Polymorphic Subtype Inference: Closing the Theory-Practice Gap

You-Chin Fuh Prateek Mishra
Department of Computer Science
The State University of New York at Stony Brook
Stony Brook, New York 11794-4400
ARPANET: yfuh@suny.sbcs.edu, mishra@suny.sbcs.edu

1 Introduction

The theory of polymorphic type inference[Mil78] has been extended to deal with subtypes: Mitchell gave a subtype extension [Mit84], and we have given algorithms and correctness proofs for several key components of a type inference system based on Mitchell's extension [FM88]. However, the actual implementation of a subtype inference system *diverges* from these theoretical underpinnings in fundamentally important ways. By examining this divergence, we close the gap between the theoretical foundations of subtype inference and its actual use.

We give two results: first, we observe that Mitchell's instance relation is exactly a preorder and hence principal types are unique only up to equivalence. We show the existence of a unique minimal representative for each equivalence class of typings and give an algorithm for computing such representatives. In practice, we find it unnecessary to transform typings into their exact minimal forms. Instead, it is sufficient to remove certain redundancies from typings; this transforms most typings into their exact minimal form. We describe an efficient algorithm that implements this transformation and prove its correctness.

Second, we propose a new "lazy" instance definition, that defers coercions wherever possible. The advantage of the lazy instance definition over Mitchell's is that its use leads to an extremely compact representation for the unique minimal form of a typing. Following the framework described above, we describe a transformation, based on the lazy instance definition, that eliminates certain redundancies from typings. For many programs, removing redundancies from typings under the lazy instance definition leads to a coercion set of size at most *one*.

1.1 Computing Unique Principal Types

Previous work [Mit84,FM88] has shown that in the presence of subtypes, both the notion of type and the definition of instance must be suitably generalized. Thus, a type is written as a combination of a coercion set and type expression pair (C,t). The *instance* relation on typings $C, A \vdash N : t$ is correspondingly generalized: typing $C', A' \vdash N : t'$ is an instance of $C, A \vdash N : t$ iff there exists a substitution S such that $A' = S(A), t' = S(t)$ and $C' \Vdash S(C)$.

We implemented two provably correct algorithms based on the above definitions [Mit84,FM88]. Each algorithm produced a different type for the same program! While the types produced were consistent with each other, each being an instance of the other, this was clearly unsatisfactory. Upon examination, we discovered that the *instance* relation as defined above is exactly a *preorder*. Thus, while programs do possess principal types, the principal type is *unique* only up to equivalence under the instance relation. This explained the phenomenon of "multiple" principal types.

*Supported in part by NSF CCR-8706973

Example 1 *The following typings are both principal types for the identity function; it is important also to observe that C_1 is not equivalent to C_2.*

$$C_1 = \{\alpha \rhd \gamma, \gamma \rhd \beta\}, \emptyset \vdash \lambda x.x : \alpha \to \beta, \quad C_2 = \{\alpha \rhd \beta\}, \emptyset \vdash \lambda x.x : \alpha \to \beta$$

As the preorder nature of instance was not recognized in previous work [Mit84,FM88], the type inference algorithms described therein choose *one* possible representation for the principal type of a program. This explained why two different, correct, algorithms produced two different types. In addition we also found both representations unwieldy: The coercion set is large, 40-60 coercions for even toy programs, and has size proportional to *program size.* This effects the readability of typings: How does the user decipher the meaning of such a typing?, and efficiency of type inference: Checking consistency of a coercion set requires an algorithm quadratic in its size.

We solve both problems, lack of uniqueness and size, by providing a precise, technical definition of minimal typing. We prove that each equivalence class of typings possess a unique minimal representation. The main idea is that we can eliminate *most* type variables occurring in the coercion set which do *not* occur in the type assumption or inferred type. Simply removing *all* such type variables from a typing is unsound, as some may place constraints on the type assumption or inferred type, as in the example below.

Example 2

$$napply \equiv \lambda f.\lambda x.\lambda n.\text{if } n = 0 \text{ then } x \text{ else } f(napply\ f\ x\ n - 1).$$

The principal typing for *napply* is:

$$C, \emptyset \vdash napply : (\alpha \to \beta) \to (v_x \to (v_n \to v))$$

where $\{v_x \rhd u, \beta \rhd u, u \rhd v, u \rhd \alpha\}$ is a subset of C and u does not occur in the inferred type or type assumption. If we get rid of all the coercions involving u we lose the constraint that in any type *napply* possesses, α and v must have a common subtype, u, which is a common supertype of v_x and β. □

In practice, we find that it appears unnecessary to transform typings into their exact minimal representations. In its place, we provide a definition for a "redundant" type variable in coercion sets and show that the elimination of such variables must yield an equivalent typing. We show that the repeated elimination of "redundant" variables defines a function over typings and can therefore be used to transform typings. An algorithm that implements the function is described; the algorithm is cubic in the size of the original coercion set and reduces coercion set size to be proportional to the size of the type expression.

1.2 A "lazy" instance definition

To defer coercions occurring in typings, we introduce the following definition of lazy instance: typing $C', A' \vdash N : t'$ is an instance of $C, A \vdash N : t$ iff there exists substitution S such that $C' \Vdash S(C), C' \Vdash A' \rhd S(A), C' \Vdash S(t) \rhd t'$. Our definition is a strict extension of the original; the difference lies in allowing the use of the information describing the context encoded in the coercion set C' to mediate the relationship between typings and in weakening the relationship between types from equality to inclusion ($\rhd$). As a consequence, many typings which were not previously *instance* related become *instance* related. Following the discussion above , we have not found it necessary to transform typings to exact minimal forms based on our instance relation. Instead, we use two transformations on typings that map most typings into a minimal form with respect to lazy instance. We describe these transformations below.

1.2.1 Cycle Elimination

Mitchell's definition of type inclusion permits subtype symmetry: two distinct types, t_1 and t_2, may contain each other. As a consequence coercion sets often contain cycles of inter-related type variables and constants. Under our lazy instance relation we can equate all the participants in such a cycle and derive a much simpler but equivalent typing. Such a transformation cannot be proven correct using Mitchell's notion of instance, even if we restrict the definition of type inclusion to be anti-symmetric.

Example 3 Let $C_1 = \{\alpha \rhd \beta, \beta \rhd \alpha\}, A_1 \vdash N : t_1 = \alpha \to \beta$ and $C_2 = [\alpha/\beta](C_1), A_2 = [\alpha/\beta](A_1) \vdash N : t_2 = [\alpha/\beta](t_1)$ be typings. It is easy to see that $C_2, A_2 \vdash N : t_2$ is an instance of $C_1, A_1 \vdash N : t_1$ w.r.t. the original definition; however the opposite is not true as we cannot find substitution S such that $t_1 = S(t_2)$. It is not difficult to see that the two typings are equivalent with respect to our lazy instance definition. □

1.2.2 Eliminating Unnecessary Coercions under lazy Instance

The principal type of a program P, as standardly derived, contains many coercions that can be *deferred* to the context in which P is actually used.

