

THE UNIVERSITY OF CHICAGO

A REWRITING SEMANTICS FOR TYPE INFERENCE

A DISSERTATION SUBMITTED TO  
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES  
IN CANDIDACY FOR THE DEGREE OF  
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY  
GEORGE KUAN

CHICAGO, ILLINOIS

MARCH 2007

## **ABSTRACT**

When students first learn programming, they often rely on a simple operational model of a program's behavior to explain how particular features work. Because such models build on their earlier training in algebra, students find them intuitive, even obvious. Students learning type systems, however, have to confront an entirely different notation with a different semantics that many find difficult to understand.

In this work, I begin to build the theoretical underpinnings for treating typechecking in a manner like the operational semantics of execution. Intuitively, each term is incrementally rewritten to its type. For example, each basic constant rewrites directly to its type and each lambda expression rewrites to an arrow type whose domain is the type of the lambda's formal parameter and whose range is the body of the lambda expression which, in turn, rewrites to the range type.

# CHAPTER 1

## INTRODUCTION

This paper represents our first steps in exploring a completely different way to think about the typechecking process. Instead of visualizing typechecking as the process of constructing a proof-tree, we explore typechecking as rewriting, in the spirit of Felleisen-Hieb [13].

We demonstrate our technique in the context of three different type systems: the simply typed lambda calculus (section 2), Curry/Hindley-style type inference (section 3), and Hindley/Milner-style let polymorphism (section 4). Along the way, we prove that our reformulations of the type systems have the same power as the existing ones. I also fill a gap in the literature, proving that the binding-depth numbering scheme used in the SML/NJ compiler [1] (which is similar to the one used in the Caml implementation [29]) is equivalent to Algorithm  $\mathcal{W}$ .

In addition to using the proofs in this paper to validate these systems, all of the rewriting systems have been implemented in PLT Redex [22] and have been carefully tested (except the system in figure 3.2, because it is not feasibly executable). They are available for download at <http://www.cs.uchicago.edu/~gkuan/rwsemtypes/>. To keep the systems in this paper as close to our PLT Redex implementations as possible, we use a Scheme-like syntax for expressions and types. In particular, arrow types are written in prefix parenthesized form and the variables bound by  $\lambda$  expressions are surrounded by parenthesis, rather than suffixed with a dot.

## CHAPTER 2

### SIMPLY TYPED $\lambda$ -CALCULUS

Fig. 2.1 contains the grammar and the traditional presentation of the type system for the simply typed  $\lambda$ -calculus (SLC). Fig. 2.2 contains the rules that define our typechecking relation,  $\mapsto_t$ , which rewrites expressions to their types. The typing context  $T$  dictates that typechecking proceeds from left to right. Numeric constants rewrite to the type num.  $\lambda$ -abstractions rewrite to an arrow type whose domain is the specified type of the parameter and whose range is the body of the original  $\lambda$ -abstraction, but with free occurrences of the parameter variable replaced by its type. Application expressions are rewritten when the function position of an application is an arrow type whose domain matches the type in the argument position. In that case, they rewrite to the range of the function type. I dub this rule  $\tau\beta$  because it is the type-level analogue of the application of a function to an argument. Terms that fail to typecheck get stuck without producing a final type. For example,  $(@ 2 3)$  rewrites to  $(@ \text{num num})$ , which does not match any of the rewrite rules.

Because the  $\mapsto_t$  relation incrementally rewrites a term to a type, intermediate states are hybrid expressions ( $e_h \in \text{SLC-H}$ ) that contain a mixture of SLC and type syntactic forms, and encompass both SLC and type expressions (i.e.,  $\text{SLC} \subseteq \text{SLC-H}$  and  $\text{TYPE} \subseteq \text{SLC-H}$ ). The “H” suffix denotes the hybrid superset of the language specified in the prefix, in case this SLC. To see how such hybrid expressions come about, consider this reduction sequence (where the redexes have been underlined):

$$\begin{aligned}
 & \underline{(\lambda (y (\rightarrow \text{num num})) (\lambda (x \text{num}) (@ y x)))} \\
 \mapsto_t & (\rightarrow (\rightarrow \text{num num}) \underline{(\lambda (x \text{num}) (@ (\rightarrow \text{num num}) x)))} && \text{by [tc-lam]} \\
 \mapsto_t & (\rightarrow (\rightarrow \text{num num}) (\rightarrow \text{num} \underline{(@ (\rightarrow \text{num num}) \text{num}))) && \text{by [tc-lam]} \\
 \mapsto_t & (\rightarrow (\rightarrow \text{num num}) (\rightarrow \text{num num})) && \text{by [tc-}\tau\beta]
 \end{aligned}$$

We start with a  $\lambda$  expression whose parameter,  $y$ , has type  $(\rightarrow \text{num num})$  and whose body is another  $\lambda$  expression whose parameter,  $x$ , has type num. The inner  $\lambda$ 's body is the application of  $y$  to  $x$ . The first reduction step rewrites the outer  $\lambda$ -abstraction into an arrow type

$$\begin{array}{l}
e ::= x \mid (\lambda (x \tau) e) \mid (@ e e) \mid \mathit{number} \\
\tau ::= \mathit{num} \mid (\rightarrow \tau \tau)
\end{array}
\quad \begin{array}{l}
\text{SLC} \\
\text{TYPE}
\end{array}$$

$$\frac{\Gamma \vdash \mathit{number} : \mathit{num}}{\Gamma \vdash \mathit{number} : \mathit{num}} \quad \frac{(x : \tau \in \Gamma)}{\Gamma \vdash x : \tau} [\text{t-var}]$$

$$\frac{\Gamma \vdash e_1 : (\rightarrow \tau_1 \tau_2) \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (@ e_1 e_2) : \tau_2} [\text{t-app}] \quad \frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : (\rightarrow \tau_1 \tau_2)} [\text{t-lam}]$$

Figure 2.1: Grammar and traditional type system for the simply typed  $\lambda$ -calculus

$$\begin{array}{l}
e_h ::= x \mid (\lambda(x \tau) e_h) \mid (@ e_h e_h) \mid \mathit{number} \mid (\rightarrow \tau e_h) \mid \mathit{num} \\
T ::= (@ T e_h) \mid (@ \tau T) \mid (\rightarrow \tau T) \mid \square
\end{array}
\quad \text{SLC-H}$$

$$\begin{array}{ll}
T[\mathit{number}] \mapsto_t T[\mathit{num}] & [\text{tc-num}] \\
T[(\lambda (x \tau) e_h)] \mapsto_t T[(\rightarrow \tau \{x \mapsto \tau\} e_h)] & [\text{tc-lam}] \\
T[(@ (\rightarrow \tau_1 \tau_2) \tau_1)] \mapsto_t T[\tau_2] & [\text{tc-}\tau\beta]
\end{array}$$

Figure 2.2: Grammar and rewriting type system for simply typed  $\lambda$ -calculus (TC)

whose domain is  $(\rightarrow \mathit{num} \mathit{num})$  and whose range is the body of the original  $\lambda$ -abstraction but with all the occurrences of  $y$  replaced by  $(\rightarrow \mathit{num} \mathit{num})$ , producing a hybrid term. The next step is to rewrite the remaining  $\lambda$  expression, this time replacing  $x$  with  $\mathit{num}$ . The final step is a  $\tau\beta$  step. It replaces the application expression with  $\mathit{num}$ , because the function position's domain matches the argument position.

**Theorem 1 (Soundness and Completeness for  $\mapsto_t$ ).**

For any  $e$  and  $\tau$ ,  $\emptyset \vdash e : \tau \Leftrightarrow e \mapsto_t^* \tau$ .

*Proof.* [sketch<sup>1</sup>] From left to right, the proof is a straightforward induction on the derivation of  $\emptyset \vdash e : \tau$ . From right to left, I must first construct the CEK machine analogue of the reduction system [10, 12], making the context search and program variable to type substitutions explicit. This transformation makes it possible to correlate the structure of the typing derivation tree and the structure of the reduction sequence.  $\square$

1. All of the proofs in this paper have been carried out in detail in the appendices.

$$\begin{aligned}
E &::= (@ E e) \mid (@ v E) \mid \square & E[(@ (\lambda (x \tau) e) v)] &\mapsto_e E[\{x \mapsto v\}e] & [\text{ev-}\beta_v] \\
v &::= (\lambda (x \tau) e) \mid \text{number}
\end{aligned}$$

Figure 2.3: Evaluation Rewriting Semantics

Although Theorem 1 guarantees that the typechecker is sensible, I might wish to relate it directly to evaluation, bypassing the traditional type system. Fig. 2.3 gives the standard evaluation contexts and rewrite rules for call-by-value SLC.

A first cut at a direct statement of type preservation for our rewriting type system is to simply take the union of the the evaluation relation  $\mapsto_e$  and the typechecking relation  $\mapsto_t$ , and then prove that it is confluent, *i.e.* each intermediate step in the evaluation sequence reduces to the same type.

**Definition 1 (Combined rewrite relation  $\mapsto$ ).**  $\mapsto = \mapsto_e \cup \mapsto_t$ .

For an example of  $\mapsto$ , see Fig. 2.4. The upper left contains the application of the identity function to 42. It can rewrite two different ways. Along the top of the diagram, typechecking rules apply, eventually reducing to the type num. Moving down from the original term, the rule for function application applies, producing 42, which also typechecks to num.

Unfortunately, the union of the relations is not confluent in general. Consider the example in Fig. 2.5. It is an ill-typed term, but after a single application becomes well-typed. Accordingly, the typechecking rewrite rules detect the error in the original term, but produce type num for the term after  $\beta$ -reduction eliminates the subexpression containing the type error.

**Theorem 2 (Non-confluence of  $\mapsto$ ).** *There exists an expression  $e$ , such that  $e \mapsto e_h$ ,  $e \mapsto e'_h$ ,  $e_h \neq e'_h$ , and both  $e_h$  and  $e'_h$  are either types or stuck under  $\mapsto$ .*

*Proof.* The expression  $(@ (\lambda (x \text{ num}) 42) (\lambda (y \text{ num}) (@ 1 1)))$  rewrites to both num and an expression that decomposes into a typechecking context with the stuck state  $(@ \text{ num num})$  in the hole.  $\square$

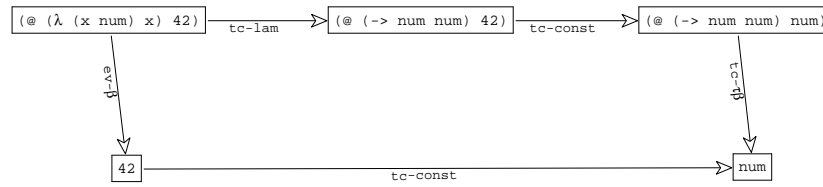


Figure 2.4: Confluent examples for combined rewrite relation

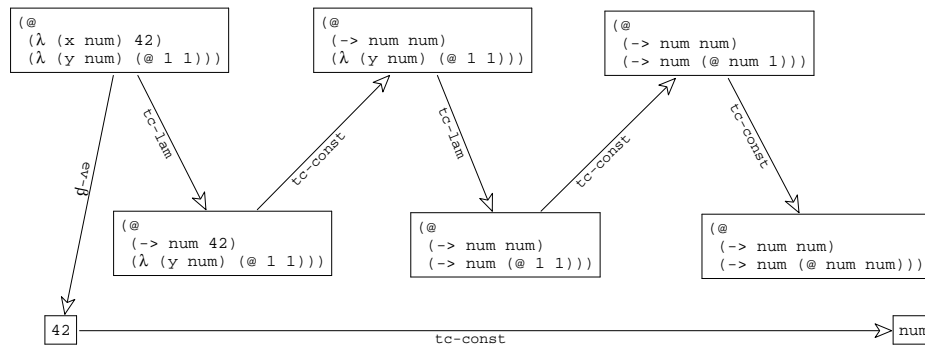


Figure 2.5: Non-confluent counterexample for combined rewrite relation

Nevertheless, I do know that  $\mapsto$  is confluent for any term that is well-typed, thus implying a preservation theorem.

**Theorem 3 (Preservation).** *If  $e \mapsto_t^* \tau$  &  $e \mapsto_e e'$ , then  $e' \mapsto_t^* \tau$*

*Proof (sketch).* This follows from the observation that, once a term takes a type checking step, it can never again take an evaluation step. That, plus Theorem 1 and a standard type preservation argument for the traditional type system tells us that the relation is confluent when the original term is well-typed, and thus the theorem holds.  $\square$

## CHAPTER 3

### CURRY/HINDLEY TYPE INFERENCE

A conceptually simple way to extend the rewrite system from section 2 to handle type inference is to erase the type annotation on the bound parameter, yielding the untyped  $\lambda$ -calculus (ULC), and re-interpret the [tc-lam] rule, allowing it to rewrite the bound variable to any type. To see how this plays out, consider the example in Fig. 3.1. It begins with the application of the identity function to 5, which decomposes into a typechecking context with  $(\lambda (x) x)$  in the hole. Since there is no longer any constraint on the bound variable, the  $\lambda$ -expression rewrites to every arrow type whose domain and range are the same. Although all but one of these choices are ultimately doomed, they can still each rewrite at least one more step, replacing 5 with num. At this point, the application rule only applies to the term where the type chosen for  $x$  was num so the top-most sequence in the figure rewrites to num. All of the rest of the choices get stuck.

Accordingly, I must also refine the notion that an expression has a type to say that an expression has a type if there exists some reduction sequence from that expression to that type. This intuition is turned into a formal system in Fig. 3.2. The [nd-lam] rule has a  $\tau$  that only appears on the right-hand side of the rule, indicating that it can be instantiated to any type. The  $n$  subscript on the  $\mapsto_n$  relation indicates that the relation models the nondeterministic choice of the type of the function parameter.

I can relate the nondeterministic system with the original one via the function  $\mathcal{E}$ , that maps SLC-H to ULC-H by erasing the type annotations in  $\lambda$ -expressions. In particular, the nondeterministic choice does not keep us from typechecking terms that typechecked before:

**Theorem 1 (Completeness of nondeterministic reduction).** *For any SLC expression  $e$  and type  $\tau$ ,*

$$e \mapsto_t^* \tau \Rightarrow \mathcal{E}(e) \mapsto_n^* \tau$$

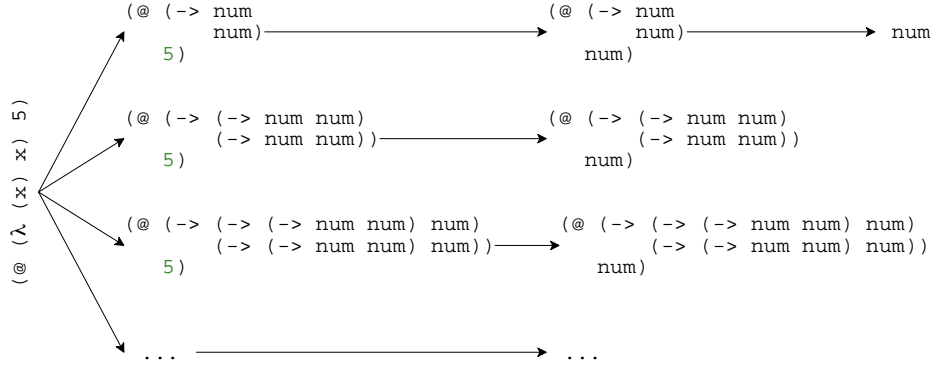


Figure 3.1: Rewriting nondeterministically

---

$e ::= x \mid (\lambda (x) e) \mid (@ e e)$	ULC
$e_n ::= x \mid (\lambda (x) e_n) \mid (@ e_n e_n) \mid \text{number} \mid (\rightarrow \tau e_n) \mid \text{num}$	ULC-H
$T_n ::= (@ T_n e_n) \mid (@ \tau T_n) \mid (\rightarrow \tau T_n) \mid \square$	CTX
$\tau ::= \text{num} \mid (\rightarrow \tau \tau)$	TYPE
$T_n[\text{number}] \mapsto_n T_n[\text{num}]$	[nd-num]
$T_n[(\lambda (x) e_n)] \mapsto_n T_n[(\rightarrow \tau \{x \mapsto \tau\} e_n)]$	[nd-lam]
$T_n[(@ (\rightarrow \tau_1 \tau_2) \tau_1)] \mapsto_n T_n[\tau_2]$	[nd- $\tau\beta$ ]

---

Figure 3.2: Grammar and rewrite rules for nondeterministic (ND) inference calculus

*Proof (sketch).* Simply erasing all of the types on the parameters in the reduction sequence for  $\mapsto_t$  produces a valid reduction sequence for  $\mapsto_n$  with the desired properties.  $\square$

But the implication in the reverse direction does not hold. In particular, the erasure of the term  $(@ (\lambda (x (\rightarrow \text{num num})) x) 1)$  has type  $\text{num}$ , even though the term itself does not. Still, it is possible to restore types to any erased term that has a type, in order to produce a typeable term, as the following theorem shows.

**Theorem 2 (Soundness of nondeterministic reduction).** *For any ULC expression  $e$  and type  $\tau$ , if  $e \mapsto_n^* \tau$  then there exists a SLC expression  $e'$  such that  $\mathcal{E}(e') = e$  and  $e' \mapsto_t^* \tau$*

*Proof (sketch).* This proof goes through by induction, once the inductive hypothesis is strengthened to allow for an arbitrary variable to type substitution to be applied to  $e$ .  $\square$

$p ::= (\mathbf{unify} \tau_u \tau_u p) \mid e_u$	Unify problems
$e_u ::= x \mid (\lambda (x) e_u) \mid (@ e_u e_u) \mid number \mid (\rightarrow \tau_u e_u) \mid num \mid \xi$	ULC-HV
$T_u ::= (@ T_u e_u) \mid (@ \tau_u T_u) \mid (\rightarrow \tau_u T_u) \mid \square$	CTX-V
$\tau_u ::= num \mid (\rightarrow \tau_u \tau_u) \mid \xi$	TYPE-V
$\xi, \zeta ::= \text{type variables}$	TVAR
$T_u[number] \mapsto_u T_u[num]$	[ch-num]
$T_u[(\lambda (x) e_u)] \mapsto_u T_u[(\rightarrow \xi \{x \mapsto \xi\} e_u)]$ $\xi \text{ fresh}$	[ch-lam]
$T_u[(@ \tau_{u_1} \tau_{u_2})] \mapsto_u (\mathbf{unify} (\rightarrow \tau_{u_2} \xi) \tau_{u_1} T_u[\xi])$ $\xi \text{ fresh}$	[ch- $\tau\beta$ ]
$(\mathbf{unify} \tau_{u_1} \tau_{u_1} p) \mapsto_u p$	[ch-u-eq]
$(\mathbf{unify} (\rightarrow \tau_{u_1} \tau_{u_2}) (\rightarrow \tau_{u_3} \tau_{u_4}) p) \mapsto_u (\mathbf{unify} \tau_{u_1} \tau_{u_3} (\mathbf{unify} \tau_{u_2} \tau_{u_4} p))$ $(\rightarrow \tau_{u_1} \tau_{u_2}) \neq (\rightarrow \tau_{u_3} \tau_{u_4})$	[ch-u-dist]
$(\mathbf{unify} \tau_u \xi p) \mapsto_u (\mathbf{unify} \xi \tau_u p)$ $\tau_u \neq \xi$	[ch-u-orient]
$(\mathbf{unify} \xi \tau_u p) \mapsto_u \{\xi \mapsto \tau_u\}p$ $\xi \notin \text{ftv}(\tau_u)$	[ch-u-inst]

$\text{ftv}(\tau) = \text{set of type variables occurring in } \tau.$

Figure 3.3: Grammar and rewrite rules for Curry/Hindley (CH) calculus

A direct implementation of this system is not feasible, for two reasons. First, it would require searching an infinitely large space and second, it would not produce a single best answer. For example, the expression  $(\lambda (x) x)$  reduces to an infinite number of types, namely all function types whose domain and range are the same. The standard approach (due to Curry and Hindley [6, 16]) to coping with this problem is to use unification, and so we add unification to my model, as shown in Fig. 3.3.

The language in Fig. 3.3 is the same as the one in Fig. 3.2, except that expressions may be wrapped with **unify** and types may be type variables ( $\xi$ ). This system uses type variables and **unify** to enforce constraints between types whereas the nondeterministic system guesses types. In particular, a  $\lambda$ -expression now reduces to an arrow type whose domain is fresh type variable. Similarly, an application of a type to another type reduces to a new type variable after wrapping the entire expression with a **unify** expression that ensures that the function type on the left hand side of the application matches the argument type on the right hand side.

Since the context where typechecking reductions occur does not contain **unify**, the

**unify**s must be reduced before any other reductions occur. The last four reductions in Fig. 3.3 cover the reductions of the **unify** expressions<sup>1</sup>. The first removes the unification of two identical types. The second distributes the unification of two different arrow types to the unification of their domains and their ranges. The third reduction orients the **unify** reduction; if the second argument to **unify** is a type variable and the first is not, the reduction swaps the arguments. The final reduction performs a unification of a type variable and another type not containing that type variable (checked by the occurrence check) by substituting the type for that type variable.

Unlike the  $\mapsto_n$  relation, the  $\mapsto_u$  relation is deterministic. For example, this is the reduction sequence for the example from Fig. 3.1:

$$\begin{aligned}
 & (@ (\lambda (x) x) 5) \\
 \mapsto_u & (@ (\rightarrow \xi \xi) 5) \\
 \mapsto_u & (@ (\rightarrow \xi \xi) \text{num}) \\
 \mapsto_u & (\mathbf{unify} (\rightarrow \xi \xi) (\rightarrow \text{num } \xi') \xi') \\
 \mapsto_u^* & \text{num}
 \end{aligned}$$

The first step generates a fresh type variable and replaces the  $\lambda$ -expression with an arrow type whose domain and range are that type variable. As in Fig. 3.1, the next step replaces 5 with num. The next step generates the unification problem that ultimately results in the type num as the final answer.

Where the first step in the  $\mapsto_n$  relation generated an infinite number of next states, the  $\mapsto_u$  relation generates a schematic expression that represents all of those states. Throughout the course of a complete ND reduction sequence, I may encounter a number of these reductions that generate multiple next states. For any particular combination of choices of next states, I can construct a ground type substitution  $\gamma: \text{TVAR} \rightarrow \text{TYPE}$  that instantiates the type variables in the CH reduction sequence to those types chosen in the ND reduction sequence. I can exploit this correspondence to make the relationship between the two type-checking relations precise, via the ground type substitution  $\gamma$  for a complete ND reduction sequence  $e \mapsto_n^* \tau$ .

---

1. Martelli and Montanari introduced a rewriting method for performing unification[21]. My unification system is related to theirs. Instead of explicitly transforming equation sets, I work on **unify** problems, each of which represents one equation.

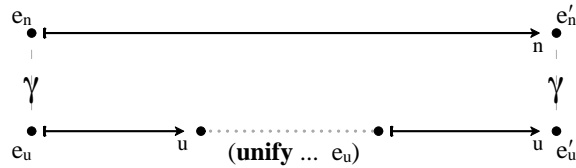
**Theorem 3 (Nondeterministic & Curry/Hindley typing relationship).**

Let  $e \in \text{ULC}$ .

**Completeness** If  $e \mapsto_n^* \tau$  then there exists a  $\tau_u$  and a type substitution  $\gamma$  such that  $e \mapsto_u^* \tau_u$  and  $\gamma\tau_u = \tau$ .

**Soundness** If  $e \mapsto_u^* \tau_u$  then for all ground types  $\tau$  that are instantiations of  $\tau_u$ ,  $e \mapsto_n^* \tau$ .

*Proof (sketch).* This proof hinges on the observation that the structure of the complete reduction sequences in  $\mapsto_n$  and  $\mapsto_u$  are related by a ground type substitution  $\gamma$  for the complete ND reduction sequence, as shown in this diagram:



Most of the work in this proof is verifying the conditions of the above diagram. To prove these conditions, I need to build the ground type substitution.

For the completeness part of the theorem, only [nd-lam] reductions in the ND reduction sequence produce terms that instantiate type variables in the corresponding CH terms. Each [nd-lam] step in the complete ND reduction sequence replaces a bound variable  $x_i$  with a parameter type  $\tau_i$ . For each [nd-lam] step, associate a fresh type variable  $\xi_i$ . The ground type substitution maps each  $\xi_i$  to  $\tau_i$ . The [ch-lam] steps should use  $\xi_i$  for the fresh type variable when reducing the  $\lambda$ -binder for  $x_i$ .

For the soundness part, the ground type substitution is the composition of two substitutions. First, I need the composition of all the unification substitutions that are produced by performing the unifications introduced by the [ch- $\tau\beta$ ] reduction. The composition of all the unification substitutions is certainly a solution for any of the individual unification problems because solved type variables are eliminated and will never be reintroduced in the CH reduction sequence. Furthermore, I need a substitution that instantiates all the residual unconstrained type variables to arbitrary types. In a complete ND reduction sequence, all [nd-lam] reductions must instantiate with a type that is an instance of the final unification solution for the type variable  $\xi$  introduced by the corresponding [ch-lam] reduction.

