T}}{\Gamma, C \vdash x : T}$$

By definition $\rho(x) \in [[S]]\eta$. Using lemma 2, $[[S]]\eta \subseteq [[T]]\eta$. Thus $\rho(x) \in [[T]]\eta$.

If the proof has more than one rule and x is not a variable, 3 possibilities:

- If the last rule applied was (λ), the expression is $(x : T \to e)$ and there is a derivation π of $\Gamma \cup \{x : T\}, C \vdash e : S$.

$$\frac{\dfrac{\overset{\pi}{}}{\Gamma \cup \{x : T\}, C \vdash e : S}}{\Gamma, C \vdash (x : T \to e) : T \to S} \ (\lambda)$$

  By induction hypothesis $\Gamma \cup \{x : T\}, C \models_\lambda e : S$, so for all (ρ, η) satisfying $\Gamma \cup \{x : T\}$ and C, we know $[[e]]\rho \in [[S]]\eta$. But all (ρ, η) in this case, satisfy (Γ, C) as well and by definition $[[x : T \to e]]\rho = Graph(\lambda d.[[e]]\rho(d/x))$. We want to show that $[[x : T \to e]]\rho \in [[T \to S]]\eta$, where

$$[[T \to S]]\eta = \{d \mid \forall d_1 \in [[T]]\eta \Rightarrow Fun(d)(d_1) \in [[S]]\eta\}$$

  Thus it is enough to show that $Fun(Graph(\lambda d.[[e]]\rho(d/x)))(d_1) \in [[S]]\eta$. As $Fun(Graph(f) = f$, we only need to show that $\lambda d.[[e]]\rho(d/x)(d_1) \in [[S]]\eta$ for all $d_1 \in [[T]]\eta$. But $[[e]]\rho \in [[S]]\eta$ by induction hypothesis.

- If the last rule applied was (*app*) then the expression is $e_1 e_2$ and there are derivations π_1 such that $\Gamma, C \vdash e_1 : T \to S$ and π_2 such that $\Gamma, C \vdash e_2 : T$.

$$\frac{\dfrac{\overset{\pi_1}{}}{\Gamma, C \vdash e_1 : T \to S} \qquad \dfrac{\overset{\pi_2}{}}{\Gamma, C \vdash e_2 : S}}{\Gamma, C \vdash e_1 e_2 : S} \ (app)$$

By induction we know $\Gamma, C \models_\lambda e_1 : T \to S$ and $\Gamma, C \models_\lambda e_2 : T$. Thus $[[e_1]]\rho \in [[T \to S]]\eta$ and $[[e_2]]\rho \in [[T]]\eta$. By definition

$$[[e_1 e_2]]\rho = Fun([[e_1]]\rho)([[e_2]]\rho)$$

hence $[[e_1 e_2]]\rho \in [[S]]\eta$.

- If the last rule applied was (*coerce*) then there is a derivation π_1 such that $\Gamma, C \vdash e : T$ and a derivation π_2 such that $C \vdash T \geq S$.

$$\frac{\dfrac{\pi_1}{\Gamma, C \vdash e : T} \qquad \dfrac{\pi_2}{\Gamma, C \vdash T \geq S}}{\Gamma, C \vdash e : S} \ (coerce)$$

By induction hypothesis $[[e]]\rho \in [[T]]]\eta$ and $[[T]]\eta \subseteq [[S]]\eta$, which implies $[[e]]\rho \in [[S]]\eta/$

- If the last rule applied was (*cast*) then there is a derivation π of $\Gamma, C \vdash e : T$.

$$\frac{\dfrac{\pi}{\Gamma, C \vdash e : T}}{\Gamma, C \vdash (e : T) : T} \ (cast)$$

By induction $[[e]]\rho \in [[T]]\eta$. By definition $[[(e : T) : T]]\rho = [[e : T]]\rho = [[e]]\rho$ and we have that $[[(e : T) : T]]\rho \in [[T]]\eta$.

The main results of Mitchell's paper are:

- the theorem which says that the type inference system is sound and complete with the added equality rule

$$\frac{\Gamma, C \vdash e : T \qquad e = f}{\Gamma, C \vdash e : S} \ (eq)$$

- a corollary that says the four first rules of M are complete for typing terms in normal form
- The four rules of M plus the equality rule above make a semantically complete set of rules, but an undecidable one.

The reason Mitchell gives for his system being incomplete does not apply to the system **PP**. In **PP** every subterm of a term of the calculus has a normal form. At least two questions here:

- Is the system **PP** complete as it is ?
- What happens to **PP** if one decides that subtyping constraints can only exist between atomic types, or less strictly, constraints do not relate a type variable with an arrow type containing it?

My guess is that if we could prove a lemma 1, the system **PP** would be complete. But at the moment it is just a conjecture.

Mitchell also presents a type-checking algorithm, with three subsidiary algorithms. One such algortithm computes coercion sets, and looks very much like *GE*. He also uses a notion of *normal well-typing* (and unification).

In a later paper Mitchell uses a generalisation of ideal models, because he's interested in interpreting $\forall V.V$ as a non-empty subset of D. Later on he uses PER models because of extensionality.

3.2 The Weak Ideal Model

Cardelli's paper describes an *extended* version of the typed lambda-calculus with subtyping, because he basically wants to discuss records and their use in object-oriented programming. If one crosses out from his system, constants, conditionals, records, variants and recursive data, as well as type constants, record types and variant types we have a system similar to **PP**.

We recall and adapt the ideas on *A Semantics of Multiple Inheritance*, to the system **PP**. In the restricted version of Cardelli's system he has as types:

$$TExp ::= K \mid T \to T'$$

as raw-terms:

$$Exp ::= x \mid x : T \to e' \mid e : T \mid e_1 e_2$$

and as rules of inference:

$$(var) \quad \frac{}{\Gamma, x : T \vdash x : S} \qquad \text{if } T \geq S$$

$$(\lambda) \quad \frac{x : T, \Gamma \vdash e : S}{\Gamma \vdash (\lambda x.e) : T \to S}$$

$$(app) \quad \frac{\Gamma \vdash e : T \to S \quad \Gamma \vdash e' : T}{\Gamma \vdash ee' : S}$$

$$(spec) \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash (e : T) : S} \qquad \text{where } T \geq S$$

The main differences between (the restriction of) Cardelli's system, call it **C**, and **PP** are that

- **C** has no type variables;
- **C** has no set C of subtyping assumptions explicitly in the formal system;
- in **C** rule (*sub*) is a derived rule;
- in **C** one proves that $\geq$ is a partial order, hence reflexive, transitive and antisymmetric; (mention casting?)
- **C** has meet and join types.

Cardelli's subtyping relation is also stricter than ours, as he has a definition:

$$T_1 \to T_2 \geq S_1 \to S_2 \qquad \text{iff} \qquad S_1 \geq T_1 \quad \text{and} \quad T_2 \geq S_2$$

The semantics Cardelli gives for **C** are based on the weak ideal model of MacQueen, Plotkin and Sethi in [MPS]. Adapting it to **PP** we have:

Let $\mathbf{V}$ be the recursively defined domain of values.

$$\mathbf{V} \cong (\mathbf{V} \to \mathbf{V}) + \mathbf{W}$$

where $\mathbf{V} \to \mathbf{V}$ is the continuous function space and $\mathbf{W} = \{\mathbf{w}\}$.

Let $\mathcal{I}(\mathbf{V})$ be the set of non-empty ideals of $\mathbf{V}$, ie left-closed subsets of $\mathbf{V}$, which are closed under least upper bounds of increasing sequences and do not contain *wrong*. To provide semantic interpretations we describe two semantic functions:

$$\mathcal{E} : Exp \to Env \to \mathbf{V}$$

$$\mathcal{T} : TExp \to TEnv \to \mathcal{I}(\mathbf{V})$$

where $Env = Var \to \mathbf{V}$ is the set of environments ranged over by ρ and $TEnv = TVar \to \mathcal{I}(\mathbf{V})$ is the set of type environments, ranged over by η. Both functions are given by structural induction.