Example 4

$$comp \equiv \lambda f.\lambda g.\lambda x.f(g(x))$$

The principal type:

$$\{\kappa \rhd \alpha, \sigma \rhd \gamma, \beta \rhd \rho\}, \emptyset \vdash comp : (\alpha \to \beta) \to (\gamma \to \kappa) \to (\sigma \to \rho)$$

□

In our approach, *all* the coercions in the type of *comp* may be deferred to the context of its use. Hence, its type is

$$\{\,\}, \emptyset \vdash comp : (\alpha \to \beta) \to (\gamma \to \alpha) \to (\gamma \to \beta)$$

When *comp* is used in expression *comp* M, we "recover" the lost coercion by permitting the type of M to be a subtype of $(\alpha \to \beta)$. In this manner, all the coercions in the original typing for *comp* are recovered when comp is actually used. Thus, the only coercions that must appear in a typing for a term N are those which are "internal" and cannot be deferred to the context in which N is used.

Following the treatment of redundancy removal given for Mitchell's instance relation , we define a transformation on typings that removes redundancies from typings with respect to our lazy instance definition. A definition of redundant type variable is given and we prove that their repeated elimination defines a function over typings. We have found that for programs *without* multiple occurrences of lambda-bound names in widely differing contexts, the use of our lazy instance definition yields a coercion set of size at most *one*. The coercion set usually involves only type variables that occur in the type expression. Some examples of programs with non-trivial typings may be found in section 6.

2 Preliminaries

Value expressions N and type expressions t are defined by the following abstract syntax.

$$N ::= c \mid x \mid \lambda x.N \mid N_1 N_2$$
$$t ::= g_c \mid t_1 \to t_2 \mid \alpha$$

A type expression t that contains no type variable α is called a *monotype*; other type expressions are called *polytypes*. A general coercion is an ordered pair of types written $t_1 \rhd t_2$; a coercion is an ordered pair of types $t_1 \rhd t_2$, where both t_1 and t_2 are atomic types (g_c, α). A (general) coercion set $C = \{t_i \rhd r_i\}$ is a set of (general) coercions. By $types(C)$ we will mean the set of types occurring in coercion set C. A substitution S is a map from type variables to type expressions that is not equal to the identity function at only finitely many type variables. $[t_1/\alpha_1, \ldots, t_n/\alpha_n]$ is the substitution that maps each α_i to t_i and is otherwise equal to the identity function. A type assumption A is a finite map from value variables to type expressions. By $A\ ;\ \bar{x} : \bar{s}$ we mean a type assumption identical to A, except that it maps value variable x_i to type expression s_i. If $domain(A) = domain(A')$, by $A \rhd A'$ we mean the coercion set $\{\ A(x) \rhd A'(x) \mid\ x \in domain(A)\}$. A *typing statement* is written $C, A \vdash N : t$. The type inference rules are given by:

$[VAR]$	$C, A \vdash x : A(x)$
$[CONSTANT]$	$C, A \vdash c : g_c$
$[APPLY]$	$\dfrac{C, A \vdash N_1 : t_1 \to t_2, N_2 : t_1}{C, A \vdash N_1 N_2 : t_2}$
$[ABS]$	$\dfrac{C, A\ ;\ x : t_1 \vdash N : t_2}{C, A \vdash \lambda x.N : t_1 \to t_2}$
$[COERCE]$	$\dfrac{C, A \vdash N : t_1, C \Vdash \{t_1 \rhd t_2\}}{C, A \vdash N : t_2}$

where the entailment relation $\Vdash$ between coercion sets is defined by the following rules:

$[AXIOM]$	$C \cup \{t_1 \rhd t_2\} \Vdash \{t_1 \rhd t_2\}$	$[CONST]$	$C \Vdash \{g_{c_i} \rhd g_{c_k}\}$
$[TRANS]$	$\dfrac{C \Vdash \{t_1 \rhd t_2, t_2 \rhd t_3\}}{C \Vdash \{t_1 \rhd t_3\}}$	$[REFLEX]$	$C \Vdash \{t \rhd t\}$
$[AND]$	$\dfrac{C \Vdash C_1 \ \wedge\ C \Vdash C_2}{C \Vdash C_1 \cup C_2}$	$[ARROW]$	$\dfrac{C \Vdash \{t_1 \rhd t_1', t_2' \rhd t_2\}}{C \Vdash \{t_1' \to t_2' \rhd t_1 \to t_2\}}$

Since the set of coercions entailed by a coercion set is transitive-closed, thanks to $[TRANS]$ rule, we can always replace a sequence of [COERCE] steps in a proof of a typing statement by a single $[COERCE]$ step. This suggests the following characterization of the set of provable typing statements:

A typing statement $\boxed{C, A \vdash N : p}$ is a *typing* if

$\boxed{N \equiv c}$ $C \Vdash \{g_c \rhd p\}$.
$\boxed{N \equiv x}$ $C \Vdash \{A(x) \rhd p\}$.
$\boxed{N \equiv \lambda x.M}$ $C, A\ ;\ x : t_1 \vdash M : t_2$ is a typing and $C \Vdash \{t_1 \to t_2 \rhd p\}$.
$\boxed{N \equiv M_1 M_2}$ $C, A \vdash M_1 : t_1 \to t_2, M_2 : t_1$ are typings and $C \Vdash \{t_2 \rhd p\}$

If $C, A \vdash N : t$ is a typing, the following specializations of $C, A \vdash N : t$ are also typings:

- $\gamma(C), \gamma(A) \Vdash N : \gamma(t)$, where γ is a substitution.
- $C', A \vdash N : t$, where $C' \Vdash C$.
- $C, A' \vdash N : t$, where $C \Vdash A'|_{FV(N)} \rhd A|_{FV(N)}$.
- $C, A \vdash N : t'$, where $C \Vdash \{t \rhd t'\}$.

That the first two specializations are typings is obvious. The other two cases are also true because we can always expand inferred type and type assumption by $[COERCE]$ steps. Therefore, the typing $C, A \vdash N : t$ can be considered as more general than any of its specializations. By combining the specializations shown above, we define the *instance* ordering on typings as follows:

Definition 1 *The typing $C', A' \vdash N : t'$ is an instance of the typing $C, A \vdash N : t$, written as $C, A \vdash N : t \preceq C', A' \vdash N : t'$, if $\exists$ substitution γ such that:*

- $C' \Vdash \gamma(C)$.
- $C' \Vdash A'|_{FV(N)} \rhd \gamma(A)|_{FV(N)}$.
- $C' \Vdash \gamma(t) \rhd t'$.

Lemma 1 *Typings are closed under the instance relation; i.e. if $C, A \vdash N : t$ is a typing then so is every instance $C', A' \vdash N : t'$.*

Proof: Given above. □

Since a typing can be considered as denoting all its instances, it's natural to regard two typings equivalent if they are instances of each other. More precisely, $C_1, A_1 \vdash N : t_1 \cong C_2, A_2 \vdash N : t_2$ iff $C_1, A_1 \vdash N : t_1 \preceq C_2, A_2 \vdash N : t_2$ and $C_2, A_2 \vdash N : t_2 \preceq C_1, A_1 \vdash N : t_1$.