Three other essential facts must be established. First, I need to establish that ground type substitutions distribute over a CH type checking context decomposition to yield an ND context decomposition, i.e.,  $\gamma(T_u[e_u]) = (\gamma T_u)[\gamma e_u]$  and  $\gamma T_u$  is a  $T_n$  and  $\gamma e_u$  is an  $e_n$ . Second, that the composition of the most general unifiers  $\sigma$  and  $\sigma'$  of types  $\tau_1$  and  $\tau_2$ , and  $\tau'_1$  and  $\tau'_2$  respectively is a most general unifier of any type constructed using those four types (due to Robinson [30]) and finally that the **unify** reductions perform unifications consistent with a most general unifier.  $\square$

## CHAPTER 4

### HINDLEY/MILNER INFERENCE

As a practical matter, it is important to add a **let**-form to my language so that programmers can bind a single value and use it with multiple types. A **let** expression has the form  $(\mathbf{let} (x e_1) e_2)$ , where  $x$  is a program variable,  $e_1$  is the definiens, whose value is bound to  $x$  in  $e_2$ , the body. The meaning of **let** expressions is the same as an application of an explicit  $\lambda$  expression:  $((\lambda (x) e_2) e_1)$ .

Type checking a **let** expression by replacing it with such an application, however, yields a typechecker that rejects too many programs. In particular, imagine that the expression bound to the variable is the identity function and that the body of the **let** expression uses the identity function on both booleans and integers. Rewriting the **let** expression as above would produce a program that does not typecheck, even though the original program certainly is safe.

A naive typechecker could overcome this problem by rewriting **let** expressions via substitution, replacing the each free occurrence of the **let**-bound variable by the definiens (i.e.  $\beta$ -reducing the redex that the let expression abbreviates). Then each resulting occurrence of the argument expression could be typechecked separately in its own context within the body, allowing the typechecker to infer different types for different uses of the bound variable.

Unfortunately, such a scheme involves redundant work in the type checker and possibly duplicate type error messages. To avoid this redundancy, Milner developed a typechecking algorithm [4, 8, 9, 25] that achieves the same result as the substitution by splitting the typechecking of the definiens into two phases: first determining a generic type that is independent of the context of use, and then for each use of the defined variable determining an instance of the generic type that fits its context. It does this by first typechecking the definiens in the context outside of the whole **let** expression (not including the **let** expression), and then partitioning the unconstrained type variables in the result into two sets:

$e_p ::= x \mid (\lambda^d (x) e_p) \mid (@ e_p e_p) \mid \text{number} \mid (\mathbf{let}^d (x e_p) e_p)$	ULCL-A
$T_p ::= (\mathbf{let}^d (x T_p) e_p) \mid (@ T_p e_p) \mid (@ \tau T_p) \mid (\rightarrow \tau T_p) \mid \square$	CTXL-VR
$p ::= (\mathbf{unify} \tau \tau_p) \mid e_p$	ULCL-HRU
$\tau ::= \text{num} \mid \xi^d \mid (\rightarrow \tau \tau)$	TYPE-VR
$\tau_p ::= \text{num} \mid \xi^d \mid (\rightarrow \tau_p \tau_p) \mid \alpha$	POLYTYPE
$\rho ::= \forall \vec{\alpha} (\{\vec{\xi} \mapsto \vec{\alpha}\} \tau_p)$	Bound Type
$c ::= e_p \mid p \mid \tau \mid \rho$	Machine Control
$m, d ::= 0 \mid 1 \mid \dots \mid \infty$	

Figure 4.1: Grammar for Hindley/Milner calculus

*polymorphic* variables that can be safely instantiated to different types at each occurrence of the **let**-bound variable, and those that cannot because they are constrained by the outer context. The result is represented as a polymorphic type  $\forall \vec{\alpha}. \tau$ , where the type variables in  $\vec{\alpha}$  are the polymorphic, or generalizable, variables.

If I try to modify the Curry/Hindley rewriting system to generalize types at **let** bindings, the problem is that the context of outer bound variables will already have been eliminated by the [ch-lam] rule, making it difficult to calculate generalizability of type variables. An alternative approach to determining generalizability is based on an idea originally suggested by Damas [7] in the early 1980s and refined and used in compilers like SML/NJ in the mid 1980s. The idea is to assign a binding depth or *rank* to type variables that reflects the level of the outermost variable binding they are associated with. Substitution for a ranked type variable must preserve a *bounded rank property*, namely, that the ranks of type variables in the term substituted cannot exceed the rank of the type variable being substituted for. The invariant is that if a type variable  $\xi^d$  has rank  $d$ , it occurs in the type of a lambda binding at nesting depth  $d$ , but in no shallower binding. If a type  $\tau$  is substituted for  $\xi^d$ , then its type variables now also appear in the type of this  $d$  level binding, and they should also have rank  $d$ , or possibly lower if they also appear in shallower bindings. Thus substitution for a variable of rank  $d$  entails globally resetting depths of type variables found the substituted type to have rank at most  $d$ , reflecting their new binding depth. Now the test for whether a type variable appears in the binding context of an expression being typed reduces to comparing its rank with the current binding depth – if its rank is greater than the current binding depth,

$E[(\lambda (x) e_p) v] \mapsto_e E[\{x \mapsto v\}e_p]$	[ev-beta]
$E[(\mathbf{let}^d (x e_{p1}) e_{p2})] \mapsto_e E[\{x \mapsto e_{p1}\}e_{p2}]$	[ev-let]
$T_p[\mathit{number}] \mapsto_p T_p[\mathit{num}]$	[tcp-num]
$T_p[(\lambda^d (x) e_p)] \mapsto_p T_p[(\rightarrow \xi^d (\{x \mapsto \xi^d\}e_p))]$ $\xi^d$ fresh	[tcp-lam]
$T_p[(\lambda (\tau_1 \tau_2))] \mapsto_p (\mathbf{unify} \tau_1 (\rightarrow \tau_2 \xi^\infty) T_p[\xi^\infty])$ $\xi^\infty$ fresh	[tcp- $\tau\beta$ ]
$T_p[(\mathbf{let}^d (x \tau) e_p)] \mapsto_p T_p[\{x \mapsto (\forall \vec{\alpha} \tau_p)\} e_p]$ $\vec{\alpha}$ fresh and $\tau_p = \{\mathcal{G}(\tau, d) \mapsto \vec{\alpha}\} \tau$	[tcp-let]
$T_p[(\forall \vec{\alpha} \tau_p)] \mapsto_p T_p[\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p]$ $\vec{\xi}^\infty$ fresh	[tcp-poly]
$(\mathbf{unify} \xi^d \tau_p) \mapsto_p (\mathcal{L}(\tau, d))(\{\xi^d \mapsto \tau\}p)$ $\xi^d \notin \text{ftv}(\tau)$	[tcp-u-inst]

The other Curry-Hindley rewrite rules, [ch-u-eq], [ch-u-dist],  
and [ch-u-orient] carry over with the  $u$  subscripts replaced by  
 $p$ . Call them [tcp-u-eq], [tcp-u-dist], etc. respectively.

$$\begin{aligned} \mathcal{G} &: \tau \times \text{depth} \rightarrow \xi \text{ list} \\ \mathcal{G}(\tau, d) &= \{\xi^m \in \text{ftv}(\tau) \mid d < m\} \\ \mathcal{L} &: \tau \times \text{depth} \rightarrow \xi \text{ depth substitution} \\ \mathcal{L}(\tau, d) &= \{\xi^m \mapsto \xi^d \mid \xi^m \in \text{ftv}(\tau) \text{ and } d < m\} \end{aligned}$$

Figure 4.2: Rewrite rules for Hindley/Milner Type Inference

then it does not appear in the context and thus can be considered polymorphic.<sup>1</sup> The fresh type variables used to generically instantiate the polymorphic type of a **let**-bound variable occurrence start with rank  $\infty$ , reflecting the fact that they initially are not free in the type of any lambda-bound variables.

Our rewriting system for Hindley/Milner inference is presented in Figs. 4.1 and 4.2. I first pre-label the binding constructs of the expression being typed with their lambda binding depths. For instance, here is an example of a term  $e$  and its depth-labeled version.

$$\begin{aligned} &(\lambda (x) (\lambda (y) (\mathbf{let} (z (\lambda (u) y)) (@ (@ z x) (@ z 1)))))) \\ &(\lambda^1 (x) (\lambda^2 (y) (\mathbf{let}^2 (z (\lambda^3 (u) y)) (@ (@ z x) (@ z 1)))))) \end{aligned}$$

---

1. Some rank systems, like the one described here and the one used in the SML/NJ typechecker, are based on nesting levels of lambda bindings. Other closely related systems, such Rémy's [29] and McAllester's [24], are based on nesting levels of **let** bindings.

The rewriting system operates on such labeled expressions from the  $e_p$  grammar. When a type variable  $\xi^d$  is generated by applying the [tcp-lam] rule to an expression  $(\lambda^d (x) e)$ , it is initially assigned the depth  $d$  of its  $\lambda$ -binding to indicate that it is associated with this depth  $d$  binder. The label  $d$  is a positional indicator that supports a short-cut method of determining the binding scope of a type variable. In the above example,  $x$ ,  $y$ , and  $u$  will be assigned type variables  $\xi_x^1$ ,  $\xi_y^2$ , and  $\xi_u^3$  respectively. Fresh type variables used to create a generic instance of a polytype in rule [tcp-poly] are given a depth of  $\infty$ , since they are (initially) not associated with any lambda-bound variable. For example, the polymorphic type of the identity function is  $(\forall \alpha (\rightarrow \alpha \alpha))$ , but [tcp-poly] will reduce the type to  $(\rightarrow \xi^\infty \xi^\infty)$ . The unification rule [tcp-u-inst] can instantiate a type variable to a type  $\tau$  that may contain other type variables. This rule enforces the maximal rank property discussed above by applying a depth-adjustment substitution  $\mathcal{L}(\tau, d)$ . The substitution acts on the full expression  $p$  to ensure that the adjustment is performed globally on all occurrences of the affected type variables.

As an example of how the system operates, consider the labeled expression

$$(\lambda^1 (x) (\mathbf{let}^1 (f (\lambda^2 (y) (@ x y))) (@ f 5)))$$

The type rewriting of this expression proceeds as follows:

$$(\lambda^1 (x) (\mathbf{let}^1 (f (\lambda^2 (y) (@ x y))) (@ f 5))) \tag{4.1}$$

$$\mapsto_p^* (\rightarrow \xi_x^1 (\mathbf{let}^1 (f (\rightarrow \xi_y^2 (@ \xi_x^1 \xi_y^2))) (@ f 5))) \tag{4.2}$$

$$\mapsto_p (\mathbf{unify} \xi_x^1 (\rightarrow \xi_y^2 \xi_3^\infty) (\rightarrow \xi_x^1 (\mathbf{let}^1 (f (\rightarrow \xi_y^2 \xi_3^\infty))) (@ f 5))) \tag{4.3}$$

$$\mapsto_p (\rightarrow (\rightarrow \xi_y^1 \xi_3^1) (\mathbf{let}^1 (f (\rightarrow \xi_y^1 \xi_3^1))) (@ f 5))) \tag{4.4}$$

$$\mapsto_p (\rightarrow (\rightarrow \xi_y^1 \xi_3^1) (@ (\rightarrow \xi_y^1 \xi_3^1) 5))) \tag{4.5}$$

$$\mapsto_p^* (\rightarrow (\rightarrow \text{num } \xi_4^1 \xi_4^1)) \tag{4.6}$$

The expression at line (2) is obtained by two applications of [tcp-lam] to rewrite the  $\lambda x$  and  $\lambda y$  binders, introducing the rank 1 type variable  $\xi_x^1$  and the rank 2 type variable  $\xi_y^2$ . At line (3), the application in the definiens of  $f$  is rewritten using [tcp- $\tau\beta$ ], introducing the fresh type variable  $\xi_3^\infty$  to represent the type of the result of the application and adding a **unify** prefix. Rewriting this with rule [tcp-u-inst] produces line (4), where the substitution

for  $\xi_y^1$  is accompanied by the reduction of the ranks of  $\xi_y^2$  and  $\xi_3^\infty$  to 1. At line (5), the rule [tcp-let] has been used to rewrite the **let**-expression. Because the **let** is at depth 1, and all the type variables in the rewritten definiens are rank 1, the set of generalizable variables  $\mathcal{G}((\rightarrow \xi_y^1 \xi_3^1), 1)$  is  $\emptyset$ , so in this case no polymorphism is introduced and  $f$  is replaced by the nonpolymorphic type  $(\rightarrow \xi_y^1 \xi_3^1)$ . In this example the lack of polymorphism is due to the occurrence of  $x$ , which is bound in an outer scope, in the body of the definition of  $f$ . If on the other hand the definition of  $f$  had been  $(\lambda 2 (y) y)$ , then the definiens would have rewritten to  $(\rightarrow \xi_y^2 \xi_y^2)$  and [tcp-let] would have generalized this to  $(\forall \alpha (\rightarrow \alpha \alpha))$ .

I prove the correctness of this typing rewrite system for let-polymorphism, which I will call HM, by showing that it is equivalent to a slightly modernized variant [18] of Milner's Algorithm  $\mathcal{W}$  [25]. I assume this Algorithm  $\mathcal{W}$  defines a function  $\mathcal{W}(\Gamma, e_p)$ , where  $\Gamma$  is a type assignment mapping variables to types, returning a pair  $(\theta, \tau)$ , where  $\theta$  is a type substitution mapping type variables to types (either monomorphic or polymorphic).  $\mathcal{W}$  has the property that:

$$\mathcal{W}(\Gamma, e_p) = (\theta, \tau) \implies \theta(\Gamma) \vdash e_p : \tau$$

**Theorem 1 (HM Rewrite Soundness and Completeness relative to  $\mathcal{W}$ ).**

*For any closed ULCL-A expression  $e_p$ ,  $e_p \mapsto_p^* \tau$  iff  $\mathcal{W}(\emptyset, e_p) = (\theta, \tau)$ .*

*Proof (sketch).* To prove the theorem, as in Section 2, we use an abstract stack machine. The machine serves to make the type substitutions and the typing environment explicit, allowing us to prove that both Algorithm  $\mathcal{W}$  and the rewriting system in Fig. 4.2 are equivalent to the machine and thus equivalent to each other.

In this case the machine is an analogue of a CEK machine, augmented with a type variable substitution register. Each machine state is of the form  $(c, \Gamma, \Sigma, K)$  where  $c$  is the control (C),  $\Gamma$  is an environment mapping program variables to types (E),  $\Sigma$  (the extra register) is a list of substitutions that map type variables to types, and  $K$  is the typechecking context. The  $\Sigma$  register is used to maintain a correspondance between the machine's states and the substitutions that recursive calls in Algorithm  $\mathcal{W}$  produce. The type checking context is similar to  $T_p$ , but rather than being a context, it is represented as a list of context frames (in some cases augmented with a little extra information).

There are three kinds of rules. The first kind searches for the next reducible expression. For example, this rule

$$((@ e e'), \Gamma, \Sigma, K) \mapsto_{pm} (e, \Gamma, \Sigma, (@ \square e')::K)$$

pushes into the left-hand side of an application. The second kind of rules are analogues of the rules in Fig. 4.2. For example, this rule:

$$((@ \tau_p \tau'_p), \Gamma, \Sigma, K) \mapsto_{pm} ((\mathbf{unify} \tau_p (\rightarrow \tau'_p \xi) \xi), \Gamma, \Sigma, K) \quad \xi \text{ is fresh}$$

is the analogue of the [tcp- $\tau\beta$ ] rule. Finally, the third kind of rule manipulates the environment. For example, this rule:

$$(x, \Gamma, \Sigma, K) \mapsto_{pm} (\Gamma(x), \Gamma, \Sigma, K)$$

looks up a variable in the environment. The  $\Sigma$  register is maintained by the unification rules, and the rules that pop contexts. The complete set of rules are given in the Appendix C.

An important technical element for relating the rewrite system, the abstract machine, and Algorithm  $\mathcal{W}$  is a demonstration that the depth label mechanism correctly models the usual type variable generalization criterion based on type environments or binding prefixes. The proof of this hinges on stating the correct invariant regarding the depth labeling of type variables occurring in the definiens of a **let**-expression and any pending unification problems throughout the process of rewriting of the definiens into its type. The invariant states that if  $\xi^d$  is one of these type variables, and if the depth of the **let** is  $d_0$ , then  $d < d_0$  if and only if  $\xi^d$  appears in the binding prefix derived from the context of the **let**.  $\square$

## CHAPTER 5

### RELATED WORK

Prior approaches to typechecking and inference using rewriting derive type constraints from expressions and then use rewriting to solve these constraint sets. The Stratego/XT program transformation language [3] offers an example of a term rewriting system for typechecking for a simple arithmetic language. Pašalić et al [27] give a graph rewriting system for the original formulation of Hindley/Milner inference without explicit generalization. In contrast, my term rewriting systems operate directly on expressions to transform them into their types.

Type checking via rewriting has a similar feel to abstract interpretation. Each rewrite step takes us a little bit closer to the knowledge of the type of the term, much in the way that abstract interpretation gathers information about the program text. Cousot [5] has formulated type checking as an abstract interpretation; my work has a concrete, operational flavor where his is more denotational. Kahrs [17] has a different formulation of typechecking via abstract interpretations; while his is more operational, like mine, it is based on a translation to a machine-like language; mine operates directly on the program text.

There have been several alternative presentations of type inference algorithms. Wand's algorithm [33] performs a Curry/Hindley style type inference by performing a syntactic traversal of an expression collecting equational constraints and then solving these constraints by unification. Much of the prior work on explaining Algorithm  $\mathcal{W}$  type inference focuses on retaining information from intermediate steps of the process. Soosaipillai [31] maintains a list of the types inferred for each subexpression. Duggan and Bent [11] as well as Wand [32] retain a list of the instantiations of all type variables. This method is similar to Rémy's keeping around a constraint set corresponding to instantiations and unification problems. Pottier [28] presents the current state of the art in constraint-based approaches to type inference. Instead of retaining specific information during inference, I present the entire process in terms of simple rewrite steps. I also apply the substitutions from unification and do not retain them.

There has been significant work done to improve the quality of error messages generated by type systems, especially in languages that have type inference [2, 14, 15, 18, 19, 20, 23, 26, 34, 35]. Like that work, I too am motivated by the desire to make typechecking easier to understand. Generally speaking, that work augments existing typechecking algorithms with more information or improves existing typechecking algorithms in order to improve the error messages produced by the typechecker. My work, in contrast, is an entirely different way to think about the behavior of a typechecker.

## CHAPTER 6

### CONCLUSION

My vision is that this work forms the technical foundation for a more ambitious program to make typecheckers easier to understand. My goal is to lay the groundwork for two related efforts in bringing modern type systems to the ordinary programmer: education and debugging. Because the rewrite-based typecheckers are based directly on the program text and the rewrite rules are relatively straightforward counterparts to evaluation, I believe students can more easily gain an intuition for how a typechecker behaves by studying them. Similarly, I expect to be able to exploit the operational flavor of the typechecking rewrite rules to build debuggers to help more experienced programmers understand why ill-typed programs fail to typecheck.

Although I believe this work succeeds in providing an accessible model for typing the simply typed  $\lambda$ -calculus and for the Curry-Hindley type inference system, the need to resort to the depth-labeling scheme for the Hindley/Milner system leads to a rewriting system that is not as simple and elegant as we would like. Nevertheless, I have managed to produce a correct version of Hindley-Milner polymorphism and to provide, to the best of my knowledge, the first proof that an algorithm based on depth-numbering is equivalent to Algorithm  $\mathcal{W}$ .

Looking to the future, I expect to continue to work on Hindley-Milner and to explore other features of modern type systems and static analyses looking for more opportunities to exploit the operational point of view based on rewriting.

## REFERENCES

- [1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.
- [2] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, March–December 1993.
- [3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT Tutorial, Examples, and Reference Manual for Stratego/XT 0.16*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, November 2005.
- [4] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.
- [5] Patrick Cousot. Types as abstract interpretations. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331, New York, NY, USA, 1997. ACM Press.
- [6] H.B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.
- [7] Luís Damas. unpublished note, 1984.
- [8] Luís Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1985.
- [9] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

- [10] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report BRICS RS-04-26, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 2004. <http://citeseer.ist.psu.edu/danvy04refocusing.html>.
- [11] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.
- [12] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Revision of 1989 edition, 2003.
- [13] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
- [14] Christian Haack and Joseph B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.
- [15] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13, New York, NY, USA, 2003. ACM Press.
- [16] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–40, 1969.
- [17] Stefan Kahrs. Polymorphic type checking by interpretation of code. Technical Report ECS-LFCS-92-238, University of Edinburgh, 1992.
- [18] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
- [19] Benjamin Lerner, Dan Grossman, and Craig Chambers. Seminal: searching for ML type-error messages. In *ML '06: Proceedings of the 2006 Workshop on ML*, pages 63–73, New York, NY, USA, 2006. ACM Press.

- [20] Xavier Leroy. Programmation du système Unix en Caml Light. Technical report 147, INRIA, 1992.
- [21] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
- [22] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Rewriting Techniques and Applications*, 2004.
- [23] Bruce McAdam. How to repair type errors automatically. *Trends in functional programming*, pages 87–98, 2002.
- [24] David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications*, 2003.
- [25] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.
- [26] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 15–26, New York, NY, USA, 2003. ACM Press.
- [27] Emir Pašalić, Jeremy G. Siek, and Walid Taha. Concoction: Mixing indexed types and Hindley-Milner type inference. Unpublished. <http://homepage.mac.com/pasalic/p2/papers/PST06.pdf>, 2006.
- [28] François Pottier. A modern eye on ML type inference: old techniques and recent developments. In *Lecture Notes for APPSEM Summer School*, September 2005.
- [29] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatique, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

- [30] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [31] Helen Soosaipillai. An explanation based polymorphic type checker for Standard ML. Master's Thesis, 1990.
- [32] Mitchell Wand. Finding the source of type errors. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–43, New York, NY, USA, 1986. ACM Press.
- [33] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.
- [34] Jun Yang. Explaining type errors by finding the source of a type conflict. In *SFP '99: Selected papers from the 1st Scottish Functional Programming Workshop (SFP99)*, pages 59–67, Exeter, UK, UK, 2000. Intellect Books.
- [35] Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, 2001.

## APPENDIX A

### TYPECHECKING SYSTEM PROOFS

This section proves the soundness and completeness of the typechecking rewrite system for SLC. The completeness direction is straightforward and will be proved directly. The soundness result requires the formulation of a machine for the rewrite system. I first prove soundness of the machine and then prove that the the rewrite system is sound with respect to the machine.

**Theorem 1 (Soundness and Completeness for  $\mapsto_t$ ).**

*For any  $e$  and  $\tau$ ,  $\emptyset \vdash e : \tau \Leftrightarrow e \mapsto_t^* \tau$ .*

*Proof.* The  $\Leftarrow$  (soundness) direction follows from lemmas 5 and 7.

The  $\Rightarrow$  (completeness) direction follows from lemma 1. □

The proof of the completeness of the rewrite system is simple because the well-typing relation  $\Gamma \vdash e : \tau$  keeps the structure of  $e$  intact (i.e., never changes the SLC expression by substituting in a type for a  $\lambda$ -bound variable) and maintains a type environment throughout the type derivation. Consequently, I have all the information necessary for constructing the corresponding type checking reduction sequence. However, to ensure that the induction goes through, I must strengthen the induction to apply for arbitrary type environment  $\Gamma$ . I define the application of a type environment  $\Gamma$  to a SLC expression  $e$  as the replacement of all the variables by their respective types in the  $\Gamma$  mapping, possibly yielding a hybrid expression. The completeness of the rewrite system is a direct corollary of lemma 1 by setting  $\Gamma$  to the empty environment.

**Lemma 1 (Reduction System Completeness).** *If  $\Gamma \vdash e : \tau$ , then  $\Gamma e \mapsto_t^* \tau$ .*

*Proof (by rule induction on  $\Gamma \vdash e : \tau$ ).*

**Case [t-num]:** *number*  $\mapsto_t$  num by [tc-num]

**Case [t-var]:** By [t-var],  $x : \tau \in \Gamma$ .  $\Gamma x = \tau \mapsto_t^0 \tau$  by the definition of  $\Gamma$  application.