Definition 8 *The semantic function* $\mathcal{T}: TExp \to TEnv \to \mathcal{I}(\mathbf{V})$ *is defined by:*

$$\begin{array}{lcl} \mathcal{T}[[V]]\eta & = & \eta(V) \\ \mathcal{T}[[T \to T']]\eta & = & \{f \in \mathbf{V} \to \mathbf{V} |\ v \in \mathcal{T}[[T]]\eta \Rightarrow f(v) \in \mathcal{T}[[T']]\eta\} \textit{ in } \mathbf{V}. \end{array}$$

The semantic function $\mathcal{E}: Exp \to Env \to \mathbf{V}$ *is defined by:*

$$\begin{array}{lcl} \mathcal{E}[[x]]\rho & = & \rho(x) \\ \mathcal{E}[[x : T \to e]]\rho & = & (\lambda v.\mathcal{E}[[e]]\rho\{v/x\}) \textit{ in } \mathbf{V} \\ \mathcal{E}[[e : T]]\rho & = & \mathcal{E}[[e]]\rho \\ \mathcal{E}[[e_1 e_2]]\rho & = & \textit{if } \mathcal{E}[[e_1]]\rho \in \mathbf{V} \to \mathbf{V} \textit{ then} \\ & & (\textit{if } \mathcal{E}[[e_2]]\rho \in \mathbf{W} \textit{ then wrong else } (\mathcal{E}[[e_1]]\rho|\mathbf{V} \to \mathbf{V})(\mathcal{E}[[e_2]]\rho)) \textit{ else wrong} \end{array}$$

Note that for all type expressions T, $\mathcal{T}[[T]]\eta$ is an ideal (hence $\perp \in \mathcal{T}[[T]]\eta$) and for all T, all η and for all $v \in \mathbf{V}$,

$$v \in \mathcal{T}[[T]]\eta \Rightarrow v \neq \textit{wrong}$$

The interpretation of subtyping constraints ($T \geq S$) is done through (set-theoretical) inclusion of ideals. Thus:

$$T \geq S \Leftrightarrow \mathcal{T}[[T]]\eta \subseteq \mathcal{T}[[S]]\eta$$

So the main difference between this semantics and the previous one is that here, because of recursion (which is not present yet!), we are taking as types restricted subsets (ideals) of the domain $\mathbf{V}$.

The same semantic soundness result we had for lambda models is obtained, if we define the relation $\models_I$. To define the relation $\models_I$, say that a type environment η *agrees with* C a set of subtyping constraints $\{T_1 \geq S_1, \ldots, T_k \geq S_k\}$ iff $\mathcal{T}[[T_i]]\eta \subseteq \mathcal{T}[[S_i]]\eta$ for all $i = 1, \ldots, k$.
Also an environment ρ *agrees with* Γ *and type environment* η iff for all assertions $(x : T)$ in Γ, $\rho[[x]] \in \mathcal{T}[[T]]\eta$.

Definition 9 *Given a context* (Γ, C) *and a judgement* $(e:T)$ *say* $\Gamma, C \models_I e : T$ *iff for every type environment* η *which agrees with* C*, and for every environment* ρ *that agrees with* Γ *and* η*, we have that* $\mathcal{E}[[e]]\rho$ *is in* $\mathcal{T}[[T]]\eta$.

Theorem 3 *If* $\Gamma, C \vdash e : T$ *then* $\Gamma, C \models_I e : T$.

Cardelli also mentions a type-checking algorithm for his calculus and proves soundness of this algorithm with respect to the formal inference system.

3.2 Modest Model

Bruce and Longo start their paper *A Modest Model of Records, Inheritance and Bounded Quantification* stating that ideal models are not sound because of the failure of weak extensionality (check BMM88). We describe a restricted form of their version of typed λ-calculus with subtyping.

Type expressions are like in **PP**, subtyping constraints are of the form $V \geq T$, where V is a type variable and T is any type expression. Expressions are like in **PP** except that their system, from now on **BL**, has no casting of expressions.

Rules of inference are similar to the ones in **PP**, namely:

$$(ass) \qquad C, V \geq T \vdash V \geq T$$

$$(ref) \qquad C \vdash T \geq T$$

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

$$(arrow) \qquad \frac{C \vdash T_2 \geq T_1 \qquad C \vdash S_1 \geq S_2}{C \vdash T_1 \to S_1 \geq T_2 \to S_2}$$

$$(var) \qquad \frac{}{C, \Gamma, x : T \vdash x : T}$$

$$(\lambda) \qquad \frac{C, x : T, \Gamma \vdash e : S}{C, \Gamma \vdash (x : T \to e) : T \to S}$$

$$(app) \qquad \frac{C, \Gamma \vdash e : T \to S \qquad C, \Gamma \vdash e' : T}{C, \Gamma \vdash ee' : S}$$

$$(sub) \qquad \frac{C, \Gamma \vdash e : T \qquad C, \Gamma \vdash T \geq S}{C, \Gamma \vdash e : S}$$

Thus without records system **BL** looks the same as **PP**, BUT they only accept subtyping constraints where the lhs is a type variable.

Following Bruce and Longo, we want to show a per model for **PP**.

Definition 10 *Let* $(\mathbf{N}, \cdot)$ *be Kleene's applicative structure. Say that* A *is a* per *on* $\mathbf{N}$ *iff* A *is a symmetric and transitive binary relation on* $\mathbf{N}$. *If* A *is a per call the set* $dom(A) = \{n | nAn\}$ *its domain.*

The category **PER** has as objects pers on $\mathbf{N}$. To define morphisms of pers we need another concept.

Definition 11 *Given a per* A *and for any* $n \in dom(A)$, *we write* $[n]_A$ *for the equivalence class of 'n' with respect to* A. *Then the quotient set of* $\mathbf{N}$ *with respect to* A, *denoted* $Q(A)$ *is the set* $\{[n]_A | n \in dom(A)\}$.

Now a morphism of pers from A to B is a map $f : Q(A) \to Q(B)$ such that

$$\exists n \forall p (pAp \Rightarrow f([n]_A) = [n.p]_B)$$

Recall the definition of function spaces or exponentials in **PER**. If A and B are per's, define $A \to_{per} B$ as the per such that:

$$\forall m, n \; m(A \to_{per} B)n \quad \textit{iff} \quad \forall p, q \; (pAq \Rightarrow m.pBn.q)$$

There are also categorical products of pers and **PER** is a cartesian closed category.

Definition 12 *Given pers B and C, we say $B \geq C$ iff $B \subseteq C$ as pairs of ordered pairs, so for all m and n if mBn then mCn.*

Proposition 1 *(Bruce and Longo) If $T_2 \geq T_1$ and $S_1 \geq S_2$ then $T_1 \to S_1 \geq T_2 \to S_2$.*

Three possibilities here:

- Following, for instance, Asperti, we might want to say types of **PP** are objects in **PER** and terms of **PP** are arrows in the category **PER**.
- Following Amadio'90 we might deal with type environments and environments.
- Follwing Mitchell'86 we can erase types to prove soundness.

In all cases, suppose a type environment η consists of a mapping from type variables into objects of the category **PER**.