A typing statement $\boxed{C, A \vdash N : p}$ is a *normal typing* if

$\boxed{N \equiv c}$ $g_c = p$.
$\boxed{N \equiv x}$ $A(x) = p$.
$\boxed{N \equiv \lambda x.M}$ $C, A\,;\, x : t_1 \vdash M : t_2$ is a normal typing and $p = t_1 \to t_2$.
$\boxed{N \equiv M_1 M_2}$ $C, A \vdash M_1 : t_1 \to p, M_2 : t_1'$ are normal typings and $C \Vdash \{t_1' \rhd t_1\}$

The idea behind *normal typing* is that the only essential coercions in a typing are those at application terms $M_1 M_2$. This corresponds to our intuition that it's enough only to coerce values passed as arguments to functions. Our definition of normal typing is also the basis for algorithm TYPE in section 3 in which we carry out coercion steps only at argument subterm of function application during type inference. The following lemma justifies this intuition.

Lemma 2 *Let $C, A \vdash N : t$ be a typing. Then there exists a normal typing $C', A' \vdash N : t'$ such that $C', A' \vdash N : t' \preceq C, A \vdash N : t$.*

Proof: By structural induction on N. □

3 Algorithm TYPE

We say general coercion $t_1 \rhd t_2$ is *matching* if t_1 and t_2 are either both atomic types or both function types with matching range and domain. In previous work [FM88], we have described algorithm MATCH together with its correctness proof. Given a general coercion set C, algorithm MATCH either finds the most general substitution S such that every coercion in $S(C)$ is matching or else it fails. If MATCH fails, no substitution that renders each coercion in C matching exists. In [FM88] we have also described algorithm SIMPLIFY, which maps a matching coercion set C into an equivalent set C' consisting only of coercions between atomic types (constants, variables).

Input $C, A, N \in$ Coercion Set $\times$ Type Assumption $\times$ Value Expression
where $FV(N) \subseteq domain(A)$.

Output $C', S, t \in$ Coercion Set $\times$ Substitution $\times$ Type Expression
such that $C', S(A) \vdash N : t$ is a typing.

$$
\begin{aligned}
TYPE(C, A, N) = \ & \textbf{case } N \textbf{ of} \\
& N \equiv c \\
& \qquad (C, Id, g_c) \\
& N \equiv x \\
& \qquad (C, Id, A(x)) \\
& N \equiv \lambda x.M \\
& \qquad \text{let } (C', S, t) = TYPE(C, A\ ;\ x : \alpha, M) \\
& \qquad \text{where } \alpha \text{ is a fresh type variable,} \\
& \qquad \text{in } (C', S, S(\alpha) \to t) \\
& N \equiv M_1 M_2 \\
& \qquad \text{let } (C_1, S_1, t_1) = TYPE(C, A, M_1); \\
& \qquad (C_2, S_2, t_2) = TYPE(C_1, S_1(A), M_2); \\
& \qquad R = UNIFY(S_2(t_1), \alpha \to \beta), \text{ where } \alpha, \beta \text{ are new} \\
& \qquad R' = MATCH(RC_2 \cup \{R(t_2) \rhd R(\alpha)\}); \\
& \qquad C' = SIMPLIFY(R'RC_2 \cup \{R'R(t_2) \rhd R'R(\alpha)\}), \\
& \qquad \text{in} \\
& \qquad (C', R'RS_2S_1, R'R(\beta))
\end{aligned}
$$

Figure 1: Algorithm TYPE

It is useful to compare the algorithm in Figure 1 with Mitchell's algorithm TYPE in [Mit84]. Mitchell's algorithm does not incorporate a type assumption (A above) that is "shared" between subexpressions of an expression; instead, in an application M_1M_2 both M_1 and M_2 are typed independently using independent type assumptions A_1 and A_2, which must later be reconciled using unification and matching. This may prove to be expensive as the size of a type assumption can be quite large in practice as it is bounded only by the total number of names defined in a program.

Theorem 1 TYPE is Sound
$(C', S, t) = TYPE(C, A, N)$ *succeeds* $\Longrightarrow C' \Vdash S(C) \wedge C', S(A) \vdash N : t$ *is a typing.*

Proof: See appendix. □

Theorem 2 TYPE is Complete
If $C_*, A_* \vdash N : t_*$ *is a typing and* $\exists T$ *s.t.* $C_* \Vdash T(C) \wedge A_*|_{FV(N)} = T(A)|_{FV(N)}$ *then* $(C', S, t) = TYPE(C, A, N)$ *succeeds and* $C_*, A_* \vdash N : t_*$ *is an instance of* $C', S(A) \vdash N : t$.

Proof: See appendix. □

3.1 Consistency checking

We say coercion set C is *consistent* if $C \Vdash g_{c_i} \rhd g_{c_k} \Rightarrow \emptyset \Vdash g_{c_i} \rhd g_{c_k}$. Expressed in words: a coercion set is consistent if every monotype coercion consequence is "true". From practical point of view, typings with inconsistent coercion set gives no information and hence should be excluded. Consistency checking can be implemented by examining the set of monotypes reachable from each monotype contained in coercion set C. Define a *well-typing* to be a typing $C, A \vdash N : t$, where C is consistent. How do we compute well-typings? We simply run algorithm TYPE and check the final coercion set for consistency. If it is consistent, we are done; otherwise, we fail and conclude that no well-typing exists:

let $(C', S, t) = TYPE(C, A, N)$ in
if C' is consistent then (C', S, t) such that $C', S(A) \vdash N : t$ is a well-typing
else fail

That this method is sound is obvious; its completeness is argued as follows: To see that the above is complete, we need to consider two cases. For the first, we have that C' is consistent. But then, the syntactic completeness of TYPE ensures the syntactic completeness of the above algorithm. For the second case, let C' be inconsistent. We will argue that no well-typing $C_*, A_* \vdash N : t_*$ exists. Assume otherwise; as TYPE is syntactically complete we can find substitution γ with $C_* \Vdash \gamma(C')$. Now, since C_* is consistent we must have that $\gamma(C')$ is consistent. But then C' must be consistent as well and we have arrived at a contradiction.

4 Computing Principal Types

4.1 Existence and Uniqueness

The elimination of type variables from the coercion set is based on the following insight: for any typing, types occurring in the coercion set component can be partitioned into two classes: those that are *visible* or *observable* in the sense future coercions may refer to them; Types that are not visible will never be involved in any future coercions. All type constants are observable, as are type variables that occur either

in the type assumption or the inferred type. All other types occurring in the coercion set are not visible. For example, consider the typing:

$$\{v \rhd \alpha, v \rhd \eta, \alpha \rhd int, \beta \rhd \gamma, \beta \rhd \alpha\}, \emptyset \vdash N : \alpha \to \eta$$

then α, η, int are observable and v, β, γ are not.

Intuitively speaking, type variables that are not observable are useless unless they constrain observable variables by "connecting" them together. In the example above β and γ are useless and we could dispense completely with the coercions $\beta \rhd \alpha$ and $\beta \rhd \gamma$ and arrive at a smaller and equivalent coercion set. However, although v is not observable we can not get rid of the coercions $v \rhd \alpha$ and $v \rhd \eta$ which involve v.