**Case [t-lam]:** By [t-lam],  $\Gamma \vdash (\lambda (x \tau) e) : (\rightarrow \tau \tau')$  only if  $\Gamma[x : \tau] \vdash e : \tau'$  by inversion.

$$\begin{aligned} \Gamma (\lambda (x \tau) e) &= (\lambda (x \tau) (\Gamma e)) \\ &\mapsto_t (\rightarrow \tau \Gamma[x : \tau]e) && \text{[tc-lam] and lem. 2 (\Gamma permutation lem. below)} \\ &\mapsto_t^* (\rightarrow \tau \tau') && \text{by induction} \end{aligned}$$

**Case [t-app]:** By [t-app],  $\Gamma \vdash (e e') : \tau$  only if  $\Gamma \vdash e : (\rightarrow \tau' \tau)$  and  $\Gamma \vdash e' : \tau'$ .

$$\begin{aligned} (@ e e') &\mapsto_t^* (@ (\rightarrow \tau' \tau) e') && \text{by induction, lem. 3 (embedding lem. given below)} \\ &\mapsto_t^* (@ (\rightarrow \tau' \tau) \tau') && \text{by induction, lem. 3 (embedding lem. given below)} \\ &\mapsto_t \tau && \text{[tc-}\tau\beta\text{]} \end{aligned}$$

□

**Lemma 2 (Permutation of  $\lambda$ -bindings in Type Environment).**  $[x : \tau](\Gamma x) = (\Gamma[x : \tau])x$

*Proof.* Because substitution is capture-avoiding,  $x \notin \text{dom}(\Gamma)$ . Thus, I can freely permute the bindings. □

**Lemma 3 (Embedding of Reduction Sequence in Context).**

*For all  $T$ , if  $e \mapsto_t^* \tau$ , then  $T[e] \mapsto_t^* T[\tau]$ .*

*Proof (by induction on the structure of  $T$ ).* The intuition is the following: Every expression can be decomposed into a context  $T'$  and a redex. By definition of the context, plugging  $T'$  into  $T$  yields a valid context. This combination of contexts yields a new context  $T''$  such that  $T[e]$  decomposes into  $T''$  and the original redex. This decomposition reduces to the original reduct plugged into  $T''$ . □

Proving soundness is considerably more involved. At first glance, it seems that each type inference rule in Fig. 2.1 corresponds to a particular rewrite rule in Fig. 2.2 except for the [t-var] rule. The rules for num are identical. The [tc-lam] rule obtains an arrow type and the [t-app] and [tc- $\tau\beta$ ] rules yield the result type after matching parameter and argument types. However, there are pronounced differences in how the rules operate and

how roles are divided among them. The conspicuous absence of a rewrite rule equivalent to [t-var] is due to the eager application of substitution in [tc-lam]. The rewrite rules keep no type environment because they substitute types for variables as soon as the reductions reach each  $\lambda$ -binding. Consequently, the rewrite rules do no environment lookup for the variable types. The typing rules also recur on subexpressions whereas the rewrite rules do not explicitly recur by themselves. Instead, the context calculation provides the necessary recursion for typechecking in the rewrite system. Because the rewrite system does implicit context calculation before every rewrite, there is no need for search rules in the rewrite system.

Having established that the rewrite system uses eager substitutions instead of environments and implicit context calculation instead of search rules, I need to bridge these differences in order to prove soundness relating the rewrite system and the traditional notion of well-typing. I can connect the rewrite system to a CEK machine for the rewrite system [10, 12]. The CEK machine can relate substitutions to environments and make the context calculation explicit. I can then easily relate the machine to the usual notion of well-typing. In Fig. A.1, I specify such a machine.

The machine in Fig. A.1 corresponds much more precisely to the type inference rules. The machine states are of the form  $(e_h, \Gamma, k)$  where  $k$  is the context factoring in stack form,  $\Gamma$  is an environment mapping  $\lambda$ -bound variables to types, and  $e_h$  is the control expression. The control is a  $\tau\beta$  redex, SLC expression, or a type because all non- $\tau\beta$  hybrid expression structure are relegated to the context stack  $k$  and SLC variables are not substituted away for types because the environment  $\Gamma$  suspends such substitutions.

Using the machine, I can use [tm-app], [tm-app-left], [tm-app-right], and [tm-arr] to make the context calculation explicit and therefore relate the rewrites to the inference search rules. The two complementary rules [tm-lam] and [tm-arr] push and pop a  $\lambda$  binding frame  $[x : \tau]$  on  $\Gamma$  to prevent mappings for  $\lambda$ -bound variables from seeping out of the correct context.

To prove soundness, a relationship between the context stack and the control must be established. The following lemma is a result of the construction of the typechecking context.

$f ::= (@ \square e) \mid (@ \tau \square) \mid (\rightarrow \tau \square)[\Gamma]$	Frames
$k ::= \bullet \mid f :: k$	Stacks
$\Gamma ::= \bullet \mid \Gamma[x : \tau]$	
$((@ e e'), \Gamma, k) \mapsto_{tm} (e, \Gamma, (@ \square e') :: k)$	[tm-app]
$(\tau, \Gamma, (@ \square e') :: k) \mapsto_{tm} (e', \Gamma, (@ \tau \square) :: k)$	[tm-app-left]
$(\tau', \Gamma, (@ \tau \square) :: k) \mapsto_{tm} ((@ \tau \tau'), \Gamma, k)$	[tm-app-right]
$((\lambda (x \tau) e), \Gamma, k) \mapsto_{tm} (x, \Gamma[x : \tau], (\rightarrow \tau \square) :: k)$	[tm-lam]
$(\tau', \Gamma[x : \tau], (\rightarrow \tau \square) :: k) \mapsto_{tm} ((\rightarrow \tau \tau'), \Gamma, k)$	[tm-arr]
$(number, \Gamma, k) \mapsto_{tm} (num, \Gamma, k)$	[tm-num]
$((@ (\rightarrow \tau \tau') \tau), \Gamma, k) \mapsto_{tm} (\tau', \Gamma, k)$	[tm- $\tau\beta$ ]
$(x, \Gamma, k) \mapsto_{tm} (\Gamma(x), \Gamma, k)$	[tm-var]

Machine states are of the form  $(e_h, \Gamma, k)$ .

Initial states are of the form  $(e, \cdot, \cdot)$ .

Figure A.1: Type Checking Machine Reductions

**Lemma 4.** *The machine rewrite rules pop the context stack if and only if the control is a  $\tau$ .*

*Proof (by inspection).*

□

**Lemma 5 (Reduction of  $e$  to  $\tau$  (under  $\Gamma$ ) is independent of the context  $k$ ).**

*For any  $k$ , if  $(e, \Gamma, k) \mapsto_{tm}^* (\tau, \Gamma, k)$ , then  $\Gamma \vdash e : \tau$ .*

*Proof (by induction on the number of reductions).*

I proceed by a case analysis on  $e$ :

**Base Cases:**

**Case  $e = number$ :**  $(number, \Gamma, k) \mapsto_{tm} (num, \Gamma, k)$

By [t-num],  $\Gamma \vdash number : num$ .

**Case  $e = x$ :**  $(x, \Gamma, k) \mapsto_{tm} (\tau, \Gamma, k)$  such that  $\Gamma(x) = \tau$ .

By [t-var],  $\Gamma \vdash x : \tau$ .

**Inductive Cases:** I want to show that  $(e, \Gamma, k) \mapsto_{tm}^{n+1} (\tau, \Gamma, k) \Rightarrow \Gamma \vdash e : \tau$  assuming the induction hypothesis that  $\forall k, \Gamma, e, \tau. (e, \Gamma, k) \mapsto_{tm}^m (\tau, \Gamma, k) \Rightarrow \Gamma \vdash e : \tau$  whenever  $m \leq n$ .

**Case  $(\lambda (x \tau) e)$ :**

$$\begin{aligned} ((\lambda (x \tau) e), \Gamma, k) &\mapsto_{tm} (e, \Gamma[x : \tau], (\rightarrow \tau \square)::k) \text{ [tm-lam]} \\ &\mapsto_{tm}^* (\tau', \Gamma[x : \tau], (\rightarrow \tau \square)::k) \text{ lemma 4} \\ &\mapsto_{tm} ((\rightarrow \tau \tau'), \Gamma, k) \text{ [tm-arr]} \end{aligned}$$

By induction,  $\Gamma[x : \tau] \vdash e : \tau'$ .

By [t-lam],  $\Gamma \vdash (\lambda (x \tau) e) : \tau \rightarrow \tau'$ .

**Case  $(@ e e')$ :**

$$\begin{aligned} ((@ e e'), \Gamma, k) &\mapsto_{tm} (e, \Gamma, (@ \square e')::k) \text{ [tm-app]} \\ &\mapsto_{tm}^* (\tau, \Gamma, (@ \square e')::k) \text{ lemma 4} \\ &\mapsto_{tm} (e', \Gamma, (@ \tau \square)::k) \text{ [tm-app-left]} \\ &\mapsto_{tm}^* (\tau', \Gamma, (@ \tau \square)::k) \text{ lemma 4} \\ &\mapsto_{tm} ((@ \tau \tau'), \Gamma, k) \text{ [tm-app-right]} \\ &\mapsto_{tm} (\tau'', \Gamma, k) \text{ [tm-}\tau\beta\text{] where } \tau = (\rightarrow \tau \tau'') \end{aligned}$$

By induction,  $\Gamma \vdash e : \tau$  and  $\Gamma \vdash e' : \tau'$ .

By [t-app],  $\Gamma \vdash (@ e e') : \tau''$ .

□

To complete the proof of soundness of the TC rewrite system with respect to the well-typing relation, I establish the soundness of the TC rewrite system with respect to the machine, which was already shown to be sound.

**Lemma 6 (Embedding of Machine Reduction Sequence in Context).**

*If  $(e, \Gamma, \cdot) \mapsto_{tm}^* (\tau, \Gamma, \cdot)$ , then  $(e, \Gamma, k) \mapsto_{tm}^* (\tau, \Gamma, k)$ .*

*Proof.* By induction on the structure of  $k$ .

□

**Lemma 7 (TC Machine Soundness).** *If  $\Gamma e \mapsto_t^* \tau$ , then  $(e, \Gamma, \bullet) \mapsto_{tm}^* (\tau, \Gamma, \bullet)$*

*Proof (by induction on the number of  $\mapsto_t$  reductions).*

**Base case:**

**Case  $e = x$ :** By definition of  $\Gamma$  application,  $\Gamma x = \tau$  if  $x : \tau \in \Gamma$ .  $(x, \Gamma, \cdot) \mapsto_{tm} (\Gamma(x), \Gamma, \cdot)$  by [tm-var]. By definition of substitution,  $\Gamma(x) = \tau$ .

Otherwise,  $(x, \Gamma, \cdot)$  is stuck, in which case the claim is vacuously true.

**Case  $e = \text{number}$ :**  $\Gamma \text{ number} \mapsto_t \text{num}$ .  $\Gamma \text{ number} = \text{number}$ .  $(\text{number}, \Gamma, \cdot) \mapsto_{tm} (\text{num}, \Gamma, \cdot)$  by [tm-num].

### Inductive cases:

**Case  $e = (\lambda (x \tau') e')$ :**

$\Gamma e = (\lambda (x \tau') e') = (\lambda (x \tau') \Gamma e')$  capture-avoiding substitution  
(rename  $x$  so that  $x \notin \text{dom}(\Gamma)$ )

$\mapsto_t (\rightarrow \tau' (\Gamma[x : \tau'] e'))$  [tc-lam]

$\mapsto_t^* (\rightarrow \tau' \tau'')$  by lem. 3

By induction,  $(e', \Gamma[x : \tau'], \bullet) \mapsto_{tm}^* (\tau'', \Gamma[x : \tau'], \bullet)$ . Because these same rewrites occur for the given control regardless of the typechecking context stack

by lem. 6, I can establish the following machine reduction sequence:

$((\lambda (x \tau') e'), \Gamma, \bullet) \mapsto_{tm} (e', \Gamma[x : \tau'], (\rightarrow \tau' \square):: \cdot)$  [tm-lam]

$\mapsto_{tm}^* (\tau'', \Gamma[x : \tau'], (\rightarrow \tau' \square):: \cdot)$  by induction

$\mapsto_{tm} ((\rightarrow \tau' \tau''), \Gamma, \bullet)$  [tm-arr]

**Case  $e = (@ e' e'')$ :** By assumption,  $\Gamma e \mapsto_t^* \tau$  (i.e.,  $\Gamma(@ e' e'') \mapsto_t^* \tau$ )

$\Gamma e = \Gamma(@ e' e'')$

$= (@ \Gamma e' \Gamma e'')$

$\mapsto_t^* (@ \tau e'')$  by lem. 3

$\mapsto_t^* (@ \tau \tau')$  by lem. 3

$\mapsto_t \tau''$  [tc- $\tau\beta$ ] where  $\tau = (\rightarrow \tau' \tau'')$

I can construct a corresponding machine reduction sequence:

$((@ e' e''), \Gamma, \bullet) \mapsto_{tm} (e', \Gamma, (@ \square e''))$  [tm-app]

$\mapsto_{tm}^* (\tau, \Gamma, (@ \square e''))$  by induction and lemma 6

$\mapsto_{tm} (e'', \Gamma, (@ \tau \square))$  [tm-app-left]

$\mapsto_{tm}^* (\tau', \Gamma, (@ \tau \square))$  by induction and lemma 6

$\mapsto_{tm} ((@ \tau \tau'), \Gamma, \bullet)$  [tm-app-right]

$\mapsto_{tm} (\tau'', \Gamma, \bullet)$  [tm- $\tau\beta$ ]

□

## APPENDIX B

### ND AND CURRY/HINDLEY SYSTEMS PROOFS

The nondeterministic type inference rewrite system in Fig. 3.2 is both sound and complete relative to the explicitly typed calculus in Fig. 2.2. To relate the two systems, use an erase function that is the usual notion of type erasure (i.e.,  $\mathcal{E}((\lambda (x \tau) e)) = (\lambda (x) \mathcal{E}(e))$ , erase of applications recurs, and erase of constants does nothing).

**Theorem 1 (ND Reduction Soundness).** *For any ULC expression  $e$  and type  $\tau$ , if  $e \mapsto_n^* \tau$ , then there exists SLC expression  $e'$  such that  $\mathcal{E}(e') = e$  and  $e' \mapsto_t^* \tau$ .*

*Proof.* This theorem follows directly from lemma 2. □

As before, I rely heavily on the observation that if a redex reduces in the empty context, it must also reduce under all well-formed contexts  $T_n \in \text{CTX}$  (and vice versa). This observation must be restated (as in lem. 1 below) because the  $T_n$  context differs from the  $T$  context. In particular, the ULC-H expressions replacing SLC-H expressions on the right side of the  $(@ T_n e_n)$  context.

**Lemma 1 (Embedding of ND Reduction Sequences in Context).**

*For all  $T_n$ , if  $e \mapsto_n^* \tau$ , then  $T_n[e] \mapsto_n^* T_n[\tau]$ .*

*Proof (by induction on the structure of  $T_n$ ).* □

To prove theorem 1, I must strengthen the induction to work when an arbitrary type environment (call it  $\Gamma$ ) is applied to the all ULC-H expressions ( $\text{ULC} \subseteq \text{ULC-H}$ ). This strengthening will allow the [nd-lam] case to go through. Note that  $\Gamma e_n \neq e_n$  only if  $\text{fv}(e_n) \neq \emptyset$  and  $\text{fv}(e_n) \cap \text{dom}(\Gamma) \neq \emptyset$ .

**Lemma 2 (ND Reduction Substitution).** *For any expression  $e_n$ , type environment  $\Gamma$ , and type  $\tau$ , if  $\Gamma e_n \mapsto_n^* \tau$ , then there exists  $e_h$  such that  $\mathcal{E}(e_h) = e_n$  and  $\Gamma e_h \mapsto_t^* \tau$ .*

*Proof (By induction on  $e_n$ ).*

**Case  $e_n = \text{number}$  or  $\text{num type}$ :** Trivial because  $\mapsto_n$  and  $\mapsto_t$  are identical for this  $e_n$  ( $\mathcal{E}(e_n) = e_n$ ).

**Case  $e_n = x$ :**

**Subcase  $x \in \text{dom}(\Gamma)$ :**  $\mathcal{E}(x) = x$  and  $\Gamma x = \tau \mapsto_t^0 \tau$ .

**Subcase  $x \notin \text{dom}(\Gamma)$ :**  $\Gamma(x) = x$  is stuck. Vacuously true

**Case  $e_n = (\lambda(y) e'_n)$ :** By definition of capture-avoiding substitution ( $\alpha$ -renaming ensures that  $y \notin \text{dom}(\Gamma)$ ),  $\Gamma(\lambda(y) e'_n) = (\lambda(y) \Gamma e'_n)$ . By lem. 1, for some types  $\tau$  and  $\tau'$ ,  $\lambda(y) \Gamma e'_n \mapsto_n (\rightarrow \tau ([y : \tau] \Gamma e'_n) \mapsto_n^* (\rightarrow \tau \tau')$ . Therefore,  $[y : \tau] \Gamma e'_n \mapsto_n^* \tau'$ . To keep consistent with the notation for type environments, the notation  $[y : \tau]$  will be used interchangeably with the substitution notation  $\{y \mapsto \tau\}$ . The two notations denote exactly the same mapping. By induction, there exists  $e'_h$  such that  $\mathcal{E}(e'_h) = e'_n$  and  $([y : \tau] \Gamma e'_h) \mapsto_t^* \tau'$ . Therefore let  $e''_h = (\lambda(y \tau) e'_h)$ . Then  $\mathcal{E}(e''_h) = (\lambda(y) e'_n)$  and  $\Gamma e''_h = (\lambda(y \tau) \Gamma e'_h) \mapsto_t (\rightarrow \tau [y : \tau] \Gamma e'_h) \mapsto_t^* (\rightarrow \tau \tau')$  by lem. 1.

**Case  $e_n = (e'_n e''_n)$ :** By definition of  $\Gamma$  application,  $\Gamma(@ e'_n e''_n) = (@ \Gamma e'_n \Gamma e''_n)$ . By lem. 1,  $(@ \Gamma e'_n \Gamma e''_n) \mapsto_n^* (@ \tau' \Gamma e''_n) \mapsto_n^* (@ \tau' \tau')$ . If  $\tau' \neq (\rightarrow \tau'' \tau)$  then the term would be stuck. Thus,  $\tau' = (\rightarrow \tau'' \tau)$  and  $(@ \tau' \tau') \mapsto_n \tau$ . By induction,  $\exists e'_h$  and  $e''_h$  such that  $\mathcal{E}(e'_h) = e'_n$ ,  $\mathcal{E}(e''_h) = e''_n$ ,  $\Gamma e'_h \mapsto_t^* \tau'$ , and  $\Gamma e''_h \mapsto_t^* \tau''$ . By lem. 1,  $\Gamma(@ e'_h e''_h) = (@ (\Gamma e'_h) (\Gamma e''_h)) \mapsto_t^* (@ \tau' (\Gamma e''_h)) \mapsto_t^* (@ \tau' \tau') = (@ (\rightarrow \tau'' \tau) \tau') \mapsto_t \tau$ .

□

The Completeness of the ND reduction relative to TC is trivial because any TC type-checking reduction can be converted into a valid ND reduction (by erasing the types in the explicitly typed expressions).

To proceed with the soundness and completeness proofs for the system in Fig. 3.3 (the CH system), I reproduce the grammar for CH and ND sublanguages here in Fig. B.1. The CH system is sound and complete relative to the ND system in an indirect way. First, because the CH system has unification reduction steps that do not directly relate to any ND reductions, I define a minor variation on the CH  $\mapsto_u$  reduction called  $\mapsto_c$ . The  $\mapsto_c$

---

$e ::= x \mid (\lambda (x) e) \mid (@ e e)$	ULC
$e_n ::= x \mid (\lambda (x) e_n) \mid (@ e_n e_n) \mid \text{number} \mid (\rightarrow \tau e_n) \mid \text{num}$	ULC-H
$T_n ::= (@ T_n e_n) \mid (@ \tau T_n) \mid (\rightarrow \tau T_n) \mid \square$	CTX
$\tau ::= \text{num} \mid (\rightarrow \tau \tau)$	TYPE
$p ::= (\mathbf{unify} \tau_u \tau_u p) \mid e_u$	Unify problems
$e_u ::= x \mid (\lambda (x) e_u) \mid (@ e_u e_u) \mid \text{number} \mid (\rightarrow \tau_u e_u) \mid \text{num} \mid \xi$	ULC-HV
$T_u ::= (@ T_u e_u) \mid (@ \tau_u T_u) \mid (\rightarrow \tau_u T_u) \mid \square$	CTX-V
$\tau_u ::= \text{num} \mid (\rightarrow \tau_u \tau_u) \mid \xi$	TYPE-V
$\xi, \zeta ::= \text{type variables}$	TVAR

Figure B.1: CH and ND languages and sublanguages from Figs. 3.2 and 3.3.

---

$T_u[\text{number}] \mapsto_c T_u[\text{num}]$	[c-ch-num]
$T_u[(\lambda (x) e_u)] \mapsto_c T_u[(\rightarrow \xi \{x \mapsto \xi\} e_u)]$ $\xi \text{ fresh}$	[c-ch-lam]
$T_u[(@ \tau_{u_1} \tau_{u_2})] \mapsto_c \mathcal{U}((\rightarrow \tau_{u_2} \xi), \tau_{u_1}) T_u[\xi]$ $\xi \text{ fresh}$	[c-ch- $\tau\beta$ ]

$\mathcal{U}(\tau_u, \tau'_u) =$  a mgu of  $\tau_u$  and  $\tau'_u$  if one exists (obtain by using  $\mapsto_f$  in Fig. B.3 below), otherwise causes a stuck state.

Figure B.2: Variation on Reduction Rules for CH System with Meta-level Unification

reduction forgoes the unification reduction rules [ch-u-eq], [ch-u-dist], [ch-u-orient], and [ch-u-inst] in favor of a metafunction  $\mathcal{U}$  that produces the unification substitution that the unification reductions would have ultimately produced. Using  $\mathcal{U}$ , [c-ch- $\tau\beta$ ] can now complete unification in a single  $\mapsto_c$  reduction step. If no mgu exists, then [c-ch- $\tau\beta$ ] gets stuck. In every other respect,  $\mapsto_c$  is identical to  $\mapsto_u$ . The  $\mapsto_c$  reduction relation is formally defined in Fig. B.2. The  $\mapsto_c$  reduction system is sound and complete with respect to the  $\mapsto_u$  reduction system (CH) by construction and by the correctness of the unification reductions (thm. 14), shown at the end of this appendix, lem. 14. Now I can match up [nd- $\tau\beta$ ] to [c-ch- $\tau\beta$ ].

**Lemma 3 (Soundness and Completeness of  $\mapsto_c$  with respect to  $\mapsto_u$ ).**

*For all ULC expression  $e$ ,  $e \mapsto_c^* \tau_u$  if and only if  $e \mapsto_u^* \tau_u$ .*

*Proof.* This lemma is true because  $\mapsto_c$  is identical to  $\mapsto_u$  by construction except for the unification reductions in  $\mapsto_u$  (listed in Fig. B.3). Instead, [c-ch- $\tau\beta$ ] uses the  $\mathcal{U}$  function to

$(\mathbf{unify} \tau_{u_1} \tau_{u_1} p) \mapsto_f p$	[ch-u-eq]
$(\mathbf{unify} (\rightarrow \tau_{u_1} \tau_{u_2}) (\rightarrow \tau_{u_3} \tau_{u_4}) p) \mapsto_f (\mathbf{unify} \tau_{u_1} \tau_{u_3} (\mathbf{unify} \tau_{u_2} \tau_{u_4} p))$	[ch-u-dist]
$(\rightarrow \tau_{u_1} \tau_{u_2}) \neq (\rightarrow \tau_{u_3} \tau_{u_4})$	
$(\mathbf{unify} \tau_u \xi p) \mapsto_f (\mathbf{unify} \xi \tau_u p)$	[ch-u-orient]
$\tau_u \neq \xi$	
$(\mathbf{unify} \xi \tau_u p) \mapsto_f \{\xi \mapsto \tau_u\}p$	[ch-u-inst]
$\xi \notin \text{ftv}(\tau_u)$	
The $f$ stands for the $f$ in $\mathbf{unify}$ .	$\mapsto_f \sqsubset \mapsto_u$

Figure B.3: CH Unification Reduction Rules

obtain the mgu of two types and [ch- $\tau\beta$ ] uses the unification reduction rules to obtain this same mgu (by lem. 14 proven below), Both rules apply this mgu to the context and (in the hole) the fresh type variable representing the return type of the function. Thus, both reduce to the same reduct if a mgu exists. If a mgu does not exist, then both get stuck.  $\square$

Type variables introduced by [c-ch-lam] may be instantiated by unification, completed by [c-ch- $\tau\beta$ ]. They also may be never instantiated. I call the type variables that are instantiated at some point **constrained** type variables. The type variables that are never instantiated are called **unconstrained** type variables. Let  $\sigma$  be a TVAR to TYPE-V substitution (a **type substitution**), i.e. a mgu substitution produced by unification. Because types  $\tau_u \in \text{TYPE-V}$  may contain type variables,  $\sigma$  does not necessarily instantiate any type variables to ground types. A **complete reduction sequence**  $e \mapsto_c^* \tau_u$  (i.e., a reduction sequence that from a ULC term to a type that may contain residual unconstrained type variables) may apply a number of unification substitutions  $\sigma_i$ 's through the course of the reduction sequence. Call the composition of all these unification substitutions the complete unification substitution  $\theta$ .