Definition 13 *The semantic function $[[\ ]]: TExp \to TEnv \to \mathbf{PER}$ is defined by:*

$$\begin{array}{lcl} [[V]]\eta & = & \eta(V) \\ [[T \to T']]\eta & = & [[T]]\eta \to_{per} [[T']]\eta \end{array}$$

A type environment η satisfies a set of subtyping constraints C if

$$[[T_i]]\eta \subseteq [[S_i]]\eta \text{ for all } T_i \geq S_i \text{ in } C$$

First, following Asperti, interpret types as pers in **PER** as above. Then say a context

$$\Gamma = \{x_1 : T_1, \ldots, x_k : T_k\}$$

is interpreted as a product of its types, so

$$[[\ \Gamma\ ]]\eta = [[T_1]]\eta \times \ldots \times [[T_k]]\eta$$

Typed terms-in-context are to be interpreted as morphisms in **PER** from the interpretation of the context to the interpretation of its type:

$$[[\ \Gamma \vdash e : T\ ]]\eta = [[\Gamma]]\eta \xrightarrow{[[e]]} [[T]]\eta$$

Thus we have an inductive definition:

- If the rule applied is (var), e is a variable x and $(x : T)$ is in Γ,

$$[[\Gamma \vdash \{x : T\}]]\eta = \pi_i : [[T_1]]\eta \times \ldots \times [[T_k]]\eta \to [[T]]$$

- If the rule is (λ), e is a lambda-abstraction $(x : T \to e) : T \to S$ and if we know

$$[[e]]\eta : [[\Gamma \cup x : T]]\eta \xrightarrow{f} [[S]]\eta$$

then

$$[[(x : T \to e) : T \to S]]\eta = [[\Gamma]]\eta \xrightarrow{\overline{f}} ([[T]]\eta \to [[S]]\eta)$$

- If the rule is (app), the expression $e_1 e_2$ is an application where $e_1 : T \to S$ and $e_2 : T$ and if we know $[[e_1]]\eta : [[\Gamma]]\eta \xrightarrow{f} [[T \to S]]\eta$ and $[[e_2]]\eta : [[\Gamma]]\eta \xrightarrow{g} [[S]]\eta$ then

$$[[e_1 e_2]]\eta = eval \circ < f, g >$$

- If the rule was $(cast)$ and we know $[[\Gamma \vdash e : T]]\eta = f : [[\Gamma]]\eta \to [[T]]\eta$, then

$$[[\Gamma \vdash (e : T) : T]]\eta = id_{[[T]]\eta} \circ f : [[\ \Gamma\ ]]\eta \to [[T]]\eta$$

- If the rule was (*sub*) and we know $[[\Gamma \vdash e : T]]\eta = f : [[\Gamma]]\eta \to [[T]]\eta$ and $[[T]]\eta \subseteq [[S]]\eta$ then the new meaning is the function f restricted to $[[S]]\eta$,

$$[[\Gamma \vdash e : S]]\eta = f|_{[[S]]\eta}$$

We could extend this interpretation to say $[[\ \Gamma, C \vdash e : T]]\eta$ if η satisfies C and $[[\Gamma \vdash e : T]]\eta$ as above. But the problem with this definition is that it is based on the inference rules and as there may be several different proofs of a certain typing, we have to prove that different proofs of the same typing give the same meaning to this expression. That's a difficult coherence result. Bruce and Longo manage to avoid this problem by omitting the rule (*sub*) and using explicit coercions in the second half of their paper, cf. Amadio'90 p22.

Now, following Amadio, define an environment ρ as a mapping from individual variables into $\bigcup_{A \in \mathbf{PER}} Q(A)$.

For all A, B in **PER**, define $\cdot_{A,B}$ as the morphism $Q(A \to_{per} B) \times Q(A) \to Q(B)$ given by $[f]_{A \to B} \cdot_{A,B} [n]_A = [fn]_B$.

Say environment ρ agrees with type environment η, notation $(\rho \downarrow \eta)$ if for all x of type T $\rho(x) \in Q([[T]]\eta)$.

Given environments η and ρ such that $(\rho \downarrow \eta)$, the interpretation of well-typed expressions is inductively defined as follows:

Definition 14 *The semantic function* $[[\ ]] : Exp \times Tenv \times Env \to \bigcup_{A \in \mathbf{PER}} Q(A)$ *is defined by:*

$$\begin{array}{lcl}
[[x : T]]\eta\rho & = & \rho(x) \\
[[(x : T \to e) : T \to S]]\eta\rho & = & \{\phi \in \mathbf{N} \mid nAn \text{ implies } \phi n \in [[e]]\rho([n]A/x)\eta\} \\
 & & \text{where } A = [[T]]\eta \\
[[e_1 e_2]]\eta\rho & = & [[e_1]]\eta\rho \cdot_{A,B} [[e_2]]\eta\rho \\
 & & \text{where } e_1 : (T \to S), e_2 : T \text{ and } A = [[T]]\eta, B = [[S]]\eta \\
[[e : T]]\eta\rho & = & [[e]]\eta\rho
\end{array}$$

Say a typing $e : T$ is satisfied by a pair of environments (η, ρ),

$$\models_{(\rho,\eta)} e : T \qquad \text{iff} \qquad [[e]]\rho\eta \in Q([[T]]\eta)$$

Say a context (Γ, C) is satisfied by a pair of environments (η, ρ) $\quad \models_{(\eta,\rho)} C, \Gamma$ if

1. η satisfies C,
2. $(\rho \downarrow \eta)$ and
3. $\models_{(\rho,\eta)} x : T$ for all $(x : T)$ in Γ.

But with this defininition of '$\models_{(\rho,\eta)}$' if $e : T$ and $T \geq S$ we do not necessarily have $e : S$, as $T \subseteq S$ as pers does not imply $Q(T) \subseteq Q(S)$. Thus to have proposition:

Proposition 2 *If* $\Gamma, C \vdash e : T$ *then* $\Gamma, C \models_{\mathbf{P}} e : T$.

we have to define $\Gamma, C \models_{\mathbf{P}} e : T$ appropriately. Maybe we should mention other models, for instance complete pers or reducibility candidates. Amadio fits them nicely under the big umbrella of per models.

Some general questions:

- Should we try to compare algorithms? they're all provably sound.
- What would be a general categorical model for simply typed-lambda-calculus with subtyping? A ccc with an extra function spaces? Curien and Ghelli suggest a ccc with a distinguished collection of coercion arrows, closed under certain operations...
- Could Wand's method be used with type variables?

4 The Type Inference Mechanism

In the larger fragment ***Ponder_***, the type inference mechanism will deal with contexts having three parts,

- The component Γ, as before, consists of type declarations of the form $\{x_1 : T_1, \ldots, x_n : T_n\}$ where x_i are individual variables and T_i are type expressions.
- The component C consists of subtyping assumptions of the form $T_i \geq S_i$, where T_i, S_i are type expressions.
- The component $\mathcal{F}$ consists of a list of type variables which are *"fixed"* by the type inference mechanism. That means that the algorithm should prove the inferred type for the expression without using any extra hypotheses on V's that are fixed.

The type inference mechanism now consists of four main parts, the functions ***type-check***, *valid*, *GE* and *OR*.

The main function, as before called ***type-check*** takes two arguments:

- a context $(\mathcal{F}_0, C_0, \Gamma)$
- a Ponder expression '*e*'

and it returns either *FAIL* or a type expression T and another (part of) context $[\mathcal{F}, C]$ with the intended meaning that (C, Γ) 'proves' that '*e*' has type T.