Following the discussion above, define $Obv(C, A \vdash N : t)$ to be the set of type variables occuring in A or t and let $Intv(C, A \vdash N : t)$ be $Vars(C) - Obv(C, A \vdash N : t)$. A substitution S is a renaming on the set V of type variables if the restriction of S to V is one to one. Similarly, S is an identity on V if the restriction of S to V is an identity.

Definition 2 A typing $C, A \vdash N : t$ is minimal iff S is an identity on $Obv(C, A \vdash N : t) \wedge C \| \!- S(C) \implies$ S is a renaming on type variables in $C, A \vdash N : t$. A typing is redundant if it is not minimal.

Observe that whenever $C \| \!- S(C)$, S must map variables in $Intv(C, A \vdash N : t)$ to either the variables in $Vars(C)$ or type constants. Further, without loss of generality we can assume S is an identity on variables outside $Vars(C)$. Define $Subs(C, A \vdash N : t)$ to be the set of substitutions S where:

$$S(v) = \begin{cases} v, & v \notin Intv(C, A \vdash N : t) \\ \text{a type constant or a variable } \in Vars(C), & v \in Intv(C, A \vdash N : t) \end{cases}$$

It's not difficult to see that $Subs(C, A \vdash N : t)$ is finite, as the number of distinct variables in a typing is finite. Thus the minimality of a typing can be decided by checking the condition $C \| \!- S(C)$ for all substitution $S \in Subs(C, A \vdash N : t)$. The following algorithm computes minimal typings:

minimize$(C, A \vdash N : t) =$

if $\exists S \in Subs(C, A \vdash N : t)$ such that $C \| \!- S(C) \wedge S$ not a renaming on $Intv(C, A \vdash N : t)$

then minimize$(S(C, A \vdash N : t))$

else $C, A \vdash N : t$

Since $Subs(C, A \vdash N : t)$ is finite and checking the entailment between two finite coercion sets is decidable the "if" condition can be effectively evaluated. Since the substitution satisfying the "if" condition must remove at least one variable in $Intv(C, A \vdash N : t)$, $\mid Intv(C, A \vdash N : t) \mid$ is strictly decresing. Therefore **minimize** always terminates and returns a minimal typing as its final result. This proves the existence of minimal typings.

Let C be an acyclic coercion set. By C^* we mean the reflexive, transitive closure of C. It's obvious that C^* exactly consists of the atomic coercions entailed by C. More precisely:

$$C \| \!- a \rhd a' \ \ iff \ \ a \rhd a' \in C^*$$

A coercion set C is equivalent to the coercion set C', written $C \equiv C'$, iff $C \| \!- C'$ and $C' \| \!- C$ or, equivalently, $C^* = C'^*$. The equivalence of two coercion sets is preserved under substitution.

Theorem 3 Minimal Typing Is Unique

If $C_1, A_1 \vdash N : t_1 \cong C_2, A_2 \vdash N : t_2$ are minimal typings then there exists a renaming S such that:

- $t_1 = S(t_2)$.
- $A_1|_{FV(N)} = S(A_2|_{FV(N)})$.
- $C_1 \equiv S(C_2)$.

Proof: By definition of $\cong$ there exists S_1, S_2 such that:

(1) $t_1 = S_1(t_2)$ and $t_2 = S_2(t_1)$.

(2) $A_1|_{FV(N)} = S_1(A_2|_{FV(N)})$ and $A_2|_{FV(N)} = S_2(A_1|_{FV(N)})$.

(3) $C_1 \Vdash S_1(C_2)$ and $C_2 \Vdash S_2(C_1)$.

By (1) and (2) we have $t_2 = S_2S_1(t_2)$ and $A_2|_{FV(N)} = S_2S_1(A_2|_{FV(N)})$ and hence S_2S_1 must be an identity on $Var(t_2) \cup Var(A_2|_{FV(N)})$. This in turn entails that S_1 is a renaming on $Var(t_2) \cup Var(A_2|_{FV(N)})$.

We now proceed to show S_1 is also a renaming on $Var(C_2) - (Var(t_2) \cup Var(A_2|_{FV(N)}))$ and $C_1 \equiv S_1(C_2)$. By (3) we have $C_2 \Vdash S_2S_1(C_2)$. Since $C_2, A_2 \vdash N : t_2$ is minimal, by the definition of minimality of typings S_2S_1 is a renaming on $Var(C_2) - (Var(t_2) \cup Var(A_2|_{FV(N)}))$ and hence S_1 is a renaming on $Var(C_2) - (Var(t_2) \cup Var(A_2|_{FV(N)}))$. By the same argument we have S_2 is a renaming on $Var(C_1) - (Var(t_1) \cup Var(A_1|_{FV(N)}))$. Since $C_1 \Vdash S_1(C_2)$, ${C_1}^* \supseteq (S_1(C_2))^*$ and hence $\mid {C_1}^* \mid \geq \mid (S_1(C_2))^* \mid$. Since renaming preserves the cardinality of coercion sets we can use (3) to get the fact that $\mid (S_1(C_2))^* \mid = \mid (C_2)^* \mid \geq \mid (S_2(C_1))^* \mid = \mid {C_1}^* \mid$. Combining the above results we conclude that ${C_1}^* = (S_1(C_2))^*$, as $\mid {C_1}^* \mid = \mid (S_1(C_2))^* \mid$ and ${C_1}^* \supseteq (S_1(C_1))^*$, and hence $C_1 \equiv S_1(C_2)$. Let S be S_1 then we are done. □

4.2 Removing Redundancy from Typings

By Theorem 3, eliminating redundancy in two equivalent typings ends up with the same minimal typing up to renaming of type variables and the equivalence of coercion sets. This suggests a unique representative for equivalent typings and thus the preorder on typings can be replaced by a partial order on minimal typings.

In practice, we have not found it necessary to compute exact minimal typings. There are two reasons for this: first, the computation of such a form appears to require exhaustive checking of the condition $C \Vdash S(C)$ for all $S \in Subs(C, A \vdash N : t)$; second, we find most redundancy is of a simple form which can be efficiently detected and removed.

Let C be a coercion set and a be an atomic type in C. Define:

$$above_C(a) = \{a' | a \rhd a' \in C^*\}$$
$$below_C(a) = \{a' | a' \rhd a \in C^*\}$$

Definition 3 *Variable $\alpha \in types(C)$, is G-subsumed by $\beta \in types(C)$ in C, written $\alpha \leq_G \beta$ in C, if*

1. $above_C(\alpha) - \{\alpha\} \subseteq above_C(\beta)$.
2. $below_C(\alpha) - \{\alpha\} \subseteq below_C(\beta)$.

In the following lemma, we show that if $\alpha \leq_G \beta$ in C then we can identify α with β without affecting any coercion consequence of C which does not involve α.