**Definition 1 (Complete Unification Substitution).** *A complete unification substitution  $\theta$  for a complete reduction sequence  $e \mapsto_c^* \tau_u$  is the composition of all the unification substitutions applied through the course of the reduction sequence. That is, the reduction sequence  $e \mapsto_c^* \sigma_1 e_{u1} \mapsto_c^* \sigma_2 e_{u2} \mapsto_c^* \dots \mapsto_c^* \sigma_n e_{un} \mapsto_c^* \tau_u$  has the complete unification substitution  $\theta = \sigma_n \dots \sigma_2 \sigma_1$  where all  $\sigma_i e_{ui}$ 's are the right hand sides of [c-ch- $\tau\beta$ ] transitions in the reduction sequence.*

Each complete reduction sequence only has a single, unique complete unification substitution because unification function  $\mathcal{U}$  yields only one (oriented) mgu<sup>1</sup> for each unification problem and the composition of all the mgus is unique.

I will be reasoning extensively about substitutions in the following proofs. Let support of a substitution  $\sigma$ ,  $\text{supp}(\sigma)$ , be the set of variables or type variables that  $\sigma$  acts upon. The set of constrained type variables for a complete reduction is  $\text{supp}(\theta)$ . The complete unification substitution does not necessarily instantiate all type variables in the reduction sequence to ground types. In general, there may be unconstrained type variables throughout the reduction sequence including in the final type reduct  $\tau_u$ . Let  $\Delta$  denote the set of all unconstrained type variables. This set of unconstrained type variables for the complete reduction sequence is  $\Delta = \text{ftv}(e \mapsto_c^* \tau_u) \setminus \text{supp}(\theta)$  (i.e., all other type variables occurring in any term of the complete reduction sequence). Thus, in order to instantiate all the type variables in each step of the reduction sequence, these unconstrained type variables must also be replaced by ground types. Because the type variables in  $\Delta$  are completely unconstrained, I may instantiate them to any ground type. Thus, any **unconstrained type variable substitution**  $\delta : \Delta \rightarrow \text{TYPE}$  whose domain is the entire set  $\Delta$  can substitute away the unconstrained type variables without interfering with the rest of reduction sequence.

**Definition 2 (Unconstrained Type Variable Substitution).** *An unconstrained type variable substitution  $\delta$  is a substitution mapping the unconstrained type variables ( $\Delta$ ) introduced in a complete reduction sequence  $e \mapsto_c^* \tau_u$  to ground types (TYPE).*

Because I can substitute any ground type (of which there are an infinite number) for unconstrained type variables, any complete reduction sequence has an infinite number of associated unconstrained type variable substitutions. Applying the composition of the complete unification substitution and the unconstrained type variable substitution  $\delta\theta$  to any term  $e_u \in \text{ULC-HV}$  in the complete reduction sequence yields the corresponding term  $\delta\theta e_u \in \text{ULC-H}$  in one of the many analogous ND reduction sequences. There are possibly an infinite number of analogous ND reduction sequences because there are possibly an infinite number of distinct unconstrained type variable substitutions.

---

1. The mgu's produced by  $\mathcal{U}$  are unique because they are "oriented". For example,  $\mathcal{U}(\xi, \zeta)$  always yields  $\{\xi \mapsto \zeta\}$  rather than  $\{\zeta \mapsto \xi\}$ .

Because unification is subject to the occurs check, unification substitutions must be idempotent as are unconstrained type variable substitutions for obvious reasons. This result is summarized in lem. 4. I prove the soundness of the CH in two steps. Completeness is relatively straightforward. Given the soundness/completeness of  $\mapsto_c$  (lem. 3), I can prove soundness of  $\mapsto_c$  with respect to  $\mapsto_n$  in lem. 10.

**Lemma 4 (Unification and Unconstrained Type Variable Substitutions are Idempotent).**

*Unification substitutions  $\theta$  and unconstrained type variable substitutions  $\delta$  are all idempotent.*

*Proof.* Unification substitutions are idempotent because if they were not, the substitution would not pass the occurs check. Unconstrained type variable substitutions are idempotent because they instantiate type variables only to ground types which contain no type variables, thus they are certainly idempotent.  $\square$

**Lemma 5 (The Composition of the Complete Unification and an Unconstrained Type Variable Substitutions Eliminates All Type Variables).** *For any complete reduction sequence  $e \mapsto_c^* \tau_u$ , let  $\theta$  and  $\delta$  be the complete unification substitution and an unconstrained type variable substitution respectively.*

*$\delta\theta$  maps type variables found in the reduction sequence to types that do not contain type variables, i.e.  $\delta\theta : TVAR \rightarrow TYPE$  and  $\delta\theta(\xi) \in TYPE$ .*

*Proof.* Let  $\xi$  be a type variable found in the reduction sequence.

$(\xi \in \text{supp}(\theta))$ :  $\delta\theta(\xi) = \delta(\tau_u)$  where  $\theta(\xi) = \tau'_u$ . Because all the substitutions are idempotent (lem. 4), none of the type variables in  $\tau'_u$  can be in the support of  $\theta$ , i.e.  $\text{ftv}(\tau'_u) \subseteq \Delta$ . Thus, by definition,  $\delta$  must map all the type variables in  $\tau_u$  to ground types. Thus, by definition  $\delta(\tau_u) \in TYPE$ .

$(\xi \notin \text{supp}(\theta))$ : Then  $\xi \in \Delta$ , and by definition  $\text{supp}(\delta) = \Delta$ , so  $\delta\theta(\xi) = \delta(\xi) \in TYPE$ .

$\square$

**Lemma 6 (Corollary to Lem. 5).** *Given a complete reduction sequence  $e \mapsto_c^* \tau_u$ , let  $\theta$  be its complete unification substitution and let  $\delta$  be its unconstrained type variable substitution. The following must be true: For any  $e_u \in \text{ULC-HV}$  occurring in the reduction sequence,  $\delta\theta e_u \in \text{ULC-H}$ .*

*For any  $\tau'_u \in \text{TYPE-V}$  occurring the reduction sequence,  $\delta\theta\tau'_u \in \text{TYPE}$ .*

*For any  $T_u \in \text{CTX-V}$  occurring in the reduction sequence,  $\delta\theta T_u \in \text{CTX}$ .*

*Proof* (By structural induction on  $e_u$ ,  $\tau'_u$ , and  $T_u$  respectively). The only interesting case is the  $\xi$  case which follows from lemma 5.  $\square$

**Lemma 7 (Embedding of ND Expressions, Types, and Contexts in CH Expressions, Types, and Contexts).** *There is an embedding from ULC-H to ULC-HV and the image of this embedding is the subset of ULC-HV consisting of expressions that that contain no type variables.*

*Analogous properties hold for TYPE to TYPE-V and CTX to CTX-V.*

*Proof.* The identity embedding will work here because  $(\text{ULC-H} \subset \text{ULC-HV})$ . There are no type variables in the image of this embedding because ULC-H does not involve type variables by definition. Furthermore, any  $e_u \in \text{ULC-HV}$  that contains no type variables is in the image of ULC-H under this embedding, and so it can be considered as belonging to ULC-H.

Similarly,  $\text{TYPE} \subset \text{TYPE-V}$  and  $\text{CTX} \subset \text{CTX-V}$ , so the respective identity embeddings work. Moreover, neither TYPE nor CTX involve type variables by definition.  $\square$

**Lemma 8 (Composition of Complete Unification and Unconstrained Type Variable Substitutions Maps CH Expressions, Types, and Contexts into ND Expressions, Types, and Contexts).** *For any  $e_u, \tau'_u, T_u$  in a complete reduction sequence,  $e \mapsto_c^* \tau_u$  where  $\theta$  and  $\delta$  are the complete unification substitution and the unconstrained type variable substitutions respectively, the following must be true:*

$\delta\theta e_u \in \text{ULC-H}$ .

$\delta\theta\tau'_u \in \text{TYPE}$ .

$\delta\theta T_u \in \text{CTX}$ .

*Proof.* This lemma follows directly from lemmas 6 and 7.  $\square$

Finally, I also need a little fact about how substitutions distribute over context-redex decomposition when applied.

**Lemma 9 (Substitutions Distribute Over Context Decomposition).**

For all  $T_u \in \text{CTX-V}$  and  $e_u \in \text{ULC-HV}$ ,  $\sigma(T_u[e_u]) = (\sigma T_u)[\sigma[e_u]]$

*Proof (By induction on the structure of  $T_u$ ).*

**Case  $T_u = \square$ :**  $\sigma(\square[e_u]) = \sigma e_u = (\sigma \square)[\sigma e_u]$

**Case  $T_u = (@ T_u e'_u)$ :**

$$\begin{aligned} \sigma((@ T_u e'_u)[e_u]) &= \sigma(@ T_u[e_u] e'_u) && \text{by definition of decomposition} \\ &= (@ \sigma(T_u[e_u]) \sigma e'_u) && \text{by definition of substitution} \\ &= (@ (\sigma T_u)[\sigma e_u] \sigma e'_u) && \text{by induction} \\ &= (@ \sigma T_u \sigma e'_u)[\sigma e_u] && \text{by definition of decomposition} \end{aligned}$$

**Cases  $T_u = (@ \tau_u T_u)$  and  $T_u = (\rightarrow \tau T_u)$ :** These cases look similar to the above case. □

**Lemma 10 (Soundness of Curry/Hindley).** *Let  $\theta$  and  $\delta$  be the complete unification substitution and an unconstrained type variable substitution for the complete reduction sequence  $e \mapsto_c^* \tau_u$  respectively. If  $e \mapsto_c^* e_u \mapsto_c e'_u$ , then  $\delta\theta e_u \mapsto_n \delta\theta e'_u$ .*

*Proof (Case analysis on the rule giving  $e_u \mapsto_c e'_u$ ).*

**[c-ch-num]**  $T_u[\text{number}] \mapsto_c T_u[\text{num}]$ . (i.e.,  $e_u = T_u[\text{number}]$  and  $e'_u = T_u[\text{num}]$ )

$$\delta\theta(T_u[\text{number}]) = (\delta\theta T_u)[\delta\theta \text{number}] \text{ by lem. 9.}$$

By definition of substitution application,  $\delta\theta(\text{number}) = \text{number}$ .

Thus,  $\delta\theta(T_u[\text{number}]) = (\delta\theta T_u)[\text{number}]$ .

$\delta\theta T_u \in \text{CTX}$  by lem. 8. Let  $T_n = \delta\theta T_u$ .

$T_n[\text{number}] \mapsto_n T_n[\mathbf{num}] = \delta\theta T_u[\mathbf{num}]$  by [nd-num].

**[c-ch-lam]**  $T_u[(\lambda(x) e_u)] \mapsto_c T_u[(\rightarrow \xi \{x \mapsto \xi\} e_u)]$ .

$\delta\theta(T_u[(\lambda(x) e'_u)])$  contains no type variables by lem. 5.

$\delta\theta(T_u[(\lambda(x) e'_u)]) = (\delta\theta T_u)[(\lambda(x) \delta\theta e'_u)]$  by lem. 8.

By lemma,  $\delta\theta T_u$  is a  $T_n$  and  $\delta\theta e'_u$  is an  $e'_n$ . Therefore, by definition of  $e_n$ ,  $(\lambda(x) \delta\theta e'_u)$  is  $(\lambda(x) e'_n)$  which is an  $e_n$ .

By [nd-lam],  $(\delta\theta T_u)[(\lambda(x) \delta\theta e'_u)] \mapsto_n (\rightarrow \delta\theta\xi \{x \mapsto \delta\theta\xi\} \delta\theta e'_u)$ .

This is because  $\xi$  is in the support of  $\delta\theta$  by lem. 5 (i.e., either it is constrained or unconstrained). Thus,  $\delta\theta(\xi) \in \text{TYPE}$ .

**[c-ch- $\tau\beta$ ]**  $T_u[(\@ \tau_u \tau'_u)] \mapsto_c \mathcal{U}(\tau_u, (\rightarrow \tau'_u \xi))(T_u[\xi])$ .

Let  $\sigma = \mathcal{U}((\rightarrow \tau'_u \xi), \tau_u)$ .

Note that  $\delta\theta\sigma = \delta\theta$  because  $\sigma$  is included in  $\theta$  ( $\theta$  is the composition including all unification substitutions) and unification substitutions are idempotent (lem. 4).

$\delta\theta\sigma(T_u[(\@ \tau_u \tau'_u)])$ .

$\delta\theta T_u \in \text{CTX}$  by lemma 8.

$\delta\theta\sigma\tau_u \in \text{TYPE}$  by lemma 8.

$\delta\theta\tau'_u \in \text{TYPE}$  by lemma 8.

$\sigma\tau_u = \sigma(\rightarrow \tau'_u \xi)$  by definition of unification.

This case in the soundness proof requires the assumption of the correctness of unification (lem. 14) to relate  $\tau\beta$  rules and a lemma on the well-formedness of the reduct after unification, lem. 16.

$\delta\theta\sigma\tau_u = \delta\theta\sigma(\rightarrow \tau'_u \xi)$

$\delta\theta\sigma\tau_u = (\rightarrow \delta\theta\sigma\tau'_u \delta\theta\sigma\xi)$ . Therefore  $\delta\theta\sigma\xi \in \text{TYPE}$  because certain a sub-component of type without type variables (i.e., a type in TYPE) is still in TYPE.

By [nd- $\tau\beta$ ],  $\delta\theta\sigma(T_u[(\@ \tau_u \tau'_u)]) = T_n[(\@ \tau_n \tau'_n)] = T_n[(\@ (\rightarrow \tau'_n \tau''_n) \tau'_n)] \mapsto_n T_n[\tau''_n] = (\delta\theta T_u)[\delta\theta\xi] = \delta\theta(T_u[\xi])$

□

**Theorem 2 (Soundness of Curry/Hindley System).** *If  $e_n \mapsto_u^* \tau_u$ , then there exists a type substitution  $\sigma$  such that  $\sigma\tau_u = \tau$  and  $e_n \mapsto_n^* \tau$ .*

*Proof.* This theorem results from the soundness of  $\mapsto_u$  with respect to  $\mapsto_c$  stated in lem. 3 and the transitive closure of lem. 10 (proving that  $\mapsto_c$  is sound with respect to  $\mapsto_n$ . Let  $\sigma = \delta\theta$  where  $\delta$  and  $\theta$  are the unconstrained type variable and complete unification substitutions respectively from lem. 10.  $\square$

For the completeness of the Curry/Hindley System, I need to introduce an **instantiating substitution**  $\sigma$ . An instantiating substitution maps a set of type variables (each associated with a particular program variable) to ground types (of the respective program variables). When constructing the analogous CH reduction sequence, for every [ch-lam] and [ch- $\tau\beta$ ] step, I use this instantiating substitution where I previously used the complete unification substitution. Because a complete ND reduction sequence guarantees that unifications go through for the analogous CH reduction sequence, I do not have to resort to the  $\mapsto_c$  reduction used in the previous section. Instead, I can prove the completeness of the CH reduction ( $\mapsto_u$ ) directly. Also, note that completeness result in lem. 12 works for any single step ND reduction but possibly multistep CH reductions ( $\mapsto_u$ ). This detail is required because where [nd- $\tau\beta$ ] directly reduces to the result type of an application, [ch- $\tau\beta$ ] introduces a unification problem that is solved in potentially many steps because it determines the result type of that application.

**Definition 3 (Instantiating Substitution).** *For each [nd-lam] type environment binding  $[x_i : \tau_i]$  applied in the course of a complete ND reduction sequence  $e \mapsto_n^* \tau$ , associate a fresh type variable  $\xi_j$ . For each [nd- $\tau\beta$ ] step that reduces to a result (range) type  $\tau'_j$ , associate a fresh type variable  $\zeta_j$ .*

*An instantiating substitution  $\sigma$  for the said complete ND reduction is a type substitution  $\{\xi_1 \mapsto \tau_1, \xi_2 \mapsto \tau_2, \dots, \xi_{n-1} \mapsto \tau_{n-1}, \xi_n \mapsto \tau, \zeta_1 \mapsto \tau'_1, \zeta_2 \mapsto \tau'_2, \dots, \zeta_{m-1} \mapsto \tau'_{m-1}, \zeta_m \mapsto \tau'_m\}$ .*

Instantiating substitutions should look very familiar because they are instances of a kind of substitution I defined earlier, the composition of the complete unification substitution and some unconstrained type variable substitution.

**Lemma 11.** *Instantiating substitutions for a complete ND reduction sequence are compositions of the complete unification and some unconstrained type variable substitutions for the corresponding complete CH reduction sequence.*

*Proof.* Note that instantiating substitutions are compositions of the complete unification substitution and some unconstrained type variable substitution. The instantiating substitution unifies the function and function schematic type in every  $[\text{ch-}\tau\beta]$  step, hence by definition of mgu, it must be a composition of some substitution with the mgus for all the  $[\text{ch-}\tau\beta]$  steps. Thus, by definition, the instantiating substitution contains the complete unification substitution. The range of the instantiating substitution does not contain type variables by definition. Thus, that substitution that is composed with the mgus must be an unconstrained type variable substitution.  $\square$

Lem. 8 holds for the composition of the complete unification substitution and any unconstrained type variable substitution. Instantiating substitutions are an instance of such compositions by lem. 11. Thus, lem. 8 holds for instantiating substitutions.

**Lemma 12 (Completeness of Curry/Hindley System).** *For any complete ND reduction sequence  $e \mapsto_n^* \tau$ , let  $\sigma$  be that reduction sequence's instantiating substitution. If  $T_n[e_n] \mapsto_n T_n[e'_n]$  and  $\sigma(T_u[e_u]) = T_n[e_n]$ , then there exists  $T'_u[e'_u]$  such that  $T_u[e_u] \mapsto_u^* T'_u[e'_u]$  and  $\sigma(T'_u[e'_u]) = T_n[e'_n]$ .*

*Proof (Case analysis on the  $\mapsto_n$  rewrite rules).*

**Case [nd-num]:**

$$T_n[e_n] = T_n[\text{number}] \mapsto_n \text{num by [nd-num]}$$

$$\sigma e_u = \text{number}, e_u = \text{number by definition of substitution.}$$

$$\text{Let } T'_u = T_u.$$

$$T_u[e_u] \mapsto_u T_u[\text{num}] \text{ by [ch-num].}$$

**Case [nd-lam]:**

$$T_n[e_n] = T_n[(\lambda(x) e'_n)] \mapsto_n T_n[(\rightarrow \tau e'_n)] \text{ by [nd-lam]}$$

$$\text{Let } T'_u = T_u.$$

$$\sigma(T_u[e_u]) = T_n[e_n] = T_n[(\lambda(x) e'_n)] \text{ by assumption.}$$

$$\text{By lem. 8, } \sigma T_u \in \text{CTX and } \sigma e_u \in \text{ULC-H.}$$

$$e_u = (\lambda(x) e'_u) \text{ such that } \sigma e'_u = e'_n \text{ by definition of substitution.}$$

Let  $\xi$  in  $\text{supp}(\sigma)$  be the type variable associated with this  $[\text{nd-}\tau\beta]$  step. Note that it

is fresh because it is distinct from any type variable introduced before (for any other [nd- $\tau\beta$ ] step).

$T_u[(\lambda(x) e'_u)] \mapsto_u T_u[(\rightarrow \xi \{x \mapsto \xi\} e'_u)]$  by [ch-lam].

$\sigma(T_u[(\rightarrow \xi \{x \mapsto \xi\} e'_u)]) = T_n[(\rightarrow \tau \{x \mapsto \tau\} e'_n)]$  because  $\sigma(T_u[(\rightarrow \xi \{x \mapsto \xi\} e'_u)]) = (\sigma T_u)[(\rightarrow \sigma\xi \{x \mapsto \sigma\xi\} \sigma e'_u)] = T_n[(\rightarrow \tau \{x \mapsto \xi\} e'_n)]$  (by definition of  $\sigma$  and assumption).

**Case [nd- $\tau\beta$ ]:**

**Assume:**

- $T_n[(\@ (\rightarrow \tau' \tau) \tau')] \mapsto_n T_n[\tau]$
- $\sigma(T_u[e_u]) = T_n[(\@ (\rightarrow \tau' \tau) \tau')]$

**To show:**  $T_u[e_u] \mapsto_u T'_u[e'_u]$  and  $\sigma(T'_u[e'_u]) = T_n[e'_n]$

$\sigma(T_u[e_u]) = (\sigma T_u)[\sigma e_u] = T_n[(\@ (\rightarrow \tau' \tau) \tau')]$

By lem. 8,  $\sigma T_u \in \text{TYPE}$  and  $\sigma e_u \in \text{ULC-H}$ . By unique decomposition,  $\sigma T_u = T_n$  and  $\sigma e_u = (\@ (\rightarrow \tau' \tau) \tau')$ . By definition of substitution,  $e_u = (\@ e''_u e'''_u)$  where (1)  $\sigma e''_u = (\rightarrow \tau' \tau)$  and (2)  $\sigma e'''_u = \tau'$ .

By definition of substitution,  $e''_u$  and  $e'''_u \in \text{TYPE-V}$ . Let  $\tau'_u = e''_u$  and  $\tau''_u = e'''_u$ . (1) and (2) become (1\*)  $\sigma \tau'_u = (\rightarrow \tau' \tau)$  and (2\*)  $\sigma \tau''_u = \tau'$  respectively. Combining (1\*) and (2\*) yields (3)  $\sigma \tau'_u = (\rightarrow \sigma \tau''_u \tau)$ .

Let  $\zeta$  be the fresh type variable associated with this [nd- $\tau\beta$ ] step. By definition of instantiating substitution,  $\{\zeta \mapsto \tau\} \in \sigma$ . Thus, (4)  $\sigma \zeta = \tau$ . Combining (3) and (4) gives  $\sigma \tau'_u = (\rightarrow \sigma \tau''_u \sigma \zeta) = \sigma(\rightarrow \tau''_u \zeta)$ .

By definition (of unifier),  $\sigma$  is a unifier of  $\tau'_u$  and  $(\rightarrow \tau''_u \zeta)$ . Hence,  $\tau'_u$  and  $(\rightarrow \tau''_u \zeta)$  are unifiable. There exists a mgu  $\theta$ . By lem. 14, (**unify**  $\tau'_u (\rightarrow \tau''_u \zeta) T_u[\zeta]) \mapsto_u^* \theta(T_u[\zeta])$ .

$$\begin{aligned} T_u[e_u] &\mapsto_u (\mathbf{unify} \tau'_u (\rightarrow \tau''_u \zeta) T_u[\zeta]) \quad [\text{ch-}\tau\beta] \\ &\mapsto_u^* \theta(T_u[\zeta]) \quad \text{lem. 14} \\ &= (\theta T_u)[\theta \zeta] \end{aligned}$$

Let  $T'_u = \theta T_u$  and  $e'_u = \theta \zeta$ . Thus,  $T_u[e_u] \mapsto_u^* T'_u[e'_u]$ .

By the definition of mgu, there exists  $\delta$  such that  $\sigma = \delta\theta$ .  $\sigma(T'_u[e'_u]) = \sigma((\theta T_u)[\theta\zeta]) = (\sigma\theta T_u)[\sigma\theta\zeta]$ . Recall that  $\theta$  is idempotent because of the occurs check. So,  $\sigma\theta = \delta\theta\theta = \delta\theta = \sigma$ . Thus  $(\sigma\theta T_u)[\sigma\theta\zeta] = (\sigma T_u)[\sigma\zeta] = (\sigma T_u)[\tau] = T_n[e'_n]$ .

□

**Theorem 3 (Completeness of Curry/Hindley System).** *If  $e_n \mapsto_n^* \tau$ , then  $\exists \tau_u$  and  $\sigma$  such that  $e_n \mapsto_u^* \tau_u$  and  $\sigma\tau_u = \tau$ .*

*Proof.* I prove completeness by showing completeness for a single step of  $\mapsto_n$  and generalizing the statement to work under all typechecking contexts  $T_n \in \text{CTX}$  in lem. 12. The intuition is that all  $\mapsto_n$  redexes are also  $\mapsto_u$  redexes. In particular,  $\text{CTX} \subseteq \text{CTX-V}$ ,  $\text{ULC-H} \subseteq \text{ULC-HV}$ , and  $\text{TYPE} \subseteq \text{TYPE-V}$ . This lemma is simply the transitive closure of lem. 12 with  $T_n = T_u = \square$  (the empty context) and  $e_u = e_n$ . □

The proof of correctness of the unification reductions requires lemma 17 in order to formalize the relation between the most general unifier of an application and the most general unifiers of the subexpressions in the application. Let  $\varepsilon$  be the identity substitution.

Before I continue, I must make some observations about how the  $\mapsto_f$  reductions work and what quantity is decreasing with each reduction. The  $[\text{ch-}\tau\beta]$  rule generates unify prefixes as defined in definition 4 which must be reduced immediately before type inference can continue. A unify reduction may produce more unify prefixes. For example,  $[\text{ch-u-dist}]$  and  $[\text{ch-u-orient}]$  both reduce prefixes to more prefixes. However, each unification reduction except for the administrative  $[\text{ch-u-orient}]$  rule must decrease either the number of type variables ( $[\text{ch-u-inst}]$ ) or the size of the types ( $[\text{ch-u-const}]$  and  $[\text{ch-u-dist}]$ ). The  $[\text{ch-u-orient}]$  must work in tandem with the  $[\text{ch-u-inst}]$ , That is to say, if a term reduces by  $[\text{ch-u-orient}]$ , then that reduct must reduce by  $[\text{ch-u-inst}]$ . Consequently, a system with  $[\text{ch-u-orient}]$  and  $[\text{ch-u-inst}]$  is equivalent to a system where a rule that does both  $[\text{ch-u-orient}]$  and  $[\text{ch-u-inst}]$  when the type variable is on the right hand side. Thus, without loss of generality, I can use a system with only the  $[\text{ch-u-inst}]$  rule. I prove this property in lemma 13 using a lexicographic ordering on the number of type variables and size of all the types in the prefix (def. 5). Let  $\mapsto_f$  be the union of  $[\text{ch-u-const}]$ ,  $[\text{ch-u-dist}]$ , and  $[\text{ch-u-inst}]$ .