Definition 15 *The function* ***type-check***$([\mathcal{F}, C], \Gamma, e)$ *is defined by cases on the structure of the Ponder expression 'e', as follows:*

var *if the expression is a variable 'x', its type is the one given in the* Γ *part of the context, that is:*

$$\textbf{type-check}([\mathcal{F}, C], \Gamma, x) = ([\mathcal{F}, C], T) \text{ if } (x : T) \text{ is in } \Gamma, \text{ FAIL otherwise.}$$

abs *if the expression is a* λ*-abstraction* $(x : T \rightarrow e)$*, its type is* $T \rightarrow S$*, provided the body 'e' has type* S *in the context augmented with* $\{x : T\}$*, so:*

$$\begin{aligned}&\textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, x : T \rightarrow e) = ([\mathcal{F}, C], T \rightarrow S)\\ &\quad \textit{if } \textbf{type-check}([\mathcal{F}_0, C_0], \Gamma \cup \{x : T\}, e) = ([\mathcal{F}, C], S)\end{aligned}$$

fall *if the expression is a for all quantification* $\Lambda V.e$*, its type is* $\forall V.S$*, provided 'e' can be type-checked to* S*, with the assumption that* V *is fixed:*

$$\begin{aligned}&\textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, \Lambda V.e) = ([\mathcal{F} \setminus \{V\}, C], \forall V.S)\\ &\quad \textit{if}\textbf{type-check}([\mathcal{F}_0 \cup \{V\}, C_0], \Gamma, e) = ([\mathcal{F}, C], S)\end{aligned}$$

cast *if the expression is a cast* $(e : T)$*, it has the casted type* T*, provided typechecking 'e' gives* S *and it is consistent to say that type* S *is more general than the cast* T*,* $S \geq T$*:*

$$\begin{aligned}&\textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, e : T) = (\textit{valid}[GE(S, T) \cup [\mathcal{F}, C]], T)\\ &\quad \textit{if}\textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, e) = ([\mathcal{F}, C], S)\end{aligned}$$

appl *if the expression is an application* $(e_1 e_2)$*, its type is the variable type* R*, provided that '*e_1*' can be type-checked to* T_1*, '*e_2*' can be type-checked to* T_2 *and these typings are consistent with* $T_1 \geq T_2 \rightarrow R$*:*

$$\begin{aligned}&\textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, e_1 e_2) = (\textit{valid}[[\mathcal{F}_1, C_1] \cup [\mathcal{F}_2, C_2] \cup GE(T_1, T_2 \rightarrow R)], R)\\ &\textit{if } \textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, e_1) = ([\mathcal{F}_1, C_1], T_1)\\ &\textit{and } \textbf{type-check}([\mathcal{F}_0, C_0], \Gamma, e_2) = ([\mathcal{F}_2, C_2], T_2)\\ &\textit{and } R \textit{ is a fresh type variable}\end{aligned}$$

Some observations before we define *valid*, GE and OR:

- The set $\mathcal{F}$ of fixed variables always start empty. Note that clause **fall** does not really increased it, since if the mechanism succeeds, the variable that was made fixed, is taken out the set $\mathcal{F}$. But as we shall see the function GE does add variables to this set. So $\mathcal{F}_0 \subseteq \mathcal{F}$.
- The set C of subtyping assumptions may start empty or not. But if clauses **cast** and **appl** are used it will increase in size, so $C_0 \subseteq C$.
- The convention is to write ***type-check*** $([\mathcal{F}_0, C_0, \Gamma) = (c, T)$ if C is not $FAIL$.Alternatively, we could write escape clauses, if $valid[\mathcal{F}, C \neq FAIL$ all over the definition.
- Two special conventions apply to contexts.
  - The Fairbairn convention says all bound variables have different names from the other variables in the context, to quote,

    > Whenever $\forall V.T$ occurs, it is assumed that V is distinct from all other variables encountered (in practice this is implemented by renaming).

  - The Γ-convention says type variables do not occur free in the initial context Γ, C fed into the type-inference mechanism.

For purposes of analysis, we think of the function *valid* as the composition of three functions,

- *clos* (for transitive closure under GE),
- *min* (for closure under minimal types) and
- *check*, which checks a set of assumptions C for consistency.

Definition 16 *The function clos takes a set $[\mathcal{F}, C]$ and returns clos $[\mathcal{F}, C]$, the least set such that:*

- $[\mathcal{F}, C] \subseteq clos\ [\mathcal{F}, C]$
- $\{T \geq V, V \geq S\} \subseteq clos\ [\mathcal{F}, C] \Rightarrow GE(T, S) \subseteq clos\ [\mathcal{F}, C]$

Definition 17 *The function min takes a set $[\mathcal{F}, C]$ and closes it under minimal types, so min $[\mathcal{F}, C]$ is the least set such that*

- $[\mathcal{F}, C] \subseteq min\ [\mathcal{F}, C]$
- $\{T \geq V, S \geq V\} \subseteq min\ [\mathcal{F}, C] \Rightarrow OR(T, S) \subseteq min\ [\mathcal{F}, C]$

The function *check*, as the name indicates, checks a set $[\mathcal{F}, C]$ for consistency.

Definition 18 *Given a set $[\mathcal{F}, C]$,*

$$check([\mathcal{F}, C]) = \begin{cases} FAIL & \text{if } \{fixed\ V, T \geq V\} \subseteq [\mathcal{F}, C] \\ FAIL & \text{if } \{fixed\ V, V \geq T\} \subseteq [\mathcal{F}, C] \\ [\mathcal{F}, C] & \text{otherwise} \end{cases}$$

Note that $check([\mathcal{F}, C])$ either returns $FAIL$ or the same set $[\mathcal{F}, C]$. (We will also consider the function $valid_r$, which is only the composition of *clos* and *check*.)

Now to define the function GE. remember that type expressions are either type variables, or arrow types or for all quantified type expressions.

Definition 19 *Given type expressions T and S the function $GE(T, S)$ returns a pair $[\mathcal{F}, C]$ where C is a set of subtyping assumptions and $\mathcal{F}$ is a set of fixed type variables. The function $GE(T, S)$ is defined by cases on the structure of T and S as follows:*

- *If T is a type variable V*

- *C1. If S is any type expression, $GE(V,S) = [\emptyset, \{V \geq S\}]$.*

- *If S is a type variable U,*
  - *C2. If T is any type expression, $GE(T,U) = [\emptyset, \{T \geq U\}]$.*

- *If T is an arrow type $T_1 \rightarrow T_2$*
  - *C3. If S is an arrow type $S_1 \rightarrow S_2$, $GE(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = GE(S_1,T_1) \cup GE(T_2,S_2)$, where union means union in both coordinates.*
  - *C4. If S is $\forall U.S'$, $GE(T_1 \rightarrow T_2, \forall U.S') = \{V\} \cup GE(T_1 \rightarrow T_2, S')$, where union means that the variable V is added to the set of fixed variables of $GE(T_1 \rightarrow T_2, S')$.*

- *If T is a for all quantification $\forall V.T'$*
  - *C5. If S is any type expression, $GE(\forall V.T', S) = GE(T', S)$.*

Examples Revisited

Example 1. Suppose I, B and V are type variables. Suppose also that f and x have been declared in Γ with types $(I \rightarrow I) \rightarrow B$ and $\forall V.(V \rightarrow V)$, respectively. (The difference from the example in section(?) is that x before was declared of type $V \rightarrow V$.)

If one applies the function ***type-check*** to try to infer a type for fx in the context where $C_0 = \emptyset$ and $\Gamma = \{f : (I \rightarrow I) \rightarrow B, x : \forall V.V \rightarrow V\}$:

$$\begin{array}{lcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fx) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, f) & = & (\emptyset, (I \rightarrow I) \rightarrow B) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, x) & = & (\emptyset, \forall V.V \rightarrow V) \end{array}$$

where

$$C = clos(GE((I \rightarrow I) \rightarrow B, (\forall V.V \rightarrow V) \rightarrow R))$$

The mechanism will call the GE with parameters $(I \rightarrow I) \rightarrow B$ and $(\forall V.V \rightarrow V) \rightarrow R$:

$$\begin{array}{lcl} GE((I \rightarrow I) \rightarrow B, (\forall V.V \rightarrow V) \rightarrow R) & = & GE(\forall V.V \rightarrow V, I \rightarrow I) \cup GE(B,R) \\ & = & GE(V \rightarrow V, I \rightarrow I) \cup GE(B,R) \\ & = & GE(I,V) \cup GE(V,I) \cup GE(B,R) \\ & = & [\emptyset, \{I \geq V, V \geq I, B \geq R\}] \end{array}$$

As before, the intuition is that the pair $[\emptyset, C]$ returned by GE, where

$$C = \{I \geq V, V \geq I, B \geq R, I \geq I, V \geq V\}$$

'proves' $(I \rightarrow I) \rightarrow B \geq (\forall V.V \rightarrow V) \rightarrow R$. It's easy to describe a possible derivation in the system ***Ponder_***.