Lemma 3 *If $\alpha \leq_G \beta$ in C and $C' = [\beta/\alpha](C)$ then:*

$$C \Vdash a \rhd a' \wedge a \neq \alpha \wedge a' \neq \alpha \iff C' \Vdash a \rhd a'$$

Proof:

($\Longrightarrow$):

Since $C \| \!\!- a \rhd a'$, $a \rhd a' \in C^*$. Hence $a \rhd a' \in [\beta/\alpha](C^*)$. Since $C \equiv C^*$ and $\equiv$ is preserved under substitution, $C' = [\beta/\alpha] \equiv [\beta/\alpha](C^*)$ and hence $C' \| \!\!- a \rhd a'$.

($\Longleftarrow$):

Let $Deleted\_coercions = \{a \rhd a' \in C \mid a = \alpha \text{ or } a' = \alpha\}$ and $Added\_coercions = \{a \rhd \beta \mid a \rhd \alpha \in C\} \cup \{\beta \rhd a \mid \alpha \rhd a \in C\}$. Observe that $C' = (C \cup Added\_coercions) - Deleted\_coercions$. Since $\alpha \leq_G \beta$ in C, the coercions in $Added\_coercions$ is actually entailed by C. Therefore, the set of coercions entailed by C' is a subset of that entailed by C.

□

The following consequences of Lemma 3 will be used frequently in the rest of this section:

fact1 if $\alpha \leq_G \beta$ in C then $C \| \!\!- [\beta/\alpha](C)$.

fact2 if $\alpha \leq_G \beta$ in C and $C' = [\beta/\alpha](C)$ then $\forall a \in C'$, $above_{C'}(a) = above_C(a) - \{\alpha\} \wedge below_{C'}(a) = below_C(a) - \{\alpha\}$.

fact3 Let $C \equiv C'$, $\alpha \leq_G \beta$ in C, and $\alpha \leq_G \beta'$ in C', then $[\beta/\alpha](C) \equiv [\beta'/\alpha](C')$.

Define the relation $\mapsto_G$ on typings as follows:

$$C, A \vdash N : t \mapsto_G C', A \vdash N : t \iff \begin{cases} (1)\ \alpha \leq_G \beta \text{ in C.} \\ (2)\ \alpha \text{ not in } A \text{ or } t. \\ (3)\ C' = [\beta/\alpha](C). \end{cases}$$

Let $C, A \vdash N : t \ \mapsto_G \ C', A \vdash N : t$ by $[\beta/\alpha]$ and $C' = [\beta/\alpha](C)$. We show that $C, A \vdash N : t \cong C', A \vdash N : t$. That $C', A \vdash N : t$ is an instance of $C, A \vdash N : t$ is obvious. To show the opposite direction observe, by **fact1**, that we have $C \| \!\!- [\beta/\alpha](C)$, hence $C, A \vdash N : t$ is also an instance of $[\beta/\alpha](C), A \vdash N : t$. This result can be extended to any sequence of $\mapsto_G$ steps. Therefore, $C, A \vdash N : t \ \overset{*}{\mapsto}_G \ C', A \vdash N : t$ implies $C', A \vdash N : t \cong C, A \vdash N : t$.

For our purposes, the most important property of $\mapsto_G$ is that it defines a *function* on typings up to the equivalence of coercion sets and thus maps each typing to a unique equivalent typing without "useless" variables.

Lemma 4 *Let* $C \equiv \overline{C}$.

$$\left\{ \begin{array}{llll} C, A \vdash N : t & \mapsto_G & C_1, A \vdash N : t & by\ [b_1/a_1] \\ \overline{C}, A \vdash N : t & \mapsto_G & C_2, A \vdash N : t & by\ [b_2/a_2] \end{array} \right\} \Longrightarrow \exists C' \equiv \overline{C}', \ such\ that:$$

$$\left\{ \begin{array}{lll} C_1, A \vdash N : t & \overset{*}{\mapsto}_G & C', A \vdash N : t \\ C_2, A \vdash N : t & \overset{*}{\mapsto}_G & \overline{C}', A \vdash N : t \end{array} \right\}$$

Proof:

$a_1 \neq b_2$ and $a_2 \neq b_1$:

If $a_1 = a_2$ then by **fact3** of lemma 3 $C_1 \equiv C_2$ and no further reduction sequences are needed to reconcile $C_1, A \vdash N : t$ and $C_2, A \vdash N : t$. Assume otherwise, then by **fact2** the subsumption relation between a_1 and b_1 in C (resp. a_2 and b_2 in $\overline{C}$) is not effected by the substitution $[b_2/a_2]$ (resp. $[b_1/a_1]$). Hence we can apply the one-step reduction $[b_2/a_2]$ to $C_1, A \vdash N : t$ and the one-step reduction $[b_1/a_1]$ to $C_2, A \vdash N : t$. By **fact2** this reconciles $C_1, A \vdash N : t$ and $C_2, A \vdash N : t$.

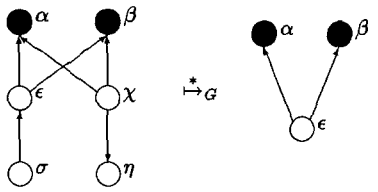

Figure 2

$a_1 \neq b_2$ and $a_2 = b_1$:

Since $a_1 \leq_G b_1$ and $b_1 = a_2 \leq_G b_2$, by transitivity of $\leq_G$, $a_1 \leq_G b_2$. By fact2 the subsumption relation between a_1 and b_2 in $\overline{C}$ (resp. a_2 and b_2 in C) is not effected by the substitution $[b_2/a_2]$ (resp. $[b_1/a_1]$). Therefore we can apply the one-step reduction $[b_2/a_1]$ to $C_2, A \vdash N : t$ and apply the one-step reduction $[b_2/a_2]$ (or $[b_2/b_1]$) to $C_1, A \vdash N : t$. By fact2 this reconciles $C_1, A \vdash N : t$ and $C_2, A \vdash N : t$.

$a_1 = b_2$ and $a_2 \neq b_1$: Same as previous case.

$a_1 = b_2$ and $a_2 = b_1$:

Then C_1 is identical to C_2 up to variable renaming.

□

Theorem 4 *Every typing $C, A \vdash N : t$ has a unique normal form under $\mapsto_G$ up to variable renaming and the equivalence of coercion sets.*

Proof: As $|types(C_i)|$ is strictly decreased in any reduction sequence, no reduction sequence continues for ever. Hence the relation $\mapsto_G$ is strongly normalizing. By Proposition 3.1.25 in [Bar84] and lemma 4, $\mapsto_G$ is Church-Rosser up to the equivalence of coercion sets. Hence every typing has a normal form upto equivalence under $\mapsto_G$. □

Example 5 *Let $C = \{\sigma \rhd \epsilon, \epsilon \rhd \alpha, \epsilon \rhd \beta, \chi \rhd \alpha, \chi \rhd \beta, \chi \rhd \eta\}$ be a coercion set where only α and β occur in the inferred type or type assumption. The minimized form of C is shown by Figure 2.*

By the theorem above, $\mapsto_G$ defines a function on typings up to the identification of equivalent coercion sets. In other words, starting from a given typing, different "maximal" sequences of $\mapsto_G$ steps all end up with typings with the same A and t components and equivalent but possibly different coercion set component. If it is so desired a unique coercion set can always be derived by replacing the transformed coercion set by its transitive reduction.