**Definition 4 (Prefixes).** The prefix of a term  $p$  is the context  $D$  such that  $D[e_u] = p$ :

$$D ::= (\mathbf{unify} \tau_u \tau'_u D)$$

|  $\square$

Define a function  $\text{size}_{\text{type}}$  that yields the size of a type:

$$\text{size}_{\text{type}}(\tau) = \begin{cases} 1 + \text{size}_{\text{type}}(\tau_u) + \text{size}_{\text{type}}(\tau'_u) & \text{if } \tau = (\rightarrow \tau_u \tau'_u) \\ 1 & \text{otherwise} \end{cases}$$

Define a function  $\text{size}_p$  that yields the sum of the sizes of every type in all the prefixes.

$$\text{size}_{\text{prefix}}(p) = \begin{cases} \text{size}_{\text{type}}(\tau'_u) + \text{size}_{\text{type}}(\tau''_u) + \text{size}_{\text{prefix}}(p') & \text{if } p = (\mathbf{unify} \tau'_u \tau''_u p') \\ 0 & \text{otherwise} \end{cases}$$

**Definition 5 (Lexicographic Ordering on Unify Expressions).**

Define a lexicographic ordering  $\prec$ :

$p \prec p'$  if and only if

- $FV(D) \subset FV(D')$  or
- $FV(D) = FV(D')$  and  $\text{size}_{\text{prefix}}(p) < \text{size}_{\text{prefix}}(p')$

where  $D[e_u] = p$  and  $D'[e'_u] = p'$ .

The  $\prec$  ordering is well-founded.

**Lemma 13 (Prefix Sizes Decreases).** If  $p \mapsto_f p'$ , then  $p' \prec p$ .

*Proof.* •  $(\mathbf{unify} \tau_1 \tau_1 p') \mapsto_f p'$ .

Either  $FV(\tau_1) \cup FV(D') \supset FV(D')$  or  $FV(\tau_1^1) \cup FV(D') = FV(D')$  (because  $\cup$  is monotonic):

– Subcase  $FV(\tau_1^1) \cup FV(D') \supset FV(D')$ :

$$\begin{aligned} FV(\mathbf{unify} \tau_u^1 \tau_u^1 D') &= FV(\tau_u^1) \cup FV(D') \\ &\supset FV(D') \end{aligned}$$

– Subcase  $FV(\tau_u^1) \cup FV(D') = FV(D')$ :  $FV(\tau_u^1) \cup FV(D') = FV(D')$  and

$$\begin{aligned} & \text{size}(\mathbf{unify} \tau_u^1 \tau_u^1 D') \\ &= \text{size}_{\text{type}}(\tau_u^1) + \text{size}(D') \\ &> \text{size}(D') \end{aligned}$$

- $(\mathbf{unify} \xi \tau_u^1 p) \mapsto_f \{\xi \mapsto \tau_u^1\} p$ .

$$\begin{aligned} FV(\{\xi \mapsto \tau_u^1\} D) &\subseteq FV(D) \setminus \{\xi\} \cup FV(\tau_u^1) \\ &= (FV(D) \cup FV(\tau_u^1)) \setminus \{\xi\} && \text{because } \xi \notin FV(\tau_u^1) \\ &\subseteq FV(D) \cup FV(\tau_u^1) \\ &\subset \{\xi\} \cup FV(\tau_u^1) \cup FV(D) \\ &= FV(\mathbf{unify} \xi \tau_u^1 D) \end{aligned}$$

- $(\mathbf{unify} (\rightarrow \tau_u^1 \tau_u^2) (\rightarrow \tau_u^3 \tau_u^4) p'') \mapsto_f (\mathbf{unify} \tau_u^1 \tau_u^3 (\mathbf{unify} \tau_u^2 \tau_u^4 p''))$

$$\begin{aligned} & FV(\mathbf{unify} (\rightarrow \tau_u^1 \tau_u^3) (\rightarrow \tau_u^2 \tau_u^4) D'') = FV(\mathbf{unify} \tau_u^1 \tau_u^3 (\mathbf{unify} \tau_u^2 \tau_u^4 D'')) \text{ and} \\ & \text{size}(\tau_u^1) + \text{size}(\tau_u^3) + \text{size}(\tau_u^2) + \text{size}(\tau_u^4) + \text{size}(D'') < 1 + \text{size}(\tau_u^1) + \text{size}(\tau_u^3) + 1 + \\ & \text{size}(\tau_u^2) + \text{size}(\tau_u^4) + \text{size}(D''). \end{aligned}$$

□

**Lemma 14 (Correctness of Unification).** *If  $\tau_u$  and  $\tau'_u$  are unifiable, then  $(\mathbf{unify} \tau_u \tau'_u p) \mapsto_f^* \sigma p$  where  $\sigma$  is a most general unifier of  $\tau_u$  and  $\tau'_u$ .*

*Proof (by induction on the lexicographic ordering  $\prec$  of the term  $(\mathbf{unify} \tau_u \tau'_u p)$ ).*

Base case  $p \mapsto_f^0 p$ : This case is vacuously true (i.e., there are no types to unify).

Inductive Cases:

Assume: For size  $n$  prefix,  $(\mathbf{unify} \tau''_u \tau'''_u p') \mapsto_f^* \sigma' p'$  where  $\sigma'$  is the mgu of  $\tau''_u$  and  $\tau'''_u$ .

To show: For size  $m > n$  prefix,  $(\mathbf{unify} \tau_u \tau'_u p) \mapsto_f^* \sigma p$  where  $\sigma$  is a mgu of  $\tau_u$  and  $\tau'_u$ .

Case  $(\mathbf{unify} \tau_u \tau_u p) \mapsto_f p$ .

$\varepsilon\tau_u = \varepsilon\tau_u$ . Hence,  $\varepsilon$  is a unifier. Because any unifier  $\theta = \lambda\varepsilon$  where  $\lambda = \theta$ ,  $\varepsilon$  is also a most general unifier.

Case  $(\mathbf{unify} \xi \tau_u p) \mapsto_f \{\xi \mapsto \tau_u\}p$

$\{\xi \mapsto \tau_u\}\xi = \tau_u = \{\xi \mapsto \tau_u\}\tau_u$  by substitution.  $\{\xi \mapsto \tau_u\}$  is a unifier. Because for any unifier  $\theta$  ( $\theta\xi = \theta\tau_u$ ),  $\theta(\xi) = \theta\tau_u = \lambda\{\xi \mapsto \tau_u\}\tau_u = \lambda\tau_u$  where  $\lambda = \theta|_{dom \text{ excl. } \xi}$  because  $\xi \notin \tau_u$  by the occurs check.  $\{\xi \mapsto \tau_u\}$  is a most general unifier by definition.

Case  $(\mathbf{unify} (\rightarrow \tau_u^1 \tau_u^2) (\rightarrow \tau_u^3 \tau_u^4) p) \mapsto_f (\mathbf{unify} \tau_u^1 \tau_u^3 (\mathbf{unify} \tau_u^2 \tau_u^4) p) \mapsto_f^* \sigma_1 (\mathbf{unify} \tau_u^2 \tau_u^4) p \mapsto_f^* \sigma_2 \sigma_1 p$ .

By induction,  $\sigma_1$  is a most general unifier for  $\tau_u^1$  and  $\tau_u^3$ , and  $\sigma_2$  is a most general unifier for  $\tau_u^2$  and  $\tau_u^4$ . By lemma 17,  $\sigma = \sigma_2\sigma_1$  is a most general unifier for  $(\rightarrow \tau_u^1 \tau_u^2)$  and  $(\rightarrow \tau_u^3 \tau_u^4)$ .

□

It is also important to establish that not only do the unification reductions obtain the correct answer, they also terminate. With lemma 13, I can easily prove the termination of unification.

**Lemma 15 (Termination of Unification).** *If  $\tau_u^1$  and  $\tau_u^2$  are unifiable, then the reduction  $\mapsto_f$  terminates for term  $(\mathbf{unify} \tau_u^1 \tau_u^2 p)$  in a finite number of steps.*

*Proof.* Suppose that the reductions do not terminate in a finite number of steps. The term decreases forever according to the lexicographic ordering  $\prec$ . This is a contradiction, however, because  $\prec$  is a well-founded ordering (i.e., no term can have less than 0 unique variables and 0 type size). Thus, the reductions must terminate in a finite number of steps. □

If the types are unifiable (i.e., unification does not get stuck), then the final reduct must always be some substitution applied to the term being unified over. Simply put, lemma 16 shows that unification applies a type variable substitution to the whole term and does nothing else.

**Lemma 16 (Well-formedness of Unification Reduct).**  *$(\mathbf{unify} \tau_u^1 \tau_u^2 p) \mapsto_f^* p'$  such that  $p' = \sigma p$ .*

*Proof.* (by induction on the number of reduction steps)

- $(\mathbf{unify} \tau_u^1 \tau_u^1 p') \mapsto_f p'$ .
- $(\mathbf{unify} \xi \tau_u^1 p) \mapsto_f \{\xi \mapsto \tau_u^1\}p$
- $(\mathbf{unify} (\rightarrow \tau_u^1 \tau_u^2) (\rightarrow \tau_u^3 \tau_u^4) p) \mapsto_f (\mathbf{unify} \tau_u^1 \tau_u^3 (\mathbf{unify} \tau_u^2 \tau_u^4 p))$

$$\begin{aligned}
 & (\mathbf{unify} \tau_u^1 \tau_u^3 (\mathbf{unify} \tau_u^2 \tau_u^4 p)) \\
 & \quad \mapsto^* \sigma_1 (\mathbf{unify} \tau_u^2 \tau_u^4 p) && \text{by induction} \\
 & \quad \mapsto^* \sigma_2 \sigma_1 p && \text{by induction}
 \end{aligned}$$

$$p' = \sigma p \text{ where } \sigma = \sigma_2 \sigma_1.$$

□

Lemma 17 is due to Robinson [30]. It establishes how to get a unifier for a term from unifiers of subterms.

**Lemma 17.** *If  $\sigma_1$  is a mgu of  $\tau_u^1$  and  $\tau_u^2$  and  $\sigma_2$  is a mgu of  $\sigma_1 \tau_u^3$  and  $\sigma_1 \tau_u^4$ , then  $\sigma_2 \sigma_1$  is a mgu of  $(\rightarrow \tau_u^1 \tau_u^3)$  and  $(\rightarrow \tau_u^2 \tau_u^4)$ .*

*Proof.* For any unifier  $\theta$  of  $(\rightarrow \tau_u^1 \tau_u^2)$  and  $(\rightarrow \tau_u^3 \tau_u^4)$ , certainly the following equations hold:

$$\theta \tau_u^1 = \theta \tau_u^3 \tag{B.1}$$

$$\theta \tau_u^2 = \theta \tau_u^4 \tag{B.2}$$

Observe that  $\theta$  is also a unifier of  $\tau_u^1$  and  $\tau_u^3$ , and a unifier of  $\tau_u^2$  and  $\tau_u^4$ . Therefore,  $\theta = \lambda_1 \sigma_1$ . Substituting into equation B.2 yields  $\lambda_1 \sigma_1 \tau_u^2 = \lambda_1 \sigma_1 \tau_u^4$ . By definition,  $\lambda_1$  is a unifier of  $\sigma_1 \tau_u^2$  and  $\sigma_1 \tau_u^4$ . Because  $\sigma_2$  is the mgu of  $\sigma_1 \tau_u^2$  and  $\sigma_1 \tau_u^4$ ,  $\lambda_1 = \lambda_2 \sigma_2$ . It follows that  $\theta = \lambda_1 \sigma_1 = \lambda_2 \sigma_2 \sigma_1$ . By definition,  $\sigma = \sigma_2 \sigma_1$  is a mgu of  $(\rightarrow \tau_u^1 \tau_u^2)$  and  $(\rightarrow \tau_u^3 \tau_u^4)$ . □

## APPENDIX C

### HINDLEY-MILNER SYSTEM PROOFS

The Hindley-Milner rewrite system is sound and complete relative to Algorithm  $\mathcal{W}$ . The Hindley-Milner system can be turned into a machine in order to better match Algorithm  $\mathcal{W}$ . I show that the machine (Fig. C.2) is sound and complete relative to Algorithm  $\mathcal{W}$  in lemmas 12 and 17. Fig. C.3 defines Algorithm  $\mathcal{W}$  as given by Lee and Yi [18]. The correctness of Danvy’s method [10] for obtaining machines from rewrite systems establishes the soundness and completeness of the rewrite system relative to the machine (see lem. 18).

ULCL expressions are ULC expressions plus a **let** form. The HM reduction system reduces ULCL expressions whose  $\lambda$  and **let** forms have been annotated with the number of enclosing  $\lambda$ s inclusive. App. D gives a well-annotation functions  $\mathcal{A}$  and  $d_\lambda$  for ULCL expressions.  $\mathcal{A}$  annotates a ULCL expression that has no context.  $d_\lambda$  annotates a ULCL expression that is being typechecked under a type environment  $\Gamma$ . The  $\mathcal{E}_a$  function erases  $\lambda$ -depth annotations.  $\tilde{e}$  denotes ULCL expressions that do not have  $\lambda$ -depth annotation.

**Definition 1 (Well Annotation).** *A ULCL expression  $e$  is well-annotated if  $\mathcal{A}(\mathcal{E}_a(e)) = e$ .*

The machine in Fig. C.2 is similar to that of the TC system. Machine states are of the form  $(c, \Gamma, \Sigma, k)$  where  $c$  is the control expression (a unify problem  $u$ , polytype  $\rho$ , or ULCL-A expression  $e$ ),  $\Gamma$  is the type environment with ranked type variables,  $\Sigma$  is the substitution list, and  $k$  is the typechecking context stack. The substitution list keeps a cumulative stack of all the substitutions produced by unify thus far. To capture the substitutions produced by each HM reduction subsequence, I keep a list of them. The difference between the substitution list register before and after reducing a subexpression corresponds to the substitution produced when  $\mathcal{W}$  typechecks that subexpression. New substitutions that are produced are applied eagerly, but I keep a record of the substitutions applied in a list in order to easily factor the current type environment into the original type environment and substitutions applied to that type environment. I need this factoring for matching up a reduction sequence against the corresponding Algorithm  $\mathcal{W}$  call.

$e ::= x \mid \text{number} \mid (\lambda^d (x \tau) e) \mid (@ e e) \mid (\mathbf{let}^d (x e) e)$	ULCL-A
$\tau ::= \text{num} \mid (\rightarrow \tau \tau) \mid \xi^d$	TYPE-R
$\tau_p ::= \text{num} \mid (\rightarrow \tau_p \tau_p) \mid \xi^d \mid \alpha$	POLYTYPE
$\rho ::= \forall \vec{\alpha}. \tau_p$	generalized type
$u ::= (\mathbf{unify} \tau \tau u) \mid \tau$	unify problem
$c ::= u \mid \rho \mid e$	control expression
$\Gamma ::= \bullet \mid \Gamma[x : \rho] \mid \Gamma[x : \tau]$	TYENV-R
$\tilde{\Gamma} ::= \bullet \mid \tilde{\Gamma}[x : \tilde{\rho}] \mid \tilde{\Gamma}[x : \tilde{\tau}]$	TYENV
$d, m, n, q ::= 0 \mid 1 \mid \dots \mid \infty$	ranking
$\xi, \phi, \psi$	type variables
$\Sigma ::= \bullet \mid \sigma :: \Sigma$	Substitution Lists
$f ::= (@ \square e) \mid (@ \tau \square) \mid (\rightarrow \tau \square) \mid (\mathbf{let}^d (x \square) e) \mid (\mathbf{let}^d \square)$	context frame
$k ::= \bullet \mid f :: k$	context stack
$\mathcal{G}_d(\tau) = \{\xi^m \in \text{ftv}(\tau) \mid d < m\}$	$\mathcal{G}_\Gamma(\tau) = \{\xi^m \in \text{ftv}(\tau) \mid \xi^m \in \text{ftv}(\Gamma)\}$
$\mathcal{P}_d(\tau) = \forall \vec{\alpha} \{\mathcal{G}_d(\tau) \mapsto \vec{\alpha}\} \tau$ ( $\vec{\alpha}$ fresh)	$\mathcal{P}_\Gamma(\tau) = \forall \vec{\alpha} \{\mathcal{G}_\Gamma(\tau) \mapsto \vec{\alpha}\} \tau$ ( $\vec{\alpha}$ fresh)

ULCL expressions are exactly the same as ULCL-A expressions except with ranking annotations erased.

Initial Machine States are of the form  $(e, \bullet, \bullet, \bullet)$ .

Machine states are of the form  $(c, \Gamma, \Sigma, k)$ .

Figure C.1: HM Machine States

Type variables in  $\Gamma$  derive their  $\lambda$ -depth rank from the  $\lambda$ -depth of the binding with the first occurrence in  $\Gamma$  of the said type variable. As such, when a type variable is first introduced into  $\Gamma$ , it inherits that depth annotation of its binding  $\lambda$ . These type variable ranks may be propagated into the control and context by means of type substitutions and types.  $\mathcal{E}_r$  denotes the rank erasure function. It is extended to work on types, type substitutions, and type environments as is usual. I distinguish type environments, type substitutions, types, and bounded types that do not contain ranks from their ranked variants by  $\tilde{\Gamma}$ ,  $\tilde{\sigma}$ ,  $\tilde{\tau}$ , and  $\tilde{\rho}$  respectively. After I establish some facts about ranking, I will define ranking functions that rank type variables based on a type environment (in def. 9 below).

Several machine rules such as [pm-app], [pm-app-left], [pm-app-right], [pm-arr], [pm-let], [pm-let-def], and [pm-let-body] maintain the typechecking context, refocusing the machine on the next subexpression to typecheck. The [pm-lam] and [pm-let-def] push a new  $\lambda$ -bound variable to type and a **let**-bound variable to polytype binding respectively

$((@ e e'), \Gamma, \Sigma, k) \mapsto_{pm} (e, \Gamma, \Sigma, (@ \square e')::k)$	[pm-app]
$(\tau, \Gamma, \Sigma, (@ \square e')::k) \mapsto_{pm} (e', \Gamma, \Sigma, (@ \tau \square)::k)$	[pm-app-left]
$(\tau', \Gamma, \Sigma, (@ \tau \square)::k) \mapsto_{pm} ((@ \tau \tau'), \Gamma, \Sigma, k)$	[pm-app-right]
$((\lambda^d (x) e), \Gamma, \Sigma, k) \mapsto_{pm} (e, \Gamma[x : \xi^d], \Sigma, (\rightarrow \xi^d \square) :: k)$ $\xi$ is fresh	[pm-lam]
$(\tau', \Gamma[x : \tau], \Sigma, (\rightarrow \tau \square)::k) \mapsto_{pm} ((\rightarrow \tau \tau'), \Gamma, \Sigma, k)$	[pm-arr]
$(x, \Gamma, \Sigma, k) \mapsto_{pm} (\Gamma(x), \Gamma, \Sigma, k)$	[pm-var]
$(number, \Gamma, \Sigma, k) \mapsto_{pm} (num, \Gamma, \Sigma, k)$	[pm-num]
$((@ \tau \tau'), \Gamma, \Sigma, k) \mapsto_{pm} ((\mathbf{unify} \tau (\rightarrow \tau' \xi^\infty) \xi^\infty), \Gamma, \Sigma, k)$ $\xi$ is fresh	[pm- $\tau\beta$ ]
$((\mathbf{let}^d (x e) e'), \Gamma, \Sigma, k) \mapsto_{pm} (e, \Gamma, \Sigma, (\mathbf{let}^d (x \square) e')::k)$	[pm-let]
$(\tau, \Gamma, \Sigma, (\mathbf{let}^d (x \square) e')::k) \mapsto_{pm} (e', \Gamma[x : \mathcal{P}_d(\tau)], \Sigma, (\mathbf{let}^d \square)::k)$	[pm-let-def]
$(\tau, \Gamma[x : \rho], \Sigma, (\mathbf{let}^d \square)::k) \mapsto_{pm} (\tau, \Gamma, \Sigma, k)$	[pm-let-body]
$(\forall \vec{\alpha} \tau_p, \Gamma, \Sigma, k) \mapsto_{pm} (\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p, \Gamma, \Sigma, k)$ $\vec{\xi}^\infty$ are fresh type variables	[pm-poly]
$((\mathbf{unify} \xi^d \tau), \Gamma, \Sigma, k) \mapsto_{pm} (\sigma u, \sigma \Gamma, \sigma :: \Sigma, \sigma k)$ $\sigma = \mathcal{L}(\tau, d) \circ \{\xi^d \mapsto \tau\}$	[pm-u-mv]
$((\mathbf{unify} \tau \tau u), \Gamma, \Sigma, k) \mapsto_{pm} (u, \Gamma, \Sigma, k)$	[pm-u-id]
$((\mathbf{unify} (\rightarrow \tau_1 \tau_2) (\rightarrow \tau_3 \tau_4) u), \Gamma, \Sigma, k)$ $\mapsto_{pm} ((\mathbf{unify} \tau_1 \tau_3 (\mathbf{unify} \tau_2 \tau_4 u)), \Gamma, \Sigma, k)$ $(\rightarrow \tau_1 \tau_2) \neq (\rightarrow \tau_3 \tau_4)$	[pm-u-arr]
$((\mathbf{unify} \tau \xi^d u), \Gamma, \Sigma, k) \mapsto_{pm} ((\mathbf{unify} \xi^d \tau u), \Gamma, \Sigma, k)$ where $\tau$ is not a type variable	[pm-u-orient]

Figure C.2: HM Machine Rules (pm - Polymorphic Rewrite Machine)

$$\begin{array}{lll}
\mathcal{W}(\Gamma, \text{number}) & = & (\varepsilon, \text{num}) \\
\mathcal{W}(\Gamma, x) & = & (\varepsilon, \{\vec{\alpha} \mapsto \vec{\xi}\} \tau_p) & \Gamma(x) = \forall \vec{\alpha}. \tau_p, \vec{\xi} \text{ fresh} \\
\mathcal{W}(\Gamma, x) & = & (\varepsilon, \tau) & \Gamma(x) = \tau \\
\mathcal{W}(\Gamma, (\lambda (x) e)) & = & \mathbf{let} (\sigma, \tau) = \mathcal{W}(\Gamma[x : \xi], e) & \xi \text{ fresh} \\
& & \text{in } (\sigma, (\rightarrow (\sigma \xi) \tau)) \\
\mathcal{W}(\Gamma, (@ e e')) & = & \mathbf{let} (\sigma, \tau) = \mathcal{W}(\Gamma, e) \\
& & (\sigma', \tau') = \mathcal{W}(\sigma\Gamma, e') \\
& & \sigma'' = \mathcal{U}(\sigma' \tau, (\rightarrow \tau' \xi)) & \xi \text{ fresh} \\
& & \text{in } (\sigma'' \sigma' \sigma, \sigma'' \xi) \\
\mathcal{W}(\Gamma, (\mathbf{let} (x e) e')) & = & \mathbf{let} (\sigma, \tau) = \mathcal{W}(\Gamma, e) \\
& & (\sigma', \tau') = \mathcal{W}(\sigma\Gamma[x : \mathcal{P}'_{\sigma\Gamma}(\tau)], e') \\
& & \text{in } (\sigma' \sigma, \tau')
\end{array}$$

$\Gamma$  here is the unranked  $\Gamma$  (with ranking annotations erased). If  $\mathcal{W}$  is applied to a ranked  $\Gamma$  and ranked ULCL-R expression, it simply erases all ranking annotation and proceeds as if it were given an unranked  $\Gamma$  and ULCL expression.

Figure C.3: Algorithm  $\mathcal{W}$  from Lee and Yi [18]

onto the type environment so that the bodies of these expressions can be typechecked under the new mapping. After the  $\lambda$  or  $\mathbf{let}$  body is completely typechecked, the [pm-arr] and [pm-let-body] rules pop the  $\lambda$ - or  $\mathbf{let}$ -binding respectively off of the type environment so that remainder of the expression will be typechecked under a type environment binding only the variables that are still in scope. The rule [pm-var] serves the same role as [tm-var] in the TC machine, looking up the variable in the type environment.

The [pm-u-mv] rule applies the unification substitution and keeps that unification substitution around explicitly by pushing it onto the  $\Sigma$  unification substitution list. Some rules such as [pm-num], [pm- $\tau\beta$ ], [pm-poly], [pm-u-id], [pm-u-arr], and [pm-u-orient] do not alter any machine register other than the control. They serve exactly the same role as the analogous reduction rules.