$$\dfrac{\dfrac{}{\emptyset \vdash \forall V.V \rightarrow V \geq I \rightarrow I}\ (inst) \qquad \dfrac{}{C \vdash B \geq R}\ (ass)}{C \vdash (I \rightarrow I) \rightarrow B \geq (\forall V.V \rightarrow V) \rightarrow R}\ (arrow)$$

But there are many others. Another derivation, closer to the algorithm would be:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \forall V.V \rightarrow V \geq V \rightarrow V} \qquad \dfrac{C \vdash I \geq V \; C \vdash V \geq I}{C \vdash V \rightarrow V \geq I \rightarrow I}}{C \vdash \forall V.V \rightarrow V \geq I \rightarrow I} \qquad \dfrac{}{C \vdash B \geq R}}{C \vdash (I \rightarrow I) \rightarrow B \geq (\forall V.V \rightarrow V) \rightarrow R}$$

Note that the first derivation above only uses one of the assumptions in C. Now to obtain one of the possible derivations of $fx : R$ we simply plugg either of the derivations above, call it π, in the following:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : (I \to I) \to B} \quad \dfrac{\pi}{C \vdash (I \to I) \to B \geq (\forall V.V \to V) \to R}}{\Gamma, C \vdash f : (\forall V.V \to V) \to R} \quad \dfrac{}{\Gamma \vdash x : \forall V.V \to V}}{\Gamma, C \vdash fx : R}$$

The main role of 'fixing' variables is to prevent derivations which could otherwise 'go through'. A variation on the example above might help to explain it.

Example 2. If the declaration part of the context Γ said instead $\{f : (\forall V.V \to V) \to B, x : I \to I\}$ and if we try to infer a type for fx,

$$\begin{array}{lcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fx) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, f) & = & (\emptyset, (\forall V.V \to V) \to B) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, x) & = & (\emptyset, I \to I) \end{array}$$

where

$$C = clos(GE((\forall V.V \to V) \to B, (I \to I) \to R))$$

The mechanism will call GE with parameters $(\forall V.V \to V) \to B$ and $(I \to I) \to R$:

$$\begin{array}{lcl} GE((\forall V.V \to V) \to B, (I \to I) \to R) & = & GE(I \to I, \forall V.V \to V) \cup GE(B, R) \\ & = & \{V\} \cup GE(I \to I, V \to V) \cup GE(B, R) \\ & = & [V\ fix, \{V \geq I, I \geq V, B \geq R\}] \end{array}$$

Intuitively it's easy to see that $(I \to I)$ cannot be more general than $\forall V.(V \to V)$, which is what the firsl line is trying to prove. In the algorithm V in $\mathcal{F}$ means that V should be any variable, so it should not have constraints imposed over it.

If we call the set $\{V \geq I, I \geq V, B \geq R\} = C$, the pair $[V, C]$ will be 'inconsistent', and $valid([V, C]) = FAIL$.

Example 3. Suppose T, A and B denote type variables, a, b and f individual variables, the initial subtyping part of a context C_o is empty and

$$\Gamma = \{f : \forall T.T \to (T \to T), a : A, b : B\}$$

To infer a type for fab in the context above, we apply $\textbf{\textit{type-check}}(\Gamma, \emptyset, fab)$. If it succeeds, we have:

$$\begin{array}{lcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fab) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, fa) & = & (C_1, R_1) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, b) & = & (C_2, R_2) \end{array}$$

where:

- R is a new type variable
- $C = clos[C_1 \cup C_2 \cup GE(R_1, R_2 \to R)]$
- (C_1, R_1) is the result of $\textbf{\textit{type-check}}(\Gamma, \emptyset, fa)$ and
- (C_2, R_2) is the result of $\textbf{\textit{type-check}}(\Gamma, \emptyset, b)$.

But $\textit{type-check}(\Gamma, \emptyset, b) = (\emptyset, B)$ and $\textit{type-check}(\Gamma, \emptyset, fa) = (C_1, R_1)$ where R_1 is a type variable and we have to calculate $C_1 = \textit{clos}[GE(\forall T.T \to (T \to T), A \to R_1)]$.

$$\begin{aligned} GE(\forall T.T \to (T \to T), A \to R_1) &= GE(A, T) \cup GE(T \to T, R_1) \\ &= \{A \geq T, T \to T \geq R_1\} \end{aligned}$$

Thus $C_1 = \{A \geq T, T \to T \geq R_1\}$, as *clos* does nothing here. Note that, as before, $C_1 \vdash \forall T.T \to (T \to T) \geq A \to R_1$ and $\Gamma, C_1 \vdash fa : R_1$. One possible derivation would be:

$$\dfrac{\dfrac{}{\Gamma \vdash f : T \to (T \to T)} \qquad \dfrac{\dfrac{}{C_1 \vdash A \geq T} \qquad \dfrac{}{C_1 \vdash T \to T \geq R_1}}{C_1 \vdash T \to (T \to T) \geq A \to R_1}\,(arrow)}{\dfrac{\Gamma, C_1 \vdash f : A \to R_1 \qquad \dfrac{}{\Gamma \vdash a : A}}{\Gamma, C_1 \vdash fa : R_1}}$$

Back to the main derivation, $C = \textit{clos}[C_1 \cup C_2 \cup C_3]$, where $C_2 = \emptyset$ and $C_3 = \{R_1 \geq B \to R\}$. Thus $C_1 \cup C_2 \cup C_3 = \{A \geq T, T \to T \geq R_1, R_1 \geq B \to R\}$ and, this time *clos* does something,

$$C = \{A \geq T, T \to T \geq R_1, R_1 \geq B \to R, T \to T \geq B \to R, B \geq T, T \geq R\}$$

One derivation of $\Gamma, C \vdash fab : R$ can be obtained as:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, C_1 \vdash fa : R_1} \qquad \dfrac{}{C_3 \vdash R_1 \geq B \to R}}{\Gamma, C_1, C_3 \vdash fa : B \to R} \qquad \dfrac{}{\Gamma \vdash b : B}}{\Gamma, C_1, C_2 \vdash fab : R}$$

Note that the function $GE(T, S)$ always terminates in $\textbf{\textit{Ponder\_}}$. It returns $\mathcal{F}$ a set of fixed variables and a set of assumptions C, where each one of the assumptions is of the form $V \geq S$ or $S \geq V$, where V is a type variable.

The type inference system seems sound, if not complete, and naively, at least the main tool to prove the soundness of the type-checker should be the following lemma. In analogy with section 2.3 it says that given two type expressions T and S, the function $GE(T, S)$ returns a pair $[\mathcal{F}, C]$ that either 'proves' $T \geq S$ or is inconsistent, in the sense that $\textit{valid}([\mathcal{F}, C]) = FAIL$. But that doesn't work, due to side conditions.

Conjecture 1 *Given two type expressions T and S, the function $GE(T, S)$ returns a pair $[\mathcal{F}, C]$ such that if $\textit{valid}([\mathcal{F}, C]) = [\mathcal{F}, C]$, then $C \vdash T \geq S$.*

Proof(???): By structural induction. As before, T and S can be either variables, or arrow types or for all quantified types. We check each one of the possibilities. Note that we run into problems in cases 4 and 6.