We implement $\mapsto_G$ in the following manner: we top-sort the coercion set and compute *above* and *below* sets for vertices in two passes over the coercion set. Comparison of *above* and *below* can be implemented in constant time by using the standard bit-string implementation of finite (and small) sets. Finding a subsumable variable may take up to n^2 comparisons in an n vertex graph; there are at most n-vertices in an n-vertex coercion set that can be G-subsumed by other vertices. Therefore, in worst case, the complexity is cubic in the size of the coercion set component.

5 Eliminating Unnecessary Coercions under lazy Instance

The *expand_polarity* of a type variable α in a type t is a subset of $\{\uparrow,\downarrow\}$ indicating the effect on occurrences of α in t when t is coerced into a super type t'. There are four ways in which occurrences of α may be effected: no effect, expansion, contraction and a combination of expansion and contraction. These four possibilities are represented by $\emptyset$, $\{\uparrow\}$, $\{\downarrow\}$, and $\{\uparrow,\downarrow\}$ respectively. The symmetric notion, *contract_polarity*, is defined similarly.

$$expand\_polarity(\alpha,t) = \begin{cases} \{\uparrow\} & ,if\ \alpha = t. \\ contract\_polarity(\alpha,t_1) \cup expand\_polarity(\alpha,t_2) & ,if\ t = t_1 \rightarrow t_2. \\ \emptyset & ,O/W. \end{cases}$$

and

$$contract\_polarity(\alpha,t) = \begin{cases} \{\downarrow\} & ,if\ \alpha = t. \\ expand\_polarity(\alpha,t_1) \cup contract\_polarity(\alpha,t_2) & ,if\ t = t_1 \rightarrow t_2. \\ \emptyset & ,O/W. \end{cases}$$

Observe that $expand\_polarity(\alpha,t) = \{\uparrow\}$ ($\{\downarrow\}$) iff $contract\_polarity(\alpha,t) = \{\downarrow\}$ ($\{\uparrow\}$). As examples, $expand\_polarity(\alpha,(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta)) = \{\uparrow\}$ and $contract\_polarity(\alpha,(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \beta)) = \{\downarrow\}$.

Definition 4 *Let $\alpha \in types(C)$ and α occurs in t. We say α is S-subsumed by r in C and t, written $\alpha \leq_S r$ in C and t, iff either*

1. $C\|\!-\{r \rhd \alpha\}$ *and* $expand\_polarity(\alpha,t) = \{\uparrow\}$ *and* $below_C(\alpha) - \{\alpha\} \subseteq below_C(r)$.
   or
2. $C\|\!-\{\alpha \rhd r\}$ *and* $expand\_polarity(\alpha,t) = \{\downarrow\}$ *and* $above_C(\alpha) - \{\alpha\} \subseteq above_C(r)$.

Observe that $\leq_S$ is a special case of $\leq_G$ and hence lemma 3 also holds for $\leq_S$.

Let $C, A \vdash N : t$ be a typing and $domain(A) = \{x_1, ..., x_n\}$. By $type\_closure(C, A \vdash N : t)$ we mean the type expression $A(x_1) \rightarrow ... \rightarrow A(x_n) \rightarrow t$. Define the relation $\mapsto_S$ on typings by the following:

$$C, A \vdash N : t \mapsto_S C', A' \vdash N : t' \iff \begin{cases} (1)\ \alpha \leq_S \beta \text{ in } C \text{ and } type\_closure(C, A \vdash N : t) \\ (2)\ C' = [\beta/\alpha](C) \\ (3)\ A' = [\beta/\alpha](A) \\ (4)\ t' = [\beta/\alpha](t) \end{cases}$$

If $\alpha \leq_S \beta$ in C and $type\_closure(C, A \vdash N : t)$ then, without loss of generality, we can assume $C\|\!-\{\beta \rhd \alpha\}$ and $expand\_polarity(\alpha, type\_closure(C, A \vdash N : t)) = \{\uparrow\}$. By the definition of *expand_polarity*, $\forall x \in domain(A)$, either $contract\_polarity(\alpha, A(x)) = \{\uparrow\}$ or $contract\_polarity(\alpha, A(x)) = \emptyset$. Since $C\|\!-\{\beta \rhd \alpha\}$, $\forall x \in domain(A)$, $C\|\!-\{A(x) \rhd [\beta/\alpha](A(x))\}$; in other words, $C\|\!-A \rhd [\beta/\alpha](A)$. Similarly, either $expand\_polarity(\alpha,t) = \{\uparrow\}$ or $expand\_polarity(\alpha,t) = \emptyset$; either entails that $C\|\!-\{[\beta/\alpha](t) \rhd t\}$. Since lemma 3 holds for $\leq_S$, $C\|\!-[\beta/\alpha](C)$. Hence $C, A \vdash N : t$ is an instance of $[\beta/\alpha](C), [\beta/\alpha](A) \vdash N : [\beta/\alpha](t)$. This implies the equivalence of this two typings, as $[\beta/\alpha](C), [\beta/\alpha](A) \vdash N : [\beta/\alpha](t)$ is an instance of $C, A \vdash N : t$.

Following the development for $\mapsto_G$, we show that $\mapsto_S$ defines a function on typings, up to the equivalence of coercion sets.

Lemma 5 *Let $C \equiv \overline{C}$ be acyclic.*

$$\left\{ \begin{array}{llll} C, A \vdash N : t & \mapsto_S & C_1, A_1 \vdash N : t_1 & by\ [b_1/a_1] \\ \overline{C}, A \vdash N : t & \mapsto_S & C_2, A_2 \vdash N : t_2 & by\ [b_2/a_2] \end{array} \right\} \Longrightarrow \exists C' \equiv \overline{C}', A', \text{ and } t' \text{ such that:}$$

$$\left\{ \begin{array}{lll} C_1, A_1 \vdash N : t_1 & \overset{*}{\mapsto}_S & C', A' \vdash N : t' \\ C_2, A_2 \vdash N : t_2 & \overset{*}{\mapsto}_S & \overline{C}', A' \vdash N : t' \end{array} \right\}$$

Proof:

$\boxed{a_1 \neq b_2 \text{ and } a_2 \neq b_1:}$

If $a_1 = a_2$ then either $C \Vdash \{a_1 \rhd b_1, a_1 \rhd b_2\}$ or $C \Vdash \{b_1 \rhd a_1, b_2 \rhd a_1\}$. This entails either $b_1 \in above_C(a_1) - \{a_1\} \subseteq above_C(b_2) \wedge b_2 \in above_C(a_1) - \{a_1\} \subseteq above_C(b_1)$ or $b_1 \in below_C(a_1) - \{a_1\} \subseteq below_C(b_2) \wedge b_2 \in below_C(a_1) - \{a_1\} \subseteq below_C(b_1)$. Hence $b_1 = b_2$, as C is acyclic. Assume otherwise, by $\boxed{\text{fact2}}$ the subsumption relation between a_1 and b_1 in $\overline{C}$ (resp. a_2 and b_2 in C) is not effected by the substitution $[b_2/a_2]$ (resp. $[b_1/a_1]$). Therefore, we can apply the one-step reduction $[b_2/a_2]$ to $C_1, A_1 \vdash N : t_1$ and the one-step reduction $[b_1/a_1]$ to $C_2, A_2 \vdash N : t_2$. This reconciles $C_1, A_1 \vdash N : t_1$ and $C_2, A_2 \vdash N : t_2$.