**(Substitution and Substitution List Notation)** I express substitution composition by juxtaposing substitutions together. For example,  $\sigma'' \sigma' \sigma e$  denotes  $\sigma''(\sigma'(\sigma e))$ . In the absence of an expression, type environment, or context stack,  $\sigma'' \sigma' \sigma$  denotes the right associative composition of those substitutions,  $\sigma'' \sigma' \sigma = \sigma'' \circ (\sigma' \circ \sigma)$ . I use  $\mathcal{C}$  to denote the substitution composition operator. I also extend the application of substitutions to expressions, type environments, and contexts in the usual way. The  $\bullet$  symbol when used in place of a sub-

stitution list denotes the empty substitution list. When  $\bullet$  is applied, it should be interpreted as applying the identity substitution  $\varepsilon$ . Note that  $\mathcal{C}(\bullet) = \varepsilon$ . For notational convenience,  $\widehat{\Sigma}$  will be used as a shorthand for  $\mathcal{C}(\Sigma)$ . Juxtaposition of substitution lists in the substitution list machine register denotes the concatenation of those lists.

The HM machine differs from the TC machine in that any rewrite step that does a unification (i.e., may produce a unification substitution) will need to apply a substitution to the context stack. Observe that the HM system unification reductions are identical to the CH unification reductions except all unification reductions propagate type variable ranks along with type variables and the [pm-u-mv] rule also applies a limit substitution  $\mathcal{L}(\tau, d)$  which only alters the type variable ranks. Because the ranks do not affect the correctness of the unification, the original unification correctness theorem (thm. B.14 holds for the HM machine unification reductions. Because the correctness of unification has been established in the previous appendix, I will treat **unify** as equivalent to the atomic metalanguage operation  $\mathcal{U}$  that produces type substitutions in this system. The unification reduction for the machine [pm-u-mv] applies the unification substitutions to the control, type environment, and context stack, and also adds the unification substitutions to the substitution list. I apply the unification substitutions to the context stack to propagate the substitution information to any type variable occurrences in the context stack. Finally, the [pm-u-mv] rule looks somewhat peculiar. Although the **unify** is wholly in the control, it still has a global substitution effect. In effect,  $(\sigma\tau, \sigma\Gamma, \sigma :: \Sigma, \sigma k)$  (where  $\sigma = \mathcal{U}(\xi^d, \tau')$ ) corresponds to  $(\mathbf{unify} \xi^d \tau' k[\tau])$ . The context stack frames of this machine contain only non-hybrid expressions and types as subexpressions (the frame itself may be hybrid) because types are only introduced in the type environment and when the control is fully typechecked. This machine completely models the HM reduction system.

This machine serves as the basis for the soundness result, lem. 12. The control for the machine states can be hybrid because [pm-app-right] gives a machine state with a hybrid expression  $(@ \tau \tau')$  for the control. The soundness lemma is only concerned about non-hybrid ULCL-A expressions  $e$  because  $\mathcal{W}$  is only defined on non-hybrid expressions and [pm- $\tau\beta$ ] immediately eliminates the hybrid  $(@ \tau \tau')$  expressions. Moreover, all expressions in the context stack are either non-hybrid ULCL-A expressions or types because the machine does not apply program variable to type substitutions in [pm-lam], but instead sim-

ply extends the type environment. In general, the pm reduction sequence for a given context stack may end with a  $\tau_p$  control because the type environment may contain polymorphic type schemas. Because  $\mathcal{W}$  is only defined on expressions without depth annotating, I need a notion of depth annotation erasure. Let  $\mathcal{E}_r$  be the depth ranking erasure function. Now I can express the soundness lemma. The proof writeup will use the shorthand tv for type variable.

For notational brevity, I introduce the notion of “valid machine states” so that I can refer to the redexes of sensible machine states in the statement of the following lemmas.

**Definition 2 (Valid Machine States (VMS)).** *A machine state  $(c, \Gamma, \Sigma, k)$  is valid if it is reachable from some valid initial machine state  $(e, \bullet, \bullet, \bullet)$  (i.e.,  $(e, \bullet, \bullet, \bullet) \mapsto_{pm}^* (c, \Gamma, \Sigma, k)$ ) where  $e \in ULCL$ .*

I proceed by proving that for all type variables found in controls from VMS the generalizability criteria from in the definitions of  $\mathcal{G}$  and  $\mathcal{G}'$  are equivalent. Underscores ( $\_$ ) indicate that the values of those particular registers are irrelevant.

**Lemma 1.** *For all  $s = (\tau, \_ , \_ , \_ ) \in \text{VMS}$ ,*

$$\forall \xi^m \in \text{ftv}(\tau) \Rightarrow (d < m \Leftrightarrow \xi^m \notin \text{ftv}(\Gamma))$$

*Proof.*

( $\Rightarrow$ ): Cor. 1 holds for all  $s \in \text{VMS}$  by lem. 9.

( $\Leftarrow$ ): Inv. 2 holds for all  $s \in \text{VMS}$  by lem. 9. □

To introduce the necessary invariants below to prove lem. 1, I first need a couple of crucial definitions concerning type environments. Within a type environment, I distinguish between  $\lambda$ -bindings which are introduced by [pm-lam] (i.e.,  $[x : \tau]$ ) and **let**-bindings that are introduced by [pm-let-def] (i.e.,  $[x : \rho]$ ).  $\lambda$ -bindings bind program variables to monotypes whereas **let**-bindings bind program variables to polytypes including degenerate polytypes with no generalized type variables (e.g.,  $\forall().(\rightarrow \text{num num})$ ).

**Definition 3.**  $|\tilde{\Gamma}| = \text{the length of } \tilde{\Gamma}$ .

**Definition 4.**  $\mathcal{D}(\Gamma) = \# \lambda$  bindings in  $\Gamma$ .

The  $\mathcal{D}$  notation will also be used to refer to the same quantity in an unranked  $\tilde{\Gamma}$ .

**Definition 5.**  $\Gamma_\lambda$  denotes the list of  $\lambda$ -bindings in  $\Gamma$ .

Again, the same notation applies to unranked  $\tilde{\Gamma}$ .

It will be necessary to identify the position of the  $\lambda$ -bindings relative to each other (but not **let**-bindings) in the type environment. I count off  $\lambda$ -bindings in a type environment from left to right. For example, in  $[x_1 : \text{num}][x_2 : \forall \alpha. (\rightarrow \alpha \alpha)][x_3 : \text{num}][x_4 : \text{num}]$ , the  $x_1$  and  $x_3$  bindings are the 1st and 2nd  $\lambda$ -bindings respectively. This convention gives each  $\lambda$ -binding a unique position relative to other  $\lambda$ -bindings in a type environment. I am particularly interested in the leftmost  $\lambda$ -binding, which I may also refer to as the **earliest**  $\lambda$ -binding in a type environment, because type variable occurrences in type environment bindings to the right (later bindings) do not affect generalizability. The position of **let**-bindings relative to other  $\lambda$ - or **let**-bindings does not play a role in the proof of the invariants. Because only occurrences in  $\lambda$ -bindings constrain generalizability, I will not need the position of **let**-bindings. Intuitively,  $\tilde{\mathcal{V}}(\xi, \Gamma)$  is the position (relative to other  $\lambda$ -bindings) of leftmost occurrence of  $\xi$  in a  $\lambda$ -binding in  $\Gamma$ .  $\mathcal{V}$  is a variant of  $\tilde{\mathcal{V}}$  that works on ranked type variables.

**Definition 6 (Type Variable Rank).**

$$\tilde{\mathcal{V}}(\xi, \bullet) = \infty$$

$$\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}[x : \tilde{\tau}]) = \min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}), m) \text{ where } m = \mathcal{D}(\tilde{\Gamma}) + 1 \text{ if } \xi \in \text{ftv}(\tilde{\tau}), \text{ otherwise } m = \infty.$$

$$\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}[x : \tilde{\rho}]) = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}).$$

**Definition 7.**  $\mathcal{V}(\xi^m, \Gamma) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma))$

**Lemma 2.** If  $\xi \notin \text{ftv}(\tilde{\Gamma}_\lambda)$  then  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}) = \infty$ .

*Proof (By induction on the structure of  $\tilde{\Gamma}$ ).*

$\tilde{\Gamma} = \bullet$ : By definition,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}) = \infty$ .

$\tilde{\Gamma} = \tilde{\Gamma}'[x : \tilde{\tau}]$ : By assumption,  $\xi \notin \text{ftv}(\tilde{\tau})$  because  $\xi \notin \text{ftv}(\tilde{\Gamma}_\lambda)$ . Because  $\xi \notin \text{ftv}(\tilde{\Gamma}_\lambda)$ , certainly  $\xi \notin \text{ftv}(\tilde{\Gamma}'_\lambda)$ . By induction,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \infty$ . Thus,  $\tilde{\mathcal{V}}(\tilde{\Gamma}_\lambda) = \infty$ .

$\tilde{\Gamma} = \tilde{\Gamma}'[x : \tilde{\rho}]$ : Because  $\xi \notin \text{ftv}(\tilde{\Gamma}_\lambda)$ , certainly  $\xi \notin \text{ftv}(\tilde{\Gamma}'_\lambda)$ . By induction,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \infty$ . Thus,  $\tilde{\mathcal{V}}(\tilde{\Gamma}_\lambda) = \infty$ .

**Lemma 3.** *If  $\xi \in \text{ftv}(\tilde{\Gamma}_\lambda)$ , then  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}) \leq \mathcal{D}(\tilde{\Gamma})$ .*

*Proof (By induction on the length of  $\tilde{\Gamma}$ ).*

**Base case  $\tilde{\Gamma} = \bullet$ :** Vacuously true

**Inductive case  $\tilde{\Gamma} = \tilde{\Gamma}'[x : \tilde{\tau}]$ :**  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \tilde{\tau}]) = \min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), m)$

**Subcase  $\xi \in \text{ftv}(\tilde{\tau})$ :**  $m = \mathcal{D}(\tilde{\Gamma}') + 1 = \mathcal{D}(\tilde{\Gamma})$ .

**Subcase  $\xi \in \text{ftv}(\tilde{\Gamma}'_\lambda)$ :** Then by induction  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') \leq \mathcal{D}(\tilde{\Gamma}')$ . Thus,  $\min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), \mathcal{D}(\tilde{\Gamma}') + 1) = \min(\mathcal{D}(\tilde{\Gamma}'), \mathcal{D}(\tilde{\Gamma}') + 1) = \mathcal{D}(\tilde{\Gamma}') < \mathcal{D}(\tilde{\Gamma}') + 1 = \mathcal{D}(\tilde{\Gamma})$ .

**Subcase  $\xi \notin \text{ftv}(\tilde{\Gamma}'_\lambda)$ :** By lem. 2,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \infty$ .  $m$  is finite. Thus,  $\min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), m) = m = \mathcal{D}(\tilde{\Gamma})$ .

**Inductive case  $\tilde{\Gamma} = \tilde{\Gamma}'[x : \rho]$ :**  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \rho]) = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}')$  by definition of  $\tilde{\mathcal{V}}$ . Thus, the above case applies. □

By definition, any type environments can be extended with more bindings. I formally define the notion of extension:

**Definition 8.**  $\tilde{\Gamma}'$  is an **extension** of  $\tilde{\Gamma}$  if and only if there exists  $\tilde{\Gamma}''$  such that  $\tilde{\Gamma}'$  is the concatenation of  $\tilde{\Gamma}$  and  $\tilde{\Gamma}''$  (in that order).

$\Gamma$  is a **prefix** of  $\Gamma'$  if and only if  $\Gamma'$  is an extension of  $\Gamma$ .

Notice that if a type variable occurs in a type environment, then extension of that type environment does not change that type variable's leftmost occurrence:

**Lemma 4 ( $\tilde{\mathcal{V}}$  Invariant Under  $\tilde{\Gamma}$  Extension).**

*If  $\xi \in \text{ftv}(\tilde{\Gamma})$  and  $\tilde{\Gamma}'$  is an extension of  $\tilde{\Gamma}$ , then  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .*

*Proof (By induction on the length of  $\tilde{\Gamma}'$ ).*

**Base Case ( $|\tilde{\Gamma}'| = |\tilde{\Gamma}|$ ):** By definition of extension,  $\tilde{\Gamma}' = \tilde{\Gamma}$ . Thus,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

**Inductive Case:** Induction Hypothesis: For  $|\tilde{\Gamma}'| = n$ , if  $\xi \in \text{ftv}(\tilde{\Gamma})$  and  $\tilde{\Gamma}'$  extends  $\tilde{\Gamma}$ , then  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

Assume  $\xi \in \text{ftv}(\tilde{\Gamma})$  and  $\tilde{\Gamma}''$  extends  $\tilde{\Gamma}$  (such that  $|\tilde{\Gamma}''| = n + 1$ ).

To show:  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

By definition of  $\tilde{\Gamma}''$ :

**Subcase**  $\tilde{\Gamma}'' = \tilde{\Gamma}'[x : \tilde{\tau}]$ :  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \tilde{\tau}])$ .

**Subcase**  $\xi \in \text{ftv}(\tilde{\tau})$ :  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \tilde{\tau}]) = \min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), \mathcal{D}(\tilde{\Gamma}') + 1)$ . By lem. 3,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') \leq \mathcal{D}(\tilde{\Gamma}') < \mathcal{D}(\tilde{\Gamma}') + 1$  ( $\mathcal{D}$  is always positive). Therefore  $\min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), \mathcal{D}(\tilde{\Gamma}') + 1) = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}')$ . By induction,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

**Subcase**  $\xi \notin \text{ftv}(\tilde{\tau})$ :  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \tilde{\tau}]) = \min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), \infty)$ . By induction,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ . By lem. 3,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}) \leq \mathcal{D}(\tilde{\Gamma}) < \infty$ . Thus,  $\min(\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'), \infty) = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

**Subcase**  $\tilde{\Gamma}'' = \tilde{\Gamma}'[x : \rho]$ : By definition,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}'[x : \rho]) = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma}')$ . By induction,  $\tilde{\mathcal{V}}(\xi, \tilde{\Gamma}') = \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$ .

□

**Lemma 5 ( $\mathcal{V}$  Invariant Under  $\Gamma$  Extension).**

If  $\xi^m \in \text{ftv}(\Gamma)$  and  $\Gamma'$  is an extension of  $\Gamma$ , then  $\mathcal{V}(\xi^m, \Gamma') = \mathcal{V}(\xi^m, \Gamma)$ .

*Proof.* By definition,  $\mathcal{V}(\xi^m, \Gamma') = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma'))$  and  $\mathcal{V}(\xi^m, \Gamma) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma))$ . Because  $\xi^m \in \text{ftv}(\Gamma)$ , it must be the case that  $\xi \in \text{ftv}(\mathcal{E}_r(\Gamma))$  by definition of  $\mathcal{E}_r$ .  $\mathcal{E}_r(\Gamma')$  is an extension of  $\mathcal{E}_r(\Gamma)$ . By lem. 4,  $\tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma)) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma'))$ . Thus, I can conclude that  $\mathcal{V}(\xi^m, \Gamma') = \mathcal{V}(\xi^m, \Gamma)$ . □

The proof that the invariants hold for [pm-u-mv] requires on the following basic observation:

**Lemma 6 (Substitution Range Preservation).** If  $\tau \in \text{rng } \Gamma$ , then  $\sigma\tau \in \text{rng } \sigma\Gamma$ .

*Proof* (follows from definition of type variable substitution on type environments).

**Lemma 7.** If  $\xi^m$  occurs in the  $n$ th  $\lambda$ -binding in  $\Gamma$  (counting from the left) and  $\xi^m$  occurs in no  $\lambda$ -binding to the left of that  $n$ th  $\lambda$ -binding, then  $\mathcal{V}(\xi^m, \Gamma) = n$ .

*Proof.*

**Case  $\Gamma = \bullet$ :** This case is vacuously true.

**Case  $\Gamma = \Gamma'[x : \tau]$ :**

**Case  $\xi^m \in \text{ftv}(\tau)$ :** If this rightmost binding  $[x : \tau]$  is the  $n$ th  $\lambda$ -binding, then by assumption there are no occurrences of  $\xi^m$  to the left. By lem. 2,  $\tilde{\mathcal{V}}(\xi, \Gamma') = \infty$ . Thus,  $\mathcal{V}(\xi^m, \Gamma'[x : \tau]) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma')[x : \mathcal{E}_r(\tau)]) = \min(\tilde{\mathcal{V}}(\xi, \Gamma'), \mathcal{D}(\Gamma') + 1) = \mathcal{D}(\Gamma') + 1$ .  $n = \mathcal{D}(\Gamma') + 1$  because  $[x : \tau]$  is the  $n$ th  $\lambda$  binding by definition of  $\mathcal{D}$ .

**Case  $\xi^m \notin \text{ftv}(\tau)$ :**  $\mathcal{V}(\xi^m, \Gamma'[x : \tau]) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma')[x : \mathcal{E}_r(\tau)]) = \min(\tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma')), \infty)$ . By induction,  $\tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma)) = n$ .  $\min(\tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma')), \infty) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma)) = n$ .

**Case  $\Gamma = \Gamma'[x : \rho]$ :**  $\mathcal{V}(\xi^m, \Gamma'[x : \rho]) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma')[x : \mathcal{E}_r(\rho)]) = \tilde{\mathcal{V}}(\xi, \mathcal{E}_r(\Gamma'))$ . The above cases apply.

□

**Invariant 1 (Inv<sub>1</sub>(s))** Let  $s = (-, \Gamma, -, -) \in \text{VMS}$ .  $\forall \xi^m \in \text{ftv}(\Gamma). \mathcal{V}(\xi^m, \Gamma) = m$

**Corollary 1 (Predicate on HM Machine State  $s$ ).**  $\forall \Gamma$  from  $s = (c, \Gamma, \Sigma, k) \in \text{VMS}$

$\xi^m \in \text{ftv}(\Gamma) \Rightarrow m \leq \mathcal{D}(\Gamma)$

*Proof.*  $\xi^m \in \text{ftv}(\Gamma) \Rightarrow \mathcal{V}(\xi^m, \Gamma) = m$  by Inv<sub>1</sub>. Thus, the  $m$ th  $\lambda$ -binding in  $\Gamma$  is the leftmost occurrence of  $\xi^m$  in  $\Gamma$ . Certainly, the  $m$ th binding is within  $\Gamma$  ( $m$  is no more than the length of  $\Gamma$ ,  $\mathcal{D}(\Gamma)$ ), hence  $m \leq \mathcal{D}(\Gamma)$  □

**Invariant 2 (Inv<sub>2</sub>(s))** Let  $s = (c, \Gamma, -, k) \in \text{VMS}$

$\forall \xi^m \in \text{ftv}(k[c]). m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$

**Invariant 3 (Inv<sub>3</sub>(s))** Let  $s = (-, \Gamma, -, k) \in \text{VMS}$ ,  $\mathcal{D}(\Gamma) =$  the number of  $\rightarrow$  frames on  $k$ .

I use  $\text{Inv}_1$  as an invariant instead of its corollary because proving  $\text{Inv}_2$  for the [pm-arr] case requires a more precise position of the first occurrence of a type variable than is supplied by the corollary.  $\text{Inv}_3$  is necessary in order to make the [pm-lam] and [pm-let-def] cases go through for  $\text{Inv}_1$ . I need to precisely specify that  $\mathcal{D}(\Gamma)$  corresponds exactly to the number of  $\lambda$ -bindings enclosing the hole in the context  $k$ , thus matching the ranking in  $\Gamma$  to the ranking in  $k$ .

**Lemma 8 (Invariants hold for Initial states).**  $\forall s \in \{(e, \bullet, \bullet, \bullet)\}$ ,  $\text{Inv}_1(s)$ ,  $\text{Inv}_2(s)$ , and  $\text{Inv}_3(s)$ .

*Proof.*

**Inv<sub>1</sub> and Inv<sub>2</sub>** Vacuously true because the type environment register is empty and the control contains no type variables

**Inv<sub>3</sub>**  $\mathcal{D}(\bullet) = 0 = \#$  of  $\rightarrow$  frames on  $\bullet$ .

□

In the following proof, I use the term “ $\rightarrow$  frame” to refer to the  $(\rightarrow \tau \square)$  context stack frame.

**Lemma 9 (Invariants hold for reduction rules).** For all  $s \in \text{VMS}$ , if  $\text{Inv}_1(s)$ ,  $\text{Inv}_2(s)$ ,  $\text{Inv}_3(s)$ , and  $s \mapsto_{pm} s'$ , then  $\text{Inv}_1(s')$ ,  $\text{Inv}_2(s')$ , and  $\text{Inv}_3(s')$ .

*Proof (By cases on reduction rules defining  $\mapsto_{pm}$ ).*

**Case [pm-app], [pm-app-left], [pm-app-right], and [pm-let]**

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** By  $k[e]/k[\tau]$  invariance

**Inv<sub>3</sub>** Assume  $\mathcal{D}(\Gamma) =$  number of  $\rightarrow$  frames in  $k$ . The number of  $\rightarrow$  frames in  $(@ \square e')$   $:: k$  is the same as the number of  $\rightarrow$  frames in  $k$ . Thus,  $\mathcal{D}(\Gamma) =$  number of  $\rightarrow$  frames in  $(@ \square e) :: k$ .

**Case [pm-lam]**

Assumptions

A1. Assume  $\forall \xi^m \in \text{ftv}(\Gamma) \ \mathcal{V}(\xi^m, \Gamma) = m$ .

A2. Assume  $\forall \xi^m \in \text{ftv}(k[(\lambda^d(x) e)]) . m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .

A3. Assume  $\mathcal{D}(\Gamma) = \text{number of } \rightarrow \text{ frames in } k$ .

**Inv<sub>1</sub>** Assume  $\phi^n \in \text{ftv}(\Gamma[x : \psi^q])$ . To show:  $\mathcal{V}(\phi^n, \Gamma) = n$ .

Either  $\phi^n \in \text{ftv}(\Gamma)$  or  $\phi^n = \psi^q$ .

**Case  $\phi^n \in \text{ftv}(\Gamma)$ :**  $\mathcal{V}(\phi^n, \Gamma) = n$  by A1. By lem. 5,  $\mathcal{V}(\phi^n, \Gamma[x : \psi^q]) = n$ .

**Case  $\phi^n = \psi^q$ :**  $\psi^q$  was chosen as fresh, so  $\psi^q \notin \text{ftv}(\Gamma)$ . By ranking correctness (thm. D.1, a property that says that the rank  $q$  of the closest  $\lambda$  enclosing the control must be the number of  $\lambda$ s enclosing that control),  $q$  is the number of  $\lambda$ s enclosing the control  $e$ . In the context stack, the  $\lambda$ s all become  $\rightarrow$  frames. Thus, by A3, there are  $q$   $\rightarrow$  frames on  $(\rightarrow \psi^q \square) :: k$ . Consequently, there are exactly  $(q - 1)$   $\rightarrow$  frames on  $k$ . By the definition of  $\mathcal{D}$ ,  $\mathcal{D}(\Gamma) = q - 1$ . Thus,  $[x : \psi^q]$  is the  $q$ th  $\lambda$ -binding in  $\Gamma[x : \psi^q]$ . By definition,  $\mathcal{V}(\psi^q, \Gamma[x : \psi^q]) = \tilde{\mathcal{V}}(\psi, \mathcal{E}_r(\Gamma)[x : \psi]) = \min(\tilde{\mathcal{V}}(\psi, \mathcal{E}_r(\Gamma)), \mathcal{D}(\mathcal{E}_r(\Gamma)) + 1)$ . By lem. 2,  $\tilde{\mathcal{V}}(\psi, \mathcal{E}_r(\Gamma)) = \infty$ . Thus,  $\min(\tilde{\mathcal{V}}(\psi, \mathcal{E}_r(\Gamma)), \mathcal{D}(\mathcal{E}_r(\Gamma)) + 1) = \mathcal{D}(\mathcal{E}_r(\Gamma)) + 1 = q$ . Thus,  $\mathcal{V}(\phi^n, \Gamma[x : \phi^n]) = n$ .

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(((\rightarrow \psi^q \square) :: k)[e]) . n \leq \mathcal{D}(\Gamma[x : \psi^q])$ .

To show:  $\phi^n \in \text{ftv}(\Gamma[x : \psi^q])$

$\mathcal{D}(\Gamma[x : \psi^q]) = 1 + \mathcal{D}(\Gamma)$  by definition of  $\mathcal{D}$ . Because by assumption  $n \leq 1 + \mathcal{D}(\Gamma)$ , either  $n = 1 + \mathcal{D}(\Gamma)$  or  $n \leq \mathcal{D}(\Gamma)$ . By ranking correctness (thm. 1),  $q = \text{the number of enclosing } \lambda \text{ s } (\rightarrow \text{ frames on } k) + 1$ . By A3,  $\mathcal{D}(\Gamma) = \# \text{ of } \rightarrow \text{ frames on } k$ . Hence,  $q = \mathcal{D}(\Gamma) + 1$ . By assumption,  $n \leq q$ . Thus, either  $n = q$  or  $n \leq q - 1 = \mathcal{D}(\Gamma)$ .