- if T is a type variable V
  - case 1: If S is any type expression, by definition, $GE(V, S) = [\emptyset, \{V \geq S\}]$. The singleton set $\{V \geq S\}$ clearly proves $V \geq S$, using rule assumption (*ass*).
- If S is a variable type U,
  - case 2: If T is any type expression, the result is as trivial as case 1, since by definition $GE(T, U) = [\emptyset, \{T \geq U\}]$.
- if T is an arrow type $T_1 \to T_2$

- case 3: If S is an arrow type $S_1 \to S_2$ then $GE(T_1 \to T_2, S_1 \to S_2) = GE(S_1, T_1) \cup GE(T_2, S_2)$. By induction hypothesis there exists $[\mathcal{F}_1, C_1]$, $valid([\mathcal{F}_1, C_1]) = [\mathcal{F}_1, C_1]$ and a derivation π_1 such that $C_1 \vdash S_1 \geq T_1$. Also there exists $[\mathcal{F}_2, C_2]$, $valid([\mathcal{F}_2, C_2]) = [\mathcal{F}_2, C_2]$ and a derivation π_2 such that $C_2 \vdash T_2 \geq S_2$. Then if $valid([\mathcal{F}_1, \mathcal{F}_2, C_1, C_2]) = [\mathcal{F}_1, \mathcal{F}_2, C_1, C_2]$, rule *(arrow)* (plus weakening) then guarantees that $C_1, C_2 \vdash T_1 \to T_2 \geq S_1 \to S_2$.

$$\frac{\dfrac{}{C_1 \vdash S_1 \geq T_1}\,\pi_1 \qquad \dfrac{}{C_2 \vdash T_2 \geq S_2}\,\pi_2}{C_1, C_2 \vdash T_1 \to T_2 \geq S_1 \to S_2}\ (arrow)$$

- case 4: If S is $\forall U.S'$, then $GE(T_1 \to T_2, \forall U.S') = \{U\} \cup GE(T_1 \to T_2, S')$. Suppose $GE(T_1 \to T_2, S') = [\mathcal{F}, C]$. By induction hypothesis, if $valid([\mathcal{F}, C] = [\mathcal{F}, C]$ then there exists derivation π such that $C \vdash T_1 \to T_2 \geq S'$.
  By FC, the bounded variable U is different from all the other type variables, so $U \notin ftv(T_1 \to T_2)$.
  Thus, if U is not free in C and if $valid[\mathcal{F} \cup U, C] = [\mathcal{F} \cup U, C]$ then we can use rule *(gen)* to conclude $C \vdash T_1 \to T_2 \geq \forall U.S'$, but that is not always the case....

$$\frac{\dfrac{}{C \vdash T_1 \to T_2 \geq S'}\,\pi}{C \vdash T_1 \to T_2 \geq \forall U.S'} \qquad U \notin ftv(C), U \notin ftv(T_1 \to T_2)$$

- if T is a for all quantification $\forall V.T'$
  - case 5: If S is either a variable or an arrow type, since $GE(\forall V.T', S) = GE(T', S)$, by induction hypothesis there exist $[\mathcal{F}, C] = valid[\mathcal{F}, C]$ and derivation π such that $C \vdash T' \geq S$ and by $(inst)$, $C \vdash \forall V.T' \geq T'$. Using transitivity we get $C \vdash \forall V.T' \geq S$.

$$\frac{\dfrac{}{\emptyset \vdash \forall V.T' \geq T'}\,(inst) \qquad \dfrac{}{C \vdash T' \geq S'}\,\pi}{C \vdash \forall V.T' \geq S}\ (trans)$$

  - case 6: If S is a for all quantification $\forall U.S'$, $GE(\forall V.T', \forall U.S') = GE(T', \forall U.S') = \{U\} \cup GE(T', S')$. By induction hypothesis there exist $[\mathcal{F}, C] = valid[\mathcal{F}, C]$ and derivation π such that $C \vdash T' \geq S'$. As before $U \notin ftv(T')$ by FC.
    If $U \notin ftv(C)$ and if $[\mathcal{F} \cup U, C] = valid[\mathcal{F}, C]$ we can use (gen), to get $C \vdash T' \geq \forall U.S'$. By instantiation $\emptyset \vdash \forall V.T' \geq T'$ and transitivity gives us $C \vdash \forall V.T' \geq \forall U.S'$.
    But there's a big if in this clause as well.

$$\frac{\dfrac{}{\emptyset \vdash \forall V.T' \geq T'}\,(inst) \qquad \dfrac{\dfrac{}{C \vdash T' \geq S'}\,\pi}{C \vdash T' \geq \forall U.S'}\,(gen)}{C \vdash \forall V.T' \geq \forall U.S'}\ (trans)$$

Now even if we had the lemma we still would have problems to prove the soundness of the type inference mechanism with respect to the typing rules of section 1. It is clear from the definition of the function ***type-check*** that the functions GE and ***valid***, hence lemma 1, are needed in the cases of e a casted expression or e an application.

Conjecture 2 *Given a context* $(\mathcal{F}_0, C_0, \Gamma)$ *and an expression in Ponder 'e' such that*

$$\textbf{\textit{type-check}}(\emptyset, C_0, \Gamma, e) = ([\mathcal{F}, C], T)$$

there exists a derivation π *according to the rules such that* π *has as its last formula* $C, \Gamma \vdash e : T$

Proof(???): By induction on the structure of the expression e. The basis corresponds to e being a variable.

BASIS: If e is a variable, $\textbf{\textit{type-check}}(\mathcal{F}_0, C_0, \Gamma, x) = ([\mathcal{F}, C], T)$, implies, by definition of the function type-check that the assertion $(x : T)$ is in the context Γ, $C = C_0$ and $\mathcal{F}_0 = \mathcal{F}$.

Now for the induction step, we have 4 cases:

- If e is a λ-abstraction $(x : T \rightarrow e)$, and if $\textbf{\textit{type-check}}(C_0, \Gamma, x : T \rightarrow e) = ([\mathcal{F}, C], T \rightarrow S)$ then $\textbf{\textit{type-check}}\ (C_0, \Gamma \cup \{x : T\}, e) = ([\mathcal{F}, C], T_e)$.
  By induction hypothesis there exists a derivation π whose last formulae is $\Gamma, x : T, C, \vdash e : T_e$ and we can use rule (λ) to get the derivation with last formula $\Gamma, C \vdash (x : T \rightarrow e) : T \rightarrow S$.

$$\dfrac{\dfrac{\pi}{\Gamma, x : T, C \vdash e : S}}{\Gamma, C \vdash (x : T \rightarrow e) : T \rightarrow S}\ (\lambda)$$

- If e is a casted expression $(e : T)$ and if $\textbf{\textit{type-check}}(C_0, \Gamma, (e : T)) = (C, T)$ then $\textbf{\textit{type-check}}(C_0, \Gamma, e) = (C_1, S)$ and $C = \textit{valid}(C_1 \cup GE(S, T))$. By induction there exists a derivation whose last formula is $C_1, \Gamma \vdash e : S$.
  By conjecture 1 (?), $GE(S, T)$ produces a set of assumptions C_2 such that $C_2 \vdash S \geq T$ and using weakening and rule (sub) we can derive $\Gamma, C_1, C_2 \vdash e : T$.

$$\dfrac{\dfrac{\pi}{C_1, \Gamma \vdash e : S} \qquad \dfrac{}{C_2 \vdash S \geq T}}{C_1, C_2, \Gamma \vdash e : T}$$

- If e is an application $e_1 e_2$ and $\textit{type} - \textit{check}(C_0, \Gamma, e_1 e_2) = (C, R)$ then:
  * $\textbf{\textit{type-check}}(C_0, \Gamma, e_1) = (C_1, T_1)$,
  * $\textbf{\textit{type-check}}(C_0, \Gamma, e_2) = (C_2, T_2)$,
  * $C = \textit{valid}(C_1 \cup C_2 \cup GE(T_1, T_2 \rightarrow R))$ and
  * R is a fresh type variable .