$\boxed{a_1 \neq b_2 \text{ and } a_2 = b_1:}$

If $C \Vdash \{a_1 \rhd b_1, b_2 \rhd b_1\}$ then $a_1 \in below_C(b_1) - \{b_1\} \subseteq below_C(b_2)$. This entails $b_2 \in above_C(b_1) - \{b_1\}$, as $b_2 \in above_C(a_1) - \{a_1\} \subseteq above_C(b_1)$. Hence, $b_1 = b_2$, as $C \Vdash \{b_1 \rhd b_2, b_2 \rhd b_1\}$ and C is acyclic. The symmetric situation where $C \Vdash \{b_1 \rhd a_1, b_1 \rhd b_2\}$ can be proved similarly. The remaining possibilities are either $C \Vdash \{a_1 \rhd b_1, b_1 \rhd b_2\}$ or $C \Vdash \{b_2 \rhd b_1, b_1 \rhd a_1\}$ which entail $a_1 \leq_S b_2$ in $\overline{C}$. By $\boxed{\text{fact2}}$, the subsumption relations between a_1 and b_2 in $\overline{C}$ (resp. $b_1 = a_2$ and b_2 in C) is not effected by the substitution $[b_2/a_2]$ (resp. $[a_1/b_1]$). Therefore we can apply the one-step reduction $[b_2/a_1]$ to $C_2, A_2 \vdash N : t_2$ and apply the one-step reduction $[b_2/a_2]$ (or $[b_2/b_1]$) to $C_1, A_1 \vdash N : t_1$. This reconciles $C_1, A_1 \vdash e : t_1$ and $C_2, A_2 \vdash N : t_2$.

$\boxed{a_1 = b_2 \text{ and } a_2 \neq b_1:}$ Same as previous case.

$\boxed{a_1 = b_2 \text{ and } a_2 = b_1:}$

Then C_1 is identical to C_2, A_1 is identical to A_2, and t_1 is identical to t_2, up to variable renaming.

□

Theorem 5 *Every typing $C, A \vdash N : t$ has a unique normal form under $\mapsto_S$ up to variable renaming and the equivalence of coercion sets.*

Proof: Similar to theorem4. □

The implementation of $\overset{*}{\mapsto}_S$ is almost the same as that of $\overset{*}{\mapsto}_G$. The only difference is checking the polarity when looking for a S-subsumed variable. This can be computed efficiently. In the following program we show the final typing of *napply* after carrying out coercion elimination:

Example 6 *Consider program napply:*

$$napply = \lambda f.\lambda x.\lambda n.\ if\ \ n = 0\ \ then\ \ x\ \ else\ \ napply\ f\ (n-1)\ (fx)$$

Algorithm TYPE followed by the minimizations described before results in the following type for napply:

$$\{\eta \rhd \alpha, \beta \rhd \eta, \eta \rhd \sigma, \gamma \rhd int\}, \emptyset \vdash napply : (\alpha \to \beta) \to \eta \to \gamma \to \sigma$$

after coercion elimination (shown in Figure 3) we arrive at the following type:

$$\emptyset, \emptyset \vdash napply : (\eta \to \eta) \to \eta \to int \to \eta$$

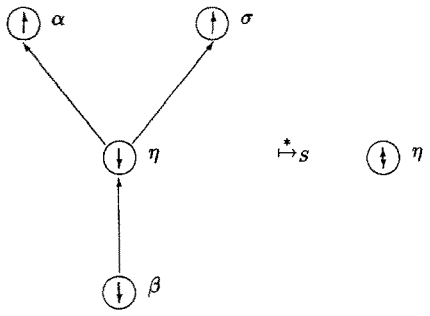

Figure 3

6 Characterizing the shape of Coercion Sets

For most programs, our techniques yield typings in which the coercion set component is at most of size one. However, it is not difficult to find examples that make full use of the power of the coercion set and type notation and where the size of the minimal coercion set is proportional to the size of the type expression component of the typing.

Example 7

$$\{\eta \rhd \alpha, \eta \rhd \tau, \gamma \rhd \tau, \sigma \rhd \alpha\}, \emptyset \vdash \lambda f.\lambda g.\lambda x.(fgx, gfx) : (\alpha \to \gamma) \to (\tau \to \sigma) \to (\eta \to \gamma * \sigma)$$

It is also possible to find examples of typings where the coercion set involves type variables that do not occur elsewhere in the typing in an essential way.

Example 8

$$\{\eta \rhd \alpha, \eta \rhd \tau, \gamma \rhd \tau\}\emptyset \vdash \lambda f.\lambda x.fst(fx, \lambda g.(fgx, gfx)) : (\alpha \to \gamma) \to (\eta \to \gamma)$$

References

[Bar84] H. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* 1984.

[FM88] Y-C Fuh and P Mishra. Type inference with subtypes. In *ESOP-88 (Also Stony Brook TR 87/25)*, March, 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. In *JCSS 17*, 1978.

[Mit84] J. C. Mitchell. Coercion and type inference. In *POPL XI*, 1984.

Appendix I: TYPE is sound and complete

Theorem 6 TYPE is Sound
$(C', S, t) = TYPE(C, A, N)$ *succeeds* $\Longrightarrow C' \Vdash S(C) \wedge C', S(A) \vdash N : t$ *is a typing.*

Proof:

$\boxed{N \equiv c:}$

Since $C' = C \wedge S = \emptyset \wedge t = g_c$, hence $C' \Vdash S(C) \wedge C', S(A) \vdash N : t$ is a typing.

$\boxed{N \equiv x:}$

Since $C' = C \wedge S = \emptyset \wedge t = A(x)$, hence $C' \Vdash S(C) \wedge C', S(A) \vdash N : t$ is a typing.

$\boxed{N \equiv \lambda x.M:}$

Let $(C', S, t') = TYPE(C, A; x : \alpha, M)$. By hypothesis

$$C' \Vdash S(C) \wedge C', S(A); x : S(\alpha) \vdash M : t'$$

is a typing. Therefore, $C', S(A) \vdash \lambda x.M : S(\alpha) \rightarrow t'$ is a typing.

$\boxed{N \equiv M_1 M_2:}$

Let $(C_1, S_1, t_1) = TYPE(C, A, M_1)$ and $(C_2, S_2, t_2) = TYPE(C_1, S_1(A), M_2)$. By hypothesis, we have:

(1) $C_1, S_1(A) \vdash M_1 : t_1$ is a typing and $C_1 \Vdash S_1(C)$.