**Case  $n = q$ :** Also by ranking correctness, there is only one type variable with rank  $q$  in this context stack  $(\rightarrow \psi^q \square) :: k$ .  $\phi^n$  cannot be in  $\text{ftv}(e)$  because  $e$  cannot contain type variables by definition of ULCL. Therefore,  $\phi^n = \psi^q$ . Certainly  $\psi^q \in \text{ftv}(\Gamma[x : \psi^q])$ . So,  $\phi^n \in \text{ftv}(\Gamma[x : \psi^q])$ .

**Case  $n \leq \mathcal{D}(\Gamma) < q$ :** By case assumption,  $\phi^n \neq \psi^q$ , so  $\phi^n \in \text{ftv}(k) \cup \text{ftv}(e)$ .  
 $\phi^n \in \text{ftv}(\Gamma) \subseteq \text{ftv}(\Gamma[x : \psi^q])$  by A2.

**Inv<sub>3</sub>**  $D(\Gamma[x : \xi^d]) = 1 + D(\Gamma)$ . By  $\lambda$ -ranking correctness (thm. D.1),  $d = 1 + \text{number of } \rightarrow \text{ in } k$  (i.e.,  $1 + \text{the rank of the } \lambda \text{ associated with the leftmost } \rightarrow \text{ frame}$ ). By assumption, the number of  $\rightarrow$  frames in  $k = \mathcal{D}(\Gamma)$ . Thus,  $D(\Gamma[x : \xi^d]) = 1 + D(\Gamma) = 1 + \text{number of } \rightarrow \text{ frame in } k = d = \text{number of } \rightarrow \text{ frames in } (\rightarrow \xi^d \square) :: k$ .

### Case [pm-arr]

A1. Assume  $\forall \xi^m \in \text{ftv}(\Gamma[x : \tau]) \ \mathcal{V}(\xi^m, \Gamma[x : \tau]) = m$ .

A2. Assume  $\forall \xi^m \in \text{ftv}(((\rightarrow \tau \square) :: k)[\tau']). m \leq \mathcal{D}(\Gamma[x : \tau]) \Rightarrow \xi^m \in \text{ftv}(\Gamma[x : \tau])$

A3. Assume  $\mathcal{D}(\Gamma[x : \tau]) = \text{the number of } \rightarrow \text{ frames in } (\rightarrow \tau \square) :: k$ .

**Inv<sub>1</sub>** Assume  $\phi^n \in \text{ftv}(\Gamma)$ . To show:  $\mathcal{V}(\phi^n, \Gamma) = n$ .

$\phi^n \in \text{ftv}(\Gamma) \subseteq \text{ftv}(\Gamma[x : \tau])$ , so  $\mathcal{V}(\phi^n, \Gamma[x : \tau]) = n$  by A1.  $\Gamma[x : \tau]$  is an extension of  $\Gamma$ . By lem. 5,  $\mathcal{V}(\phi^n, \Gamma[x : \tau]) = \mathcal{V}(\phi^n, \Gamma)$ . By A1,  $\mathcal{V}(\phi^n, \Gamma[x : \tau]) = n$ . Thus,  $\mathcal{V}(\phi^n, \Gamma) = n$ .

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[(\rightarrow \tau \tau')])$ .  $n \leq \mathcal{D}(\Gamma)$ . To show  $\phi^n \in \text{ftv}(\Gamma)$ .

$\phi^n \in \text{ftv}(k[(\rightarrow \tau \tau')]) \subseteq \text{ftv}(((\rightarrow \tau \square) :: k)[\tau'])$  and  $n \leq \mathcal{D}(\Gamma) < \mathcal{D}(\Gamma[x : \tau])$ . By A2,  $\phi^n \in \text{ftv}(\Gamma[x : \tau])$ . By definition of  $\text{ftv}$ , either  $\phi^n \in \text{ftv}(\Gamma)$  or  $\phi^n \in \text{ftv}(\tau)$ .

**Case  $\phi^n \in \text{ftv}(\Gamma)$ :** Done

**Case  $\phi^n \in \text{ftv}(\tau)$ :** By A1,  $\phi^n$ 's leftmost binding occurrence must be in the  $n$ th binding in  $\Gamma[x : \tau]$ . Because  $n < \mathcal{D}(\Gamma[x : \tau])$ , the  $n$ th binding cannot be the rightmost binding  $[x : \tau]$ . Thus,  $\phi^n$ 's leftmost binding occurrence cannot be in the rightmost binding  $[x : \tau]$ . So  $[x : \tau]$  is not the leftmost binding with an occurrence  $\phi^n$ . Thus,  $\phi^n$  must occur in  $\text{ftv}(\Gamma)$ .

**Inv<sub>3</sub>** By definition of  $\mathcal{D}$ ,  $\mathcal{D}(\Gamma) = \mathcal{D}(\Gamma[x : \tau]) - 1$ .  $\mathcal{D}(\Gamma[x : \tau]) - 1 = \text{number of } \rightarrow \text{ frames in } ((\rightarrow \tau \square) :: k) - 1$  by A3. The number of  $\rightarrow$  frames in  $((\rightarrow \tau \square) :: k) - 1 = \text{the number of } \rightarrow \text{ frames in } k$ .

**Case [pm-var]**

A2. Assume  $\forall \xi^m \in \text{ftv}(k[x]). m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[\Gamma(x)]). n \leq \mathcal{D}(\Gamma)$ . To show:  $\phi^n \in \text{ftv}(\Gamma)$ .

Either  $\phi^n \in \text{ftv}(k)$  or  $\phi^n \in \text{ftv}(\Gamma(x))$ .

**Case  $\phi^n \in \text{ftv}(k)$ :** By A2,  $\phi^n \in \text{ftv}(\Gamma)$ .

**Case  $\phi^n \in \text{ftv}(\Gamma(x))$ :**  $x : \tau \in \Gamma$  by definition of  $\Gamma(x)$ .  $\phi^n \in \text{ftv}(\tau)$ . Thus  $\phi^n \in \text{ftv}(\Gamma)$ .

**Inv<sub>3</sub>** By  $\Gamma$  invariance

**Case [pm-num]**

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** By  $\text{ftv}(k[c])$  invariance

**Inv<sub>3</sub>** By  $\Gamma$  invariance

**Case [pm- $\tau\beta$ ]**

A2. Assume  $\forall \xi^m \in \text{ftv}(k[(@ \tau \tau')]). m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[(\mathbf{unify} \tau (\rightarrow \tau' \psi^\infty) \psi^\infty)]). n \leq \mathcal{D}(\Gamma)$ .

To show:  $\phi^n \in \text{ftv}(\Gamma)$ .

One of the following cases must be true by definition of  $\text{ftv}$ :

**Case  $\phi^n \in \text{ftv}(k) \cup \text{ftv}(\tau) \cup \text{ftv}(\tau')$ :**  $\phi^n \in \text{ftv}(k) \cup \text{ftv}(\tau) \cup \text{ftv}(\tau') = \text{ftv}(k[(@ \tau \tau')])$   
by definition of  $\text{ftv}$ . By A2,  $\phi^n \in \text{ftv}(\Gamma)$ .

**Case  $\phi^n = \psi^\infty$ :** This case is impossible because  $\infty \notin \mathcal{D}(\Gamma)$ , always a finite number.

**Inv<sub>3</sub>** By  $\Gamma$  invariance

**Case [pm-let-def]**

A1. Assume  $\forall \xi^m \in \text{ftv}(\Gamma) \ \mathcal{V}(\xi^m, \Gamma) = m$ .

- A2. Assume  $\forall \xi^m \in \text{ftv}((\mathbf{let}^d (x \square) e') :: k[\tau]).m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .
- A3. Let  $k_1 = (\mathbf{let}^d (x \square) e') :: k$ . Assume  $\mathcal{D}(\Gamma) =$  the number of  $\rightarrow$  frames in  $k_1$ .

**Inv<sub>1</sub>** Assume  $\phi^n \in \text{ftv}(\Gamma[x : \mathcal{P}_d(\tau)])$ . To show:  $\mathcal{V}(\phi^n, \Gamma[x : \mathcal{P}_d(\tau)]) = n$ .

Either  $\phi^n$  is in  $\mathcal{P}_d(\tau)$  or it is in  $\Gamma$ .

Assume that  $\phi^n \in \text{ftv}(\mathcal{P}_d(\tau))$ . By definition of  $\mathcal{P}_d(\tau)$ ,  $\phi^n \in \text{ftv}(\forall \vec{\alpha} \{ \mathcal{G}_d(\tau) \mapsto \vec{\alpha} \} \tau) = \text{ftv}(\tau) \setminus \text{supp}(\mathcal{G}_d(\tau))$ . By definition of  $\mathcal{G}_d(\tau)$ ,  $\phi^n \in \text{ftv}(\tau).n \leq d$ . By ranking correctness (thm. D.1) and A3,  $d = \mathcal{D}(\Gamma)$  where  $d$  is the rank associated with the  $(\mathbf{let}^d (x \square) e')$  frame on the top of the context stack. By A2,  $\phi^n \in \text{ftv}(\tau) \subseteq \text{ftv}(((\mathbf{let}^d (x \square) e') :: k)[\tau]).n \leq \mathcal{D}(\Gamma)$  implies that  $\phi^n \in \text{ftv}(\Gamma)$ . Hence if  $\phi^n \in \text{ftv}(\Gamma[x : \mathcal{P}_d(\tau)])$ ,  $\phi^n \in \text{ftv}(\Gamma)$ .

Now assume that  $\phi^n \in \text{ftv}(\Gamma)$ ,  $\mathcal{V}(\phi^n, \Gamma) = n$  by A1.  $\phi^n$  must occur in a binding in  $\Gamma$ . By lem. 5,  $\mathcal{V}(\phi^n, \Gamma[x : \mathcal{P}_d(\tau)]) = \mathcal{V}(\phi^n, \Gamma) = n$ .

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}((\mathbf{let}^d \square) :: k[e']).n \leq \mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)])$ .

To show:  $\phi^n \in \text{ftv}(\Gamma[x : \mathcal{P}_d(\tau)])$ .

$\phi^n \in \text{ftv}((\mathbf{let}^d \square) :: k[e']) \subseteq \text{ftv}((\mathbf{let}^d (x \square) e') :: k[\tau])$  by definition of  $\text{ftv}$ . Because  $[x : \mathcal{P}_d(\tau)]$  is not a  $\lambda$ -binding,  $n \leq \mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)]) = \mathcal{D}(\Gamma)$ . By A2,  $\phi^n \in \text{ftv}(\Gamma) \subseteq \text{ftv}(\Gamma[x : \rho])$ .

**Inv<sub>3</sub>** Let  $k_2 = (\mathbf{let}^d \square) :: k$ . To show:  $\mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)]) =$  number of  $\rightarrow$  frames in  $k_2$ .  $\mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)]) = \mathcal{D}(\Gamma)$  because  $[x : \mathcal{P}_d(\tau)]$  is not a  $\lambda$ -binding. By A3,  $\mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)]) = \mathcal{D}(\Gamma) =$  number of  $\rightarrow$  frames in  $k_1$ . The top of  $k_1$  is not a  $\rightarrow$  frame, thus the number of  $\rightarrow$  frames in  $k_1$  is equal to the number of  $\rightarrow$  frames in  $k$ . The top of the  $k_2$  stack is also not a  $\rightarrow$  frame, thus the number of  $\rightarrow$  frames in  $k_2$  is also the number of  $\rightarrow$  frames in  $k$ . Consequently,  $\mathcal{D}(\Gamma[x : \mathcal{P}_d(\tau)]) =$  the number of  $\rightarrow$  frames in  $k_2$ .

### Case [pm-let-body]

- A1. Assume  $\forall \xi^m \in \text{ftv}(\Gamma[x : \rho]) \mathcal{V}(\xi^m, \Gamma[x : \rho]) = m$ .
- A2. Assume  $\forall \xi^m \in \text{ftv}(((\mathbf{let}^d \square) :: k)[\tau]).m \leq \mathcal{D}(\Gamma[x : \rho]) \Rightarrow \xi^m \in \text{ftv}(\Gamma[x : \rho])$ .
- A3. Assume  $\mathcal{D}(\Gamma[x : \rho]) =$  number of  $\rightarrow$  frames in  $(\mathbf{let}^d \square) :: k$ .

**Inv<sub>1</sub>** Assume  $\phi^n \in \text{ftv}(\Gamma)$ . To show:  $\mathcal{V}(\phi^n, \Gamma) = n$ .

$\phi^n \in \text{ftv}(\Gamma[x : \rho])$  because  $\text{ftv}(\Gamma) \subseteq \text{ftv}(\Gamma[x : \rho])$ .  $\Gamma[x : \rho]$  is an extension of  $\Gamma$ . By lem. 5,  $\mathcal{V}(\phi^n, \Gamma) = \mathcal{V}(\phi^n, \Gamma[x : \rho])$ . By A1,  $\mathcal{V}(\phi^n, \Gamma[x : \rho]) = n$ . Thus,  $\mathcal{V}(\phi^n, \Gamma) = n$ .

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[\tau]).n \leq \mathcal{D}(\Gamma)$ . To show:  $\phi^n \in \text{ftv}(\Gamma)$

$\phi^n \in \text{ftv}(k[\tau]) = \text{ftv}(((\mathbf{let}^d \square) :: k)[\tau])$  by definition of  $\text{ftv}$ .  $n \leq \mathcal{D}(\Gamma) = \mathcal{D}(\Gamma[x : \rho])$  because  $[x : \rho]$  is not a  $\lambda$ -binding. By A2,  $\phi^n \in \text{ftv}(\Gamma[x : \rho])$ . By A1,  $\phi^n$  first occurs in the  $n$ th  $\lambda$ -binding in  $\Gamma[x : \rho]$ .  $[x : \rho]$  is a **let** binding, so  $\phi^n$  must first occur in  $\Gamma$ .

**Inv<sub>3</sub>**  $\mathcal{D}(\Gamma) = \mathcal{D}(\Gamma[x : \rho]) = \text{number of } \rightarrow \text{ frames in } (\mathbf{let}^d \square) :: k = \text{the number of } \rightarrow \text{ frames in } k$ .

### Case [pm-poly]

A2. Assume  $\forall \xi^m \in \text{ftv}(k[\forall \vec{\alpha} \tau_p]).m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[\{\vec{\alpha} \mapsto \vec{\psi}^\infty\} \tau_p]).n \leq \mathcal{D}(\Gamma)$ . To show:  $\phi^n \in \text{ftv}(\Gamma)$ .

Either  $\phi^n \in \text{ftv}(\vec{\psi}^\infty)$  or  $\phi^n \in \text{ftv}(k) \cup \text{ftv}(\tau_p)$ .

**Case  $\phi^n \in \text{ftv}(\vec{\psi}^\infty)$ :** This invariant is vacuously true because  $\infty \notin \mathcal{D}(\Gamma)$ .

**Case  $\phi^n \in \text{ftv}(k) \cup \text{ftv}(\tau_p)$ :** Because  $\vec{\alpha}$  are distinct from typical, finite type variables,  $\text{ftv}(k) \cup \text{ftv}(\tau_p) = \text{ftv}(k[\forall \vec{\alpha} \tau_p])$ . By A2,  $\phi^n \in \text{ftv}(\Gamma)$ .

**Inv<sub>3</sub>** By invariance of  $\Gamma$  and  $k$ .

### Case [pm-u-mv]

A1.  $\forall \xi^m \in \text{ftv}(\Gamma) \mathcal{V}(\xi^m, \Gamma) = m$ .

A2.  $\forall \xi^m \in \text{ftv}(k[(\mathbf{unify} \psi^d \tau u)]).m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$

A3.  $\mathcal{D}(\Gamma) = \# \text{ of } \rightarrow \text{ frames in } k$ .

**Inv<sub>1</sub>** Assume  $\phi^n \in \text{ftv}(\sigma\Gamma)$ . To show:  $\mathcal{V}(\phi^n, \sigma\Gamma) = n$ .

Because  $\phi^n \in \text{ftv}(\sigma\Gamma)$ , one of the following cases must be true (by definition of substitution):

**Subcase  $\phi^n \in \text{ftv}(\Gamma) \setminus (\{\xi^d\} \cup \text{supp}(\mathcal{L}(\tau, d)) \cup \text{ftv}(\tau))$ :**

$\phi^n \in \text{ftv}(\Gamma) \setminus (\{\xi^d\} \cup \text{supp}(\mathcal{L}(\tau, d)) \subseteq \text{ftv}(\Gamma)$ . By A1,  $\mathcal{V}(\phi^n, \Gamma) = n$ . Because  $\phi^n$  is not in  $(\{\xi^d\} \cup \text{supp}(\mathcal{L}(\tau, d)) \cup \text{ftv}(\tau))$ , it is in neither the range nor the support of  $\sigma$ . Thus,  $\sigma$  does not touch occurrences of  $\phi^n$  in  $\Gamma$ .  $\mathcal{V}(\phi^n, \sigma\Gamma) = n$ .

**Subcase  $\phi^n \in \text{ftv}(\tau)$  such that  $n \leq d$  (i.e. not in  $\text{supp}(\mathcal{L}(\tau, d))$ ):**

Because  $\sigma = \mathcal{L}(\tau, d) \circ \{\xi^d \mapsto \tau\}\Gamma$  and  $\phi^n \notin \text{ftv}(\Gamma) \cup \text{ftv}(\text{rng}(\mathcal{L}(\tau, d)))$ ,  $\xi^d \in \text{ftv}(\Gamma)$  (because otherwise the  $\{\xi^d \mapsto \tau\}$  substitution would not do anything including introducing  $\phi^n$ ). By A1,  $\mathcal{V}(\xi^d, \Gamma) = d$ . By lem. 3,  $d = \mathcal{V}(\xi^d, \Gamma) \leq \mathcal{D}(\Gamma)$ . It must be the case that  $\phi^n \in \text{ftv}(\tau) \subset \text{ftv}(k[(\text{unify } \xi^d \tau u)])$ . By assumption and the above reasoning  $n \leq d \leq \mathcal{D}(\Gamma)$ . By A2,  $\phi^n \in \text{ftv}(\Gamma)$ . By A1,  $\mathcal{V}(\phi^n, \Gamma) = n$ . Because  $\phi^n$  is not in the  $\mathcal{L}(\tau, d)$  substitution, it is not promoted, hence  $\sigma$  does not change its rank. Moreover,  $\sigma$  only replaces  $\xi^d$  with  $\tau$ , hence the leftmost binding it could add  $\phi^n$  to is the  $d$ th  $\lambda$ -binding which cannot be to the left of the leftmost  $\lambda$ -binding occurrence of  $\phi^n$  in  $\Gamma$  at the  $n$ th binding ( $n \leq d$ ). That occurrence remains intact in  $\sigma\Gamma$  (possibly with some additional occurrences of  $\phi^n$  in that same  $\lambda$ -binding). By lem. 7,  $\mathcal{V}(\phi^n, \sigma\Gamma) = n$ .

**Subcase  $\phi^n \in \text{ftv}(\text{rng}(\mathcal{L}(\tau, d)))$  such that  $\phi^m \in \text{ftv}(\tau), m > d = n$ :**

By definition  $\mathcal{L}(\tau, d), n = d$  and  $\phi^m \in \text{ftv}(\tau)$  such that  $m > d$ .

**(No Demotion)** Assume that  $\xi^d \notin \text{ftv}(\Gamma)$ , then by contrapositive of A2,  $d > \mathcal{D}(\Gamma)$ . Thus  $m > d > \mathcal{D}(\Gamma)$ .  $\mathcal{V}(\phi^m, \Gamma) \neq m$  because there is no  $m$ th binding in  $\Gamma$ . By contrapositive of A1,  $\phi^m \notin \text{ftv}(\Gamma)$ . Then  $\{\phi^m \mapsto \phi^d\} \circ \{\xi^d \mapsto \tau\}\Gamma = \Gamma$ . By definition, the remainder of  $\mathcal{L}(\tau, d)$  does not mention  $\phi^d$ . Hence  $\phi^d \notin \text{ftv}(\sigma\Gamma)$ .  $\Rightarrow \Leftarrow$

Thus, it must be the case that  $\xi^d \in \text{ftv}(\Gamma)$ .

**(Main Body)** There are only two possible kinds of  $\lambda$ -bindings in  $\Gamma$  where  $\phi^d$  can be found after substitution: any  $\lambda$ -bindings containing  $\phi^m$  and any  $\lambda$ -bindings containing  $\xi^d$ . Only the bindings containing  $\xi^d$  matter because the leftmost occurrence of  $\xi^d$  comes before all bindings with  $\phi^m$ .

**Subcase  $\phi^m \in \text{ftv}(\Gamma)$ :** By A1,  $\mathcal{V}(\phi^m, \Gamma) = m > d$ . All the  $\lambda$ -bindings with  $\phi^m$  occurrences come after the  $d$ th binding. In  $\sigma\Gamma$ ,  $\phi^d$  occurs in all the  $\lambda$ -bindings formerly containing  $\phi^m$  occurrences which come strictly after the  $d$ th binding. The remaining occurrences of  $\phi^d$  come from the  $\xi^d$  substitutions. These occurrences are covered below.

**Subcase  $\phi^m \notin \text{ftv}(\Gamma)$ :** The only remaining bindings with  $\phi^d$  occurrences must come from the  $\xi^d$  substitution. By A1,  $\mathcal{V}(\xi^d, \Gamma) = d$ .  $\sigma$  replaces all  $\xi^d$  with  $\mathcal{L}(\tau, d)\tau$  containing  $\phi^d$ . These are the only  $\lambda$ -bindings with  $\phi^d$ -occurrences in  $\sigma\Gamma$ . So the leftmost occurrence of  $\phi^d$  in a  $\lambda$ -binding in  $\sigma\Gamma$  must correspond to the leftmost occurrence of  $\xi^d$  in  $\Gamma$ , namely the  $d$ th binding. By lem. 7,  $\mathcal{V}(\phi^d, \sigma\Gamma) = d$ .

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(\sigma k[\sigma u]).n \leq \mathcal{D}(\sigma\Gamma)$

To show  $\phi^n \in \text{ftv}(\sigma\Gamma)$

**Case  $\phi^n \in \text{ftv}(k[u]) \setminus (\{\xi^d\} \cup \text{supp}(\mathcal{L}(\tau, d)) \cup \text{ftv}(\tau))$ :**

Because substitutions do not affect number of  $\lambda$ -bindings in a type environment,  $n \leq \mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$ . By A2,  $\phi^n \in \text{ftv}(\Gamma)$ . Because  $\phi^n \neq \xi^d$ ,  $\phi^n \notin \text{supp}(\mathcal{L}(\tau, d))$ , and  $\phi^n \neq \xi^d$ ,  $\phi^n \in \text{ftv}(\sigma\Gamma)$ .

**Case  $\phi^n \in \text{ftv}(\tau) \setminus \text{ftv}(\text{supp}(\mathcal{L}(\tau, d)))$ :**  $\phi^n \in \text{ftv}(\tau) \subseteq \text{ftv}(k[(\text{unify } \xi^d \tau u)])$ .

Because substitutions do not change the number of  $\lambda$ -bindings in a type environment,  $n \leq \mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$ . By A2,  $\phi^n \in \text{ftv}(\Gamma)$ . Because  $\phi^n$  is not substituted away by  $\sigma$ ,  $\phi^n \in \text{ftv}(\Gamma) \setminus (\{\xi^d\} \cup \text{supp}(\mathcal{L}(\tau, d))) \subseteq \text{ftv}(\sigma\Gamma)$ .

**Case  $\phi^n \in \text{ftv}(\text{rng}(\mathcal{L}(\tau, d)))$ :** By definition of  $\mathcal{L}(\tau, d)$ ,  $n = d$  and  $\psi^q \in \text{ftv}(\tau)$  such that  $q > d = n$ .

**Case  $q \leq \mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$ :** By A2,  $\psi^q \in \text{ftv}(\Gamma)$ .  $\psi^q \in \text{rng}(\Gamma)$  by definition of  $\text{rng } \Gamma$  and  $\psi^q$ . By substitution range preservation lemma (lem. 6),  $\sigma\psi^q = \phi^n \in \text{ftv}(\sigma\Gamma)$ .

**Case  $q > \mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$ :**  $\mathcal{V}(\psi^q, \Gamma) \neq q$  because there is no  $q$ th  $\lambda$ -binding in  $\Gamma$ . By contrapositive of A1,  $\psi^q \notin \text{ftv}(\Gamma)$ . By A2,  $(\xi^d \in \text{ftv}(k[(\text{unify } \xi^d \tau u)]))$  and  $d \leq \mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$  by given)  $\xi^d \in \text{ftv}(\Gamma) \Rightarrow \xi^d \in \text{rng } \Gamma \Rightarrow \sigma\xi^d = \tau \in \text{rng}(\sigma\Gamma)$  by substitution range preservation lemma (lem. 6).

$\phi^d \in (\sigma\Gamma)$  because  $\phi^d \notin \text{supp}(\mathcal{L}(\tau, d))$ .

**Inv<sub>3</sub>** Because substitution does not change how many  $\lambda$ -bindings there are in a type environment,  $\mathcal{D}(\sigma\Gamma) = \mathcal{D}(\Gamma)$ .  $\mathcal{D}(\Gamma) = \#$  of  $\rightarrow$  frames in  $k$  by A3. Thus,  $\mathcal{D}(\sigma\Gamma) = \#$  of  $\rightarrow$  frames in  $k$  (transitivity).