  By induction there is a derivation π_1 such that its last formula is $C_1, \Gamma \vdash e_1 : T_1$. Also there is a derivation π_2 which proves $C_2, \Gamma \vdash e_2 : T_2$. By conjecture (?) $GE(T_1, T_2 \rightarrow R)$ produces a set C_3 such that $C_3 \vdash T_1 \geq T_2 \rightarrow R$, via say π_3 and using rules (sub) and (app) we have $C, \Gamma \vdash e_1 e_2 : R$.

$$\dfrac{\dfrac{\dfrac{\pi_3}{C_3 \vdash T_1 \geq T_2 \rightarrow R} \qquad \dfrac{\pi_1}{C_1, \Gamma \vdash e_1 : T_1}}{C_1, C_2, \Gamma \vdash e_1 : T_2 \rightarrow R} \qquad \dfrac{\pi_2}{C_3, \Gamma \vdash e_2 : T_2}}{C_1, C_2, C_3, \Gamma \vdash e_1 e_2 : R}\ (app)$$

- If e is an expression $\Lambda V.e$ and $\textbf{\textit{type-check}}(C_0, \Gamma, \Lambda V.e) = (C \setminus \{\textit{fixed}\ V\}, \forall V.S)$ then $\textbf{\textit{type-check}}(C_0, \Gamma, e) = (C \cup \{\textit{fixed}\ V\}, S)$. By induction there exists a derivation π whose last formula is $\Gamma, C \cup \{\textit{fixed}\ V\} \vdash e : S$.
  IF V is not free in C then the rule for for all quantification applies.

Thus, even if we had the lemma, we'd still have a problem.

But the problem with the lemma is a serious one, in the sense that, the way the algorithm is defined, it is possible to accept an expression e, given a contex Γ, producing type expression T and subtyping constraints C, such that there is NO proof in the formal system of

$$\Gamma, C \vdash e : T$$

Some considerations are in order:

- the examples of failure of the mechanism I have at the moment do not satisfy the second convention on contexts mentioned just before the definition of *valid*. That is they do have free type variables in Γ. So, in a sense, they are not practically-minded as programmers would not declare free type variables.
- one of the points about the problematic examples in the next section is that GE returns a set of fixed variables F and apart from using F to fail a set, when that's the case, F is not used anymore, so we are throwing valuable information away.
- lack of soundness is a much more serious problem as far as type inference mechanis are concerned than lack of completeness. But it is no use at all to have a sound system which does almost nothing.

We will discuss some examples and some possibilities of fixing the type inference mechanism in the next section.

5 Problematic Examples

The reader may have noticed that we skimmed over the role of fixing variables. ***Fixing variables*** is an implementational trick. The fixing of variables should stop undesirable derivations of constraints of the form $V \to V \geq \forall T.T \to T$ and also make sure that when dealing with type abstraction over terms, one is really allowed to abstract.

But to fulfill its dual role, it seems to me that the algorithm has to be changed. We start with an easy example.

Example 4 Suppose V, U are type variables. Suppose also that Γ consists of a single declaration assumption $\{x : \forall V.(V \to V)\}$ and that C_0 is the empty set. If we want to ***type-check*** the cast expression $(x : \forall U.(U \to U))$ in the context (Γ, C_0) we have:

$$\begin{array}{rcll} \textbf{\textit{type-check}}(\Gamma, \emptyset, (x : \forall U.(U \to U)) & = & (C, \forall U.(U \to U)) & \text{if} \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, x) & = & (C_1, S) & \end{array}$$

where $C = \mathit{valid}(GE(S, \forall U.(U \to U)) \cup C_1)$ and S is the type the mechanism assigns to x.

It is clear that $\textbf{\textit{type-check}}(\Gamma, \emptyset, x) = (\emptyset, \forall V.(V \to V))$. And it is obvious that the type-checker should accept the type $\forall U.(U \to U)$ as an alpha-variant of the type declared, but the way functions were defined

$$GE(\forall V.(V \to V), \forall U.(U \to U)) = \{U \; \mathit{fix}, U \geq V, V \geq U\}$$

and *valid* fails this set.

Note that this example shows lack of completeness, not unsoundness!

One of the several possible ways of fixing it, is to say that when the $\forall$ quantification of V gets stripped off, V is marked as not fixed, or general. So when we come to the comparison between the variables we could say that comparing a variable which is fixed with one which is general, shouldn't fail the set.

That means a change in function GE as well as in *check*, but they are not serious changes.

Definition 20 *Given type expressions T and S the function $GE(T,S)$ returns a pair $(\mathcal{F},\mathcal{G},C)$ where C is a set of subtyping assumptions, $\mathcal{F}$ is a set of fixed type variables and $\mathcal{G}$ is a set of general variables. The function $GE(T,S)$ is defined by cases on the structure of T and S as follows:*

- *If T is a type variable V*
  - *C1. If S is any type expression, $GE(V,S) = [\emptyset, \{V \geq S\}]$.*
- *If S is a type variable U,*
  - *C2. If T is any type expression, $GE(T,U) = [\emptyset, \{T \geq U\}]$.*
- *If T is an arrow type $T_1 \rightarrow T_2$*
  - *C3. If S is an arrow type $S_1 \rightarrow S_2$, $GE(T_1 \rightarrow T_2, S_1 \rightarrow S_2) = GE(S_1,T_1) \cup GE(T_2,S_2)$, where union means union in all coordinates.*
  - *C4. If S is $\forall U.S'$, $GE(T_1 \rightarrow T_2, \forall U.S') = \{U\ fix\} \cup GE(T_1 \rightarrow T_2, S')$, where union means that the variable V is added to the set of fixed variables of $GE(T_1 \rightarrow T_2, S')$.*
- *If T is a for all quantification $\forall V.T'$*
  - *C5. If S is any type expression, $GE(\forall V.T', S) = \{V\ gen\} \cup GE(T', S)$ where union means that the variable V is added to the set of general variables of $GE(T', S)$.*

Definition 21 *Given a set $[\mathcal{F},\mathcal{G},C]$,*

$$check_1([\mathcal{F},\mathcal{G},C]) = \begin{cases} FAIL & \text{if } \{V\ fix, T \geq V\} \subseteq [\mathcal{F},\mathcal{G},C] \text{ and } T \text{ not a general variable} \\ FAIL & \text{if } \{V\ fix, V \geq T\} \subseteq [\mathcal{F},\mathcal{G},C] \text{ and } T \text{ not a genral variable} \\ [\mathcal{F},\mathcal{G},C] & \text{otherwise} \end{cases}$$

But problems can be a lot more serious, as the example below shows:
Example 5. Suppose U, V, S, T_1 and T_2 denote type variables. Suppose also that f and x have been declared with types $\forall U(S \rightarrow (U \rightarrow U)) \rightarrow V$ and $T_1 \rightarrow T_2$, respectively, in Γ and C_0 is empty.

If we apply the function ***type-check*** to try to infer a type for fx in the context (Γ, C_0) we have:

$$\begin{array}{rcc} \textbf{\textit{type-check}}(\Gamma,\emptyset,fx) & = & (C,R) \\ \textbf{\textit{type-check}}(\Gamma,\emptyset,f) & = & (\emptyset, \forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V) \\ \textbf{\textit{type-check}}(\Gamma,\emptyset,x) & = & (\emptyset, T_1 \rightarrow T_2) \end{array}$$

where

$$C = valid(GE(\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V, (T_1 \rightarrow T2) \rightarrow R))$$

This calls the function GE with parameters $\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V$ and $(T_1 \rightarrow T_2) \rightarrow R$:

$$\begin{array}{rcl} GE(\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V, (T_1 \rightarrow T_2) \rightarrow R) & = & GE(T_1 \rightarrow T_2, \forall U.(S \rightarrow (U \rightarrow U)) \cup GE(V,R) \\ & = & GE(T_1 \rightarrow T_2, S \rightarrow (U \rightarrow U)) \cup GE(V,R) \cup \{U\ fix\} \\ & = & GE(T_2, U \rightarrow U) \cup GE(S,T_1) \cup GE(V,R) \cup \{U\ fix\} \\ & = & \{S \geq T_1, T_2 \geq (U \rightarrow U), V \geq R\} \cup \{U\ fix\} \end{array}$$

Then $valid(\{S \geq T_1, T_2 \geq U \rightarrow U, V \geq R\} \cup \{U\ fix\})$ is the same set,

$$C = \{S \geq T_1, T_2 \geq U \rightarrow U, V \geq R\}$$

and the intuition should be that the C 'proves' $\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V \geq (T_1 \rightarrow T_2) \rightarrow R$
But there is NO possible derivation

$$C \vdash \forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V \geq (T_1 \rightarrow T_2) \rightarrow R$$

in the system **Ponder_**!!!