(2) $C_2, S_2 S_1(A) \vdash M_2 : t_2$ is a typing and $C_2 \Vdash S_2(C_1) \Vdash S_2 S_1(C)$.

which in turn entail the following consequences:

(1') $C', R'RS_2S_1(A) \vdash M_1 : R'R(\alpha) \rightarrow R'R(\beta)$ is a typing,
as $R'R(\alpha) \rightarrow R'R(\beta) = R'RS_2(t_1)$, $C' \Vdash R'R(C_2) \Vdash R'RS_2(C_1)$, and typing is closed under instance relation.

(2') $C', R'RS_2S_1(A) \vdash M_2 : R'R(t_2)$ is a typing, as $C' \Vdash R'R(C_2)$ and typing is closed under instance relation.

Combining the typings in (1') and (2'), we conclude that $C', R'RS_2S_1(A) \vdash M_1 M_2 : R'R(\beta)$ is a typing, as $C' \Vdash \{R'R(t_2) \rhd R'R(\alpha)\}$. Since $C' \Vdash R'RS_2(C_1)$ and $C_1 \Vdash S_1(C)$, $C' \Vdash R'RS_2S_1(C)$.

□

In the following proof for syntactic completeness, we will assume the fact:

FACT:

Let $(C', S, t) = TYPE(C, A, N)$ succeeds.

If v does not occur in $A|_{FV(N)}$ **and** v is not created by $TYPE$ **then** $S(v) = v$.

This corresponds to the intuition: if the type inference algorithm $TYPE$ does not refer to variable v it does not affect v at all.

Theorem 7 TYPE is Complete

If $C_*, A_* \vdash N : t_*$ *is a typing and* $\exists T$ *s.t.* $C_* \Vdash T(C) \wedge A_*|_{FV(N)} = T(A)|_{FV(N)}$ *then* $(C', S, t) = TYPE(C, A, N)$ *succeeds and* $C_*, A_* \vdash N : t_*$ *is an instance of* $C', S(A) \vdash N : t$.

proof:

$\boxed{N \equiv c:}$

Since $C' = C \wedge S = Id \wedge t = g_c$, we can choose γ to be T.

$\boxed{N \equiv x:}$

Since $C' = C \wedge S = Id \wedge t = A(x)$, we can choose γ to be T.

$\boxed{N \equiv \lambda x.M:}$

By the definition of typing, $\exists r_1, r_2$ such that

$$C_*, A_*; x : r_1 \vdash M : r_2 \wedge C_* \Vdash \{r_1 \to r_2 \rhd t_*\}$$

Let $T' = T[\alpha \leftarrow r_1]$ such that $(A_*; x : r_1)|_{FV(N)} = (T'(A; x : \alpha))|_{FV(N)}$ and $C_* \Vdash T'(C)$. By hypothesis, $\exists \gamma$ such that:

- $C_* \Vdash \gamma(C')$
- $A_*|_{FV(\lambda x.M)} = \gamma(S(A))|_{FV(\lambda x.M)}$, as $(A_*; x : r_1)|_{FV(M)} = (\gamma(S(A); x : S(\alpha)))|_{FV(M)}$
- $C_* \Vdash \{\gamma(t) \rhd r_2\}$

The last condition to show is that $C_* \Vdash \{\gamma(S(\alpha) \to t) \rhd r_1 \to r_2\}$. Since $C_* \Vdash \{\gamma(t) \rhd r_2\}$, it is enough to show $C_* \Vdash \{r_1 \rhd \gamma(S(\alpha))\}$. If $x \in FV(M)$ then we are done, as $(A_*; x : r_1)|_{FV(M)} = (\gamma(S(A); x : S(\alpha)))|_{FV(M)}$. Otherwise α does not occur in C', S, and t and we can choose γ to be $\gamma[\alpha \leftarrow r_1]$ without affecting the other conditions.

$\boxed{N \equiv M_1 M_2:}$

By the definition of typing, $\exists r_1, r_2$ such that:

- $C_*, A_* \vdash M_1 : r_1 \to r_2$
- $C_*, A_* \vdash M_2 : r_1$
- $C_* \Vdash \{r_2 \rhd t_*\}$

By hypothesis and **FACT** $\exists \gamma_1$:

(1) $A_*|_{FV(M_1 M_2)} = (\gamma_1(S_1(A)))|_{FV(M_1 M_2)}$

(2) $C_* \Vdash \gamma_1(C_1)$

(3) $C_* \Vdash \{\gamma_1(t_1) \rhd r_1 \to r_2\}$

Similarly, applying hypothesis to the typing corresponding to M_2 and by **FACT**, $\exists \gamma_2$ such that:

$(1)'$ $A_*|_{FV(M_1 M_2)} = (\gamma_2(S_2(S_1(A))))|_{FV(M_1 M_2)}$

$(2)'$ $C_* \Vdash \gamma_2(C_2) \Vdash \gamma_2(S_2(C_1))$ (Since $C_2 \Vdash S_2(C_1)$)

$(3)'$ $C_* \Vdash \{\gamma_2(t_2) \rhd r_1\}$

By **FACT**, S_2 can only act on type variables in t_1 that occur in $S_1(A)$. By (1) and $(1)'$, for such variables $\gamma_2 \circ S_2$ must act exactly as γ_1 does. Therefore we can rewrite (3) to:

$(3)''$ $C_* \Vdash \{\gamma_2(S_2(t_1)) \rhd r_1 \to r_2\}$

Since γ_2 instantiates $S_2(t_1)$ to an "arrow" type, by the property of UNIFY $\exists \gamma_3$ such that:

(a) $\gamma_2 = \gamma_3 \circ R$

The following entailments are consequence of (a), $(3)'$, and $(3)''$:

(b) $C_* \Vdash \{\gamma_3(R(t_2) \rhd R(\alpha))\}$

(c) $C_* \Vdash \{\gamma_3(R(\beta)) \rhd r_2\}$

Finally, by (b) and the property of MATCH $\exists \gamma$ such that:

(*d*) $\gamma_3 = \gamma \circ R'$

From the arguments above we conclude that:

- $A_*|_{FV(M_1 M_2)} = \gamma R' R S_2 S_1(A)|_{FV(M_1 M_2)}$ (By (1)', (*a*), and (*d*))
- $C_* \Vdash \{\gamma R' R(\beta) \rhd t_*\}$ (By (*c*), (*d*), and $C_* \Vdash \{r_2 \rhd t_*\}$)
- $C_* \Vdash \gamma(C')$ (By (2)', (*a*), (*b*), and (*d*))

□

There are two comments to our completeness result. First, the theorem is still true if we change condition $A_*|_{FV(N)} = T(A)|_{FV(N)}$ in the theorem to $C_* \Vdash A_*|_{FV(N)} \rhd T(A)|_{FV(N)}$. Secdondly, in the proof, instead of showing $C_* \Vdash A_*|_{FV(N)} \rhd \gamma(S(A))|_{FV(N)}$ we show a stronger result: $A_*|_{FV(N)}$ is actually equal to $\gamma(S(A))|_{FV(N)}$ which does not fully use the flexibility of the definition for *instance*. This is no longer true after we minimize the principal typings computed by $TYPE$.