Notice that we could make the algorithm sound by changing function *check* so that, whenever V was fixed and V appeared *free* in one of the subtyping constraints (not against a general variable), the set failed. But as we mentioned before, the system would be very poor with this definition of *check*. Moreover, it would make no sense to have the example above $FAIL$ and the one below all right.

Example 6 Suppose now that f and x have been declared with types $(S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V$ and $T_1 \rightarrow T_2$, respectively, in Γ and C_0 is empty.

If we apply the function ***type-check*** to try to infer a type for fx in the context (Γ, C_0) we have:

$$\begin{array}{rcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fx) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, f) & = & (\emptyset, (S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, x) & = & (\emptyset, T_1 \rightarrow T_2) \end{array}$$

where

$$C = \mathit{valid}(GE((S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V, (T_1 \rightarrow T2) \rightarrow R))$$

The function GE will be called with parameters $(S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V$ and $(T_1 \rightarrow T_2) \rightarrow R$:

$$\begin{array}{rcl} GE((S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V, (T_1 \rightarrow T_2) \rightarrow R) & = & GE(T_1 \rightarrow T_2, (S \rightarrow \forall U.(U \rightarrow U)) \cup GE(V, R) \\ & = & GE(T_2, \forall U.U \rightarrow U) \cup GE(S, T_1) \cup GE(V, R) \cup \\ & = & \{S \geq T_1, T_2 \geq \forall U.U \rightarrow U, V \geq R\} \end{array}$$

But $(S \rightarrow \forall U.(U \rightarrow U)) \rightarrow V$ and $\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V$ are equivalent and the system **Ponder_** proves that. Thus it makes no sense to have example 5 rejected and 6 accepted. Note also that the set above is what the mechanism should have returned in the example 5, as it indeed proves what we want.

Another observation is that had the context in example 5 been closer to programmers' intuition, as for instance, it is in the example below, it'd have been posssible to give a derivation. One question is whether, if one uses only contexts satisfying the two conventions mentioned, is the type inference mechanism sound.

Example 7 Suppose now that f and x have been declared with types $\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V$ and $\forall T_1, T_2.(T_1 \rightarrow T_2)$, respectively, in Γ and C_0 is empty.

If we apply the function ***type-check*** to try to infer a type for fx in the context (Γ, C_0) we have:

$$\begin{array}{rcl} \textbf{\textit{type-check}}(\Gamma, \emptyset, fx) & = & (C, R) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, f) & = & (\emptyset, \forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V) \\ \textbf{\textit{type-check}}(\Gamma, \emptyset, x) & = & (\emptyset, \forall T_1, T_2.(T_1 \rightarrow T_2)) \end{array}$$

where

$$C = \mathit{valid}(GE(\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V, \forall T_1, T_2.(T_1 \rightarrow T_2) \rightarrow R))$$

Thus the mechanism will call the function GE with parameters $\forall U.(S \rightarrow (U \rightarrow U)) \rightarrow V$ and $\forall T_1, T_2.(T_1 \rightarrow T_2) \rightarrow R$:

$$\begin{array}{ll} GE(\forall U.S \rightarrow (U \rightarrow U) \rightarrow V, \forall T_1, T_2.T_1 \rightarrow T_2 \rightarrow R) & = \\ GE(\forall T_1, T_2.T_1 \rightarrow T_2, \forall U.S \rightarrow (U \rightarrow U) \cup GE(V, R) & = \\ GE(T_1 \rightarrow T_2, \forall U.S \rightarrow (U \rightarrow U) \cup GE(V, R) \cup \{T_1, T_2\ \mathit{gen}\} & = \\ GE(T_2, U \rightarrow U) \cup GE(S, T_1) \cup GE(V, R) \cup \{T_1, T_2\ \mathit{gen}\} \cup \{U\ \mathit{fix}\} & = \\ \{S \geq T_1, T_2 \geq U \rightarrow U, V \geq R\} \cup \{T_1, T_2\ \mathit{gen}\} \cup \{U\ \mathit{fix}\} & \end{array}$$

6 Conclusions and further work

After looking at examples 5 and 6 one could think that to attain soundness of the type inference mechanism, it would be enough to "normalise" types, by that I mean, vaguely, to push into the expresssion as far as possible the quantifiers. But that does not gives us a proof of soundness and there is some evidence that it is not desirable in all cases. It seems that a serious study of the proofs in the calculus **Ponder** needs to be undertaken.

I would like to mention that both ideal models and Per models should be able to cope with modelling **Ponder**$_-$. But there is no point in doing it unless we can get a sound type inference mechanism for **Ponder**$_-$.

Acknowledgments I would like to thank Martin Hyland, Jon Fairbairn, Andy Pitts, Robin Milner, Mike Gordon, Andy Gordon and Randy Pollack for several discussions on the subject of this report. Especially I would like to thank Eugenio Moggi for a careful reading of the manuscript. Some of his suggestions and Martin Hyland's ideas have been stored away for work in preparation.

References

[AMA] R. Amadio *Typed Equivalence, Type Assignment and Type Containment* to appear in Proc. CTRS'90, March '90.

[ASP] A. Asperti *Categorical Topics in Computer Science* Technical Report TD- 7/90 from the University of Pisa.

[BCGS] V. Breazu-Tannen, T. Coquand, C. Gunter and A. Scedrov *Inheritance and Explicit Coercion* LICS'89

[B&L] K. Bruce and G. Longo *A modest model of records, inheritance and bounded quantification* LICS'88

[Card] L. CARDELLI *A Semantics of Multiple Inheritance*, Information and Computation,**76** 138-164, 1988.

[C&W] L. CARDELLI and P. WEGNER *On Undertanding Types, Data Abstraction and Polymorphism* Comp. Surveys, 1985

[C&G] P.L. Curien and G. Ghelli *Coherence of Subsumption* Technical Report TD- 34/89 from the University of Pisa.

[D&M] L. Damas and R. Milner *Principal type schemes for functional programs* POPL'82

[Fair] J. FAIRBAIRN *Design and Impletementation of a Simple Typed Language based on the Lambda-Calculus*, Technical Report 75, Computer Laboratory, University of Cambridge, May 1985.

[Fair] J. FAIRBAIRN *Ponder and its Type System* Tech. Report 31 Computer Laboratory University of Cambridge, Nov'82.

[Fair] J. FAIRBAIRN *A New Type-Checker for a Functional Language* Tech. Report 53 Computer Laboratory University of Cambridge, 84.

[Ghe] G. Ghelli *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism* - Technical Report TD- 6/90 from the University of Pisa.

[Fair] J. FAIRBAIRN *Some Typeswith Inclusion Properties in* $\forall, \rightarrow, \mu$ Tech. Report 31 Computer Laboratory University of Cambridge, June 89.

[JM] L. Jategaonkar and J.C. Mitchell *ML with Extended Pattern Matching and Subtypes* ACM Conference on LISP and Functional Programming July 88.

[J] L. Jategaonkar *ML with Extended Pattern Matching and Subtypes* Tech Report MIT/LCS/TR-468, August 89.

[MPS] D.MACQUEEN, G. PLOTKIN and R. SETHI An Ideal Model for Recursive Polymorphic Types Info&Control 71

[Mitc] J.MITCHELL *Coercion and Type Inference* ACM Symp on LISP and Functional Programming, 82.

[Mitc] J.MITCHELL *Polymorphic Type Inference and Containment* Info&Comp 76

[Mitc] J.MITCHELL *A Type-Inference Approach to Reduction Properties and Semantics of Polymorphic Expressions* POPL-86

[Wan] M. Wand *Type Inference for record concatenation and multiple inheritance* LICS'89.