**Case [pm-u-id]**

A2.  $\forall \xi^m \in \text{ftv}(k[(\mathbf{unify} \ \tau \ \tau \ u])).m \leq \mathcal{D}(\Gamma) \Rightarrow \xi^m \in \text{ftv}(\Gamma)$ .

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** Assume  $\phi^n \in \text{ftv}(k[u]).n \leq \mathcal{D}(\Gamma).$  To show:  $\phi^n \in \text{ftv}(\Gamma)$ .

$\phi^n \in \text{ftv}(k[u]) \subseteq \text{ftv}(k[(\mathbf{unify} \ \tau \ \tau \ u)]).$  By A2,  $\phi^n \in \text{ftv}(\Gamma)$ .

**Inv<sub>3</sub>** By  $\Gamma$  and  $k$  invariance

**Case [pm-u-arr]**

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** By  $k[c]$  invariance

**Inv<sub>3</sub>** By  $\Gamma$  and  $k$  invariance

**Case [pm-u-orient]**

**Inv<sub>1</sub>** By  $\Gamma$  invariance

**Inv<sub>2</sub>** By  $k[c]$  invariance

**Inv<sub>3</sub>** By  $\Gamma$  and  $k$  invariance

□

**Lemma 10 (Invariants hold for Reachable states).**  $\forall s \in \text{Reachable}$ , all three invariants hold for  $s$ .

*Proof.* Invariants hold for initial states. Invariants hold for each reduction. Hence they hold for any length of reduction sequence from the initial states. □

The correspondence between the  $\lambda$ -depth ranking ( $\mathcal{G}_d$ ) to the Milner criterion ( $\mathcal{G}_\Gamma$ ) must hold for both the definien expression and the unify prefix (i.e., any unification problems that must be done). It is necessary for the correspondence to hold for the type variables in the unify prefix also because otherwise it would be possible to introduce type variables with incorrect depth ranking by means of the unify prefix.

**Lemma 11 (Equivalence of Generalizability Criteria).**

For all HM machine states  $((\mathbf{let}^d (x e) e'), \Gamma, \Sigma, k) \in \text{VMS}$ , if  $((\mathbf{let}^d (x e) e'), \Gamma, \Sigma, k) \mapsto_{pm}^* (\tau, \Sigma' \Gamma, \Sigma' \Sigma, (\mathbf{let}^d (x \square) e') :: (\mathcal{C}\Sigma')k)$ , then  $\mathcal{E}_r(\mathcal{G}_d(\tau)) = \mathcal{G}_{\mathcal{E}_r(\Sigma' \Gamma)}(\mathcal{E}_r(\tau))$ .

*Proof.* This lemma follows from lem. 1 given below and the definitions of  $\mathcal{G}_d$  and  $\mathcal{G}_{\mathcal{E}_r(\Sigma' \Gamma)}$ . □

**Lemma 12 (Soundness of Hindley-Milner Machine).**

For all HM machine states  $(e, \Gamma, \Sigma, k) \in \text{VMS}$ , if  $(e, \Gamma, \Sigma, k) \mapsto_{pm}^* (\tau, \widehat{\Sigma}' \Gamma, \Sigma' \Sigma, \widehat{\Sigma}' k)$ , then  $\mathcal{W}(\mathcal{E}_r(\Gamma), \mathcal{E}_a(e)) = (\mathcal{E}_r(\widehat{\Sigma}'), \mathcal{E}_r(\tau))$ .

*Proof (Induction on the structure of  $e$ ).*

**Case  $e = (\lambda^d(x) e')$**

By definition of  $\mapsto_{pm}$ :

$$\begin{aligned} ((\lambda^d(x) e'), \Gamma, \Sigma, k) &\mapsto_{pm} (e', \Gamma[x : \xi^d], \Sigma, (\rightarrow \xi^d \square) :: k) \\ &\mapsto_{pm}^* (\tau, \widehat{\Sigma}'(\Gamma[x : \xi^d]), \Sigma' \Sigma, (\rightarrow \widehat{\Sigma}' \xi^d \square) :: \widehat{\Sigma}' k) (*) \\ &\mapsto_{pm} ((\rightarrow \widehat{\Sigma}' \xi^d \tau), \widehat{\Sigma}' \Gamma, \Sigma' \Sigma, \widehat{\Sigma}' k) \end{aligned}$$

Applying the induction hypothesis to step (\*),  $\mathcal{W}(\mathcal{E}_r(\Gamma)[x : \xi], \mathcal{E}_r(e')) = (\mathcal{E}_r(\widehat{\Sigma}'), \mathcal{E}_r(\tau))$ .

By the definition of  $\mathcal{W}$ ,  $\mathcal{W}(\mathcal{E}_r(\Gamma), (\lambda(x) \mathcal{E}_r(e'))) = (\mathcal{E}_r(\widehat{\Sigma}'), \mathcal{E}_r(\tau))$ .

**Case  $e = (@ e_1 e_2)$**

$$\begin{aligned}
& ((@ e_1 e_2), \Gamma, \Sigma, k) \\
& \mapsto_{pm} (e_1, \Gamma, \Sigma, (@ \square e_2) :: k) \\
& \mapsto_{pm}^* (\tau_1, \widehat{\Sigma}'\Gamma, \Sigma'\Sigma, (@ \square \widehat{\Sigma}'e_2) :: \widehat{\Sigma}'k) \quad (*) \\
& \mapsto_{pm} (\widehat{\Sigma}'e_2, \widehat{\Sigma}'\Gamma, \Sigma'\Sigma, (@ \tau_1 \square) :: \widehat{\Sigma}'k) \\
& = (e_2, \widehat{\Sigma}'\Gamma, \Sigma'\Sigma, (@ \tau_1 \square) :: \widehat{\Sigma}'k) \quad e_2 \text{ has no tv} \\
& \mapsto_{pm}^* (\tau_2, \widehat{\Sigma}''\widehat{\Sigma}'\Gamma, \Sigma''\Sigma'\Sigma, (@ \widehat{\Sigma}''\tau_1 \square) :: \widehat{\Sigma}''\widehat{\Sigma}'k) \quad (**) \\
& \mapsto_{pm} ((\mathbf{unify} \widehat{\Sigma}''\tau_1 (\rightarrow \tau_2 \xi^\infty) \xi^\infty), \widehat{\Sigma}''\widehat{\Sigma}'\Gamma, \Sigma''\Sigma'\Sigma, \widehat{\Sigma}''\widehat{\Sigma}'k) \\
& \mapsto_{pm}^* (\widehat{\Sigma}'''\xi^\infty, \widehat{\Sigma}'''\widehat{\Sigma}''\widehat{\Sigma}'\Gamma, \Sigma'''\Sigma''\Sigma'\Sigma, \widehat{\Sigma}'''\widehat{\Sigma}''\widehat{\Sigma}'k) \quad \text{unify correctness}
\end{aligned}$$

Applying the induction hypothesis to (\*),  $\mathcal{W}(\mathcal{E}_r(\Gamma), \mathcal{E}_r(e_1)) = (\mathcal{E}_r(\widehat{\Sigma}'), \mathcal{E}_r(\tau_1))$ .

Applying the induction hypothesis to (\*\*),  $\mathcal{W}(\mathcal{E}_r(\widehat{\Sigma}'\Gamma), \mathcal{E}_r(e_2)) = (\mathcal{E}_r(\widehat{\Sigma}'''), \mathcal{E}_r(\tau_2))$ .

By definition of  $\mathcal{U}$  (and unify correctness),  $\mathcal{U}((\rightarrow \mathcal{E}_r(\tau_2) \xi), \mathcal{E}_r(\widehat{\Sigma}''\tau_1)) = \mathcal{E}_r(\widehat{\Sigma}''')$ .

By definition of  $\mathcal{W}$ ,  $\mathcal{W}(\mathcal{E}_r(\Gamma), \mathcal{E}_r(@ e_1 e_2)) = (\mathcal{E}_r(\widehat{\Sigma}'''\widehat{\Sigma}''\widehat{\Sigma}'), (\mathcal{E}_r(\widehat{\Sigma}'''))\xi$ .

**Case  $e = (\mathbf{let}^d (x e') e'')$**

$$\begin{aligned}
& ((\mathbf{let}^d (x e') e''), \Gamma, \Sigma, k) \\
& \mapsto_{pm} (e', \Gamma, \Sigma, (\mathbf{let}^d (x \square) e'') :: k) \\
& \mapsto_{pm}^* (\tau', \Gamma, \Sigma'\Sigma, (\mathbf{let}^d (x \square) \widehat{\Sigma}'e'') :: \widehat{\Sigma}'k) \quad (*) \\
& = (\tau', \Gamma, \Sigma'\Sigma, (\mathbf{let}^d (x \square) e'') :: \widehat{\Sigma}'k) \quad \text{no tv's in } e'' \\
& \mapsto_{pm} (e'', (\widehat{\Sigma}'\Gamma)[x : \mathcal{P}_d(\tau')], \Sigma'\Sigma, (\mathbf{let}^d \square) :: \widehat{\Sigma}'k) \\
& \mapsto_{pm}^* (\tau'', \widehat{\Sigma}''(\widehat{\Sigma}'\Gamma)[x : \mathcal{P}_d(\tau')], \Sigma''\Sigma'\Sigma, \widehat{\Sigma}''((\mathbf{let}^d \square) :: \widehat{\Sigma}'k)) \quad (**) \\
& \mapsto_{pm} (\tau'', \widehat{\Sigma}''\widehat{\Sigma}'\Gamma, \Sigma''\Sigma'\Sigma, \widehat{\Sigma}''\widehat{\Sigma}'k)
\end{aligned}$$

Applying the induction hypothesis to (\*),  $\mathcal{W}(\mathcal{E}_r(\Gamma), \mathcal{E}_r(e')) = (\mathcal{E}_r(\widehat{\Sigma}'), \mathcal{E}_r(\tau'))$ .

Applying the induction hypothesis to (\*\*),  $\mathcal{W}(\mathcal{E}_r((\widehat{\Sigma}'\Gamma)[x : \mathcal{P}_d(\tau')]), \mathcal{E}_r(e'')) = (\mathcal{E}_r(\widehat{\Sigma}'''), \mathcal{E}_r(\tau''))$ .

Let  $\Gamma' = \mathcal{E}_r(\widehat{\Sigma}'\Gamma)$ .

By equivalence of generalization criteria (lem. 11),  $\mathcal{E}_r(\mathcal{G}_d(\tau')) = \mathcal{G}_{\Gamma'}(\mathcal{E}_r(\tau'))$ .

By definition of  $\mathcal{P}_{\Gamma'}$ ,  $\mathcal{E}_r(\mathcal{P}_d(\tau')) = \mathcal{P}_{\Gamma'}(\mathcal{E}_r(\tau'))$ .

Thus,  $\mathcal{W}(\mathcal{E}_r((\widehat{\Sigma}'\Gamma)[x : \mathcal{P}_{\Gamma'}(\mathcal{E}_r(\tau'))]), \mathcal{E}_r(e'')) = (\mathcal{E}_r(\widehat{\Sigma}'''), \mathcal{E}_r(\tau''))$ .

By definition,  $\mathcal{W}(\mathcal{E}_r(\Gamma), (\mathbf{let} (x \mathcal{E}_r(e')) \mathcal{E}_r(e''))) = (\mathcal{E}_r(\widehat{\Sigma}'''), \mathcal{E}_r(\tau''))$ .

**Case  $e = x$**

$$\begin{aligned}
& (x, \Gamma, \Sigma, k) \mapsto_{pm} (\Gamma(x), \Gamma, \Sigma, k) \\
& = (c, \Gamma, \Sigma, k)
\end{aligned}$$

By [pm-var],  $x : c \in \Gamma$ .

**Case  $x : \rho \in \Gamma$  such that  $\rho = \forall \vec{\alpha} \tau_p$ :**  $(\forall \vec{\alpha} \tau_p, \Gamma, \Sigma, k) \mapsto_{pm} (\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p, \Gamma, \Sigma, k)$  where  $\vec{\xi}^\infty$  consists of fresh type variables.

By definition,  $\mathcal{W}(\mathcal{E}_r(\Gamma), x) = (\varepsilon, \{\vec{\alpha} \mapsto \vec{\xi}\} \mathcal{E}_r(\tau_p))$ .

**Case  $x : \tau \in \Gamma$  where  $\tau$  is a monotype:** By definition,  $\mathcal{W}(\mathcal{E}_r(\Gamma), x) = (\bullet, \mathcal{E}_r(\tau))$ .

□

To express the completeness lemma, I need a few auxiliary definitions. A type environment is unranked if it does not contain a mapping to a type with ranked type variables. Similarly, an unranked type variable to type substitution contains only unranked type variables in the domain and in the range. Again, the  $\Sigma$  register maintains an explicit representation of the substitutions that correspond to those that are obtained by  $\mathcal{W}$ .

The ranking functions in def. 9 produce ranked types, type environments, and type substitutions from their unranked variants.

**Definition 9 (Ranking Functions).**

$$\mathcal{R}_{tv} : TVAR \times TENV \rightarrow TVAR-R \quad \mathcal{R}_{tv}(\xi, \tilde{\Gamma}) = \xi \tilde{\mathcal{V}}(\xi, \tilde{\Gamma})$$

$$\mathcal{R}_{ty} : TYPE-V \times TENV \rightarrow TYPE-VR$$

$\mathcal{R}_{ty}(\tilde{\tau}, \tilde{\Gamma})$  recursively replaces type variables in  $\tilde{\tau}$  with ranked type variables

$$\mathcal{R}_{tenv} : TENV-V \rightarrow TENV \quad \mathcal{R}_{tenv}(\tilde{\Gamma}) = \text{map} (\lambda (x, \tilde{\tau}). [x : \mathcal{R}_{ty}(\tilde{\tau}, \tilde{\Gamma})]) \tilde{\Gamma}$$

$$SUBST = TVAR \rightarrow TYPE-V \quad SUBST-R = TVAR-R \rightarrow TYPE-VR$$

$$\mathcal{R}_{subst} : SUBST \times TENV \rightarrow SUBST-R$$

$$\mathcal{R}_{subst}(\tilde{\sigma}, \tilde{\Gamma}) = \lambda \xi^n. \mathcal{R}_{ty}(\tilde{\sigma}(\mathcal{E}_r(\xi^n)))$$

$\mathcal{R}(\tilde{\Gamma}, \tilde{\Gamma})$ ,  $\mathcal{R}(\tilde{\rho}, \tilde{\Gamma})$ , and  $\mathcal{R}(\tilde{\sigma}, \tilde{\Gamma})$  are the extensions of  $\mathcal{R}(\xi, \tilde{\Gamma})$  over type environments, types, bounded types, and type substitutions respectively. Because I only rank a type environment with respect to itself,  $\mathcal{R}(\tilde{\Gamma})$  will be shorthand for  $\mathcal{R}(\tilde{\Gamma}, \tilde{\Gamma})$ .

**Lemma 13 (Unique Type Variable Rank).**

For all  $(c, -, -, k) \in \text{VMS}$ , if  $\xi^n, \xi^m \in \text{ftv}(k[c])$ , then  $n = m$ .

*Proof.* I can verify that type variables introduced by most of the machine rules have unique ranks by inspection (either a fresh type variable at rank  $\infty$  in [pm- $\tau\beta$ ] and [pm-poly] or  $\lambda$ -bound type variable at the depth annotation in [pm-lam]). The [pm-u-mv] rule is the only

nontrivial case. This rule can alter type variable ranks by promotion (decreasing the rank). Notice that the rule applies the unification substitution globally. Thus, any rank change is applied to all type variables in the control, context, and type environment.  $\square$

**Lemma 14.** *For all  $(-, -, \Sigma, -) \in \text{VMS}$ , let  $\sigma$  be the composition of some sublist of  $\Sigma$ . If  $\xi^n, \xi^m \in \text{ftv}(\text{supp}(\sigma))$ , then  $n = m$ .*

*Proof.* Substitutions in  $\Sigma$  are only introduced in [pm-u-mv]. The support of the substitutions always come from the control and the support of the limit substitution. By lem. 13, the type variables in the support contributed by the control have unique ranks. By construction of the limit substitution, the support also comes from the control. Consequently, those type variables contributed by the support of the limit substitution also have unique ranks.  $\square$

**Definition 10.** *Define the application of a ranked  $\sigma$  to an unranked type variable  $\xi$  as the following:*

$$\sigma(\xi) = \begin{cases} \xi & \text{if no } \xi^n \in \text{supp}(\sigma) \text{ for any } n \\ \tau & \text{if } \xi^n \in \text{supp}(\sigma) \text{ and } \sigma(\xi^n) = \tau \text{ (for some unique } n) \end{cases}$$

*If  $\xi^n \in \text{supp}(\sigma)$ , it is unique by lem. 14.*

**Definition 11 (Rank Erasure Functions).**

$$\mathcal{E}_r : \text{SUBST-R} \rightarrow \text{SUBST} \quad \mathcal{E}_r(\sigma) = \lambda \xi. \mathcal{E}_r(\sigma(\xi))$$

$\mathcal{E}_r(\sigma)$  is an unranked substitution and thus is extended as usual over all type variable containing constructs.

**Lemma 15 (Ranking Erasure Distributes over Type Substitution).**

*For  $(c, \Gamma, \dots, \Sigma_2 \Sigma_1 \dots, k) \in \text{VMS}$ , let  $\xi^n \in \text{ftv}(k) \cup \text{ftv}(c)$ ,*

$$\mathcal{E}_r(\widehat{\Sigma}_1 \xi^n) = (\mathcal{E}_r(\widehat{\Sigma}_1)) \xi \quad \mathcal{E}_r(\widehat{\Sigma}_2 \widehat{\Sigma}_1) = (\mathcal{E}_r(\widehat{\Sigma}_2)) (\mathcal{E}_r(\widehat{\Sigma}_1))$$

*Proof.*  $\widehat{\Sigma}_1$  is a composition of unification substitutions produced by the [pm-u-mv] rule.

Let  $\widetilde{\Gamma} = \mathcal{E}_r(\Gamma)$ . Either  $\xi^n \in \text{ftv}(\text{supp}(\widehat{\Sigma}_1, \xi^n))$  or  $\xi^n \notin \text{ftv}(\text{supp}(\widehat{\Sigma}_1, \xi^n))$ .

By definition of  $\mathcal{E}_r$ ,

$$(\mathcal{E}_r(\widehat{\Sigma}_1)) \mathcal{E}_r(\xi^n) = (\lambda \phi. \mathcal{E}_r(\widehat{\Sigma}_1 \phi)) \xi$$

By  $\beta$ -reduction,

$$(\lambda \phi. \mathcal{E}_r(\widehat{\Sigma}_1 \phi)) \xi = \mathcal{E}_r(\widehat{\Sigma}_1(\xi))$$

**Case  $\xi^n \in \text{ftv}(\text{supp}(\widehat{\Sigma}_1, \xi^n))$ :** There exists  $\tau$  such that  $\widehat{\Sigma}_1 \xi^n = \tau$  by definition of substitution.

$$\mathcal{E}_r(\widehat{\Sigma}_1 x^n) = \mathcal{E}_r(\tau).$$

By def. 10,

$$\mathcal{E}_r(\widehat{\Sigma}_1(\xi)) = \mathcal{E}_r(\tau)$$

**Case  $\xi^n \notin \text{ftv}(\text{supp}(\widehat{\Sigma}_1, \xi^n))$ :**  $\widehat{\Sigma}_1 \xi^n = \xi^n$  by the definition of substitution.

$$\mathcal{E}_r(\widehat{\Sigma}_1 x^n) = \mathcal{E}_r(\xi^n) = \xi.$$

By def. 10,

$$\mathcal{E}_r(\widehat{\Sigma}_1(\xi)) = \mathcal{E}_r(\xi^n)$$

Thus,  $\mathcal{E}_r(\widehat{\Sigma}_1 \xi^n) = (\mathcal{E}_r(\widehat{\Sigma}_1)) \xi$ .

The statement for  $\mathcal{E}_r(\widehat{\Sigma}_2 \widehat{\Sigma}_1)$  follows from the construction of  $\mathcal{E}_r(\widehat{\sigma})$ . □

**Lemma 16 (Ranking Distributes over Type Substitution).**

$$\mathcal{R}(\widetilde{\sigma} \xi, \widetilde{\Gamma}) = (\mathcal{R}(\widetilde{\sigma}, \widetilde{\Gamma}))(\mathcal{R}(\xi, \widetilde{\Gamma})) \quad \mathcal{R}(\widetilde{\sigma} \widetilde{\tau}, \widetilde{\Gamma}) = (\mathcal{R}(\widetilde{\sigma}, \widetilde{\Gamma}))(\mathcal{R}(\widetilde{\tau}, \widetilde{\Gamma}))$$

$$\mathcal{R}(\widetilde{\sigma} \widetilde{\rho}, \widetilde{\Gamma}) = (\mathcal{R}(\widetilde{\sigma}, \widetilde{\Gamma}))(\mathcal{R}(\widetilde{\rho}, \widetilde{\Gamma})) \quad \mathcal{R}(\widetilde{\sigma} \widetilde{\Gamma}) = (\mathcal{R}(\widetilde{\sigma}, \widetilde{\Gamma}))(\mathcal{R}(\widetilde{\Gamma}))$$

*Proof.* The first equation is by construction of the ranking functions. The rest follow in sequence. □

**Lemma 17 (Completeness of Hindley-Milner Reduction).**

For any  $\tilde{e} \in \text{ULCL}$  and  $\tilde{\Gamma} \in \text{TYENV}$ , if  $\mathcal{W}(\tilde{\Gamma}, \tilde{e}) = (\tilde{\sigma}, \tilde{\tau})$  then  $\forall \Sigma, \forall k. \exists \Sigma'. \exists \tau \in \text{TYPE-}R(e, \Gamma, \Sigma, k) \mapsto_{pm}^* (\tau, \widehat{\Sigma}' \Gamma, \Sigma' \Sigma, \tilde{\Gamma}' k)$  where  $e = d_\lambda(\tilde{e}, \mathcal{D}(\tilde{\Gamma}))$ ,  $\Gamma = \mathcal{R}(\tilde{\Gamma})$ ,  $\mathcal{E}_r(\tau) = \tilde{\tau}$ , and  $\mathcal{E}_r(\widehat{\Sigma}') = \sigma$ .

*Proof* (By induction on the structure of  $\tilde{e}$ ).

**Case  $\tilde{e} = (\lambda(x) \tilde{e}')$ :** By assumption,

$$\mathcal{W}(\tilde{\Gamma}, (\lambda(x) \tilde{e}')) = (\tilde{\sigma}, \tilde{\tau})$$

By the definition of  $\mathcal{W}$ ,

$$\mathcal{W}(\tilde{\Gamma}[x : \xi], \tilde{e}') = (\tilde{\sigma}', \tilde{\tau}') \quad (1)$$

for some fresh  $\xi$ , where  $\tilde{\tau} = (\rightarrow \tilde{\sigma}' \xi \tilde{\tau}')$  and  $\tilde{\sigma}' = \tilde{\sigma}$ .

By definition of  $d_\lambda$ ,

$$d_\lambda((\lambda(x) \tilde{e}'), \mathcal{D}(\tilde{\Gamma})) = (\lambda^n(x) e') \quad (2)$$

$$\text{where } n = \mathcal{D}(\tilde{\Gamma}) + 1 \text{ and } e' = d_\lambda(\tilde{e}', \mathcal{D}(\tilde{\Gamma}) + 1) \quad (3)$$

Let  $\Gamma = \mathcal{R}(\tilde{\Gamma})$ .

For any  $\Sigma$  and  $k$ , by [pm-lam],

$$((\lambda^n(x) e'), \Gamma, \Sigma, k') \mapsto_{pm} (e', \Gamma[x : \xi^n], \Sigma, (\rightarrow \xi^n \square) :: k) \quad (4)$$

where  $\xi^n$  is fresh.  $\xi^n$  can be the ranked version of that fresh  $\xi$  in (1).

By the definition of  $\mathcal{R}$ ,  $\Gamma[x : \xi^n] = \mathcal{R}(\tilde{\Gamma}[x : \xi])$ .

Let  $\sigma' = \widehat{\Sigma}'$ . Now, by induction, it must be the case that there exists  $\Sigma'$  and  $\tau'$  such that

$$\begin{aligned} (e', \Gamma[x : \xi^n], \Sigma, k) \\ \mapsto_{pm}^* (\tau', \widehat{\Sigma}'(\Gamma[x : \xi^n]), \Sigma' \Sigma, \widehat{\Sigma}'((\rightarrow \xi^n \square) :: k)) \end{aligned} \quad (5)$$

$$\text{where } \tilde{\tau}' = \mathcal{E}_r(\tau') \text{ and } \tilde{\sigma}' = \mathcal{E}_r(\sigma') \quad (6)$$

By the definition of the application of substitutions to type environments and contexts

$$\begin{aligned} (\tau', \sigma'(\Gamma[x : \xi^n]), \Sigma' \Sigma, \sigma'((\rightarrow \xi^n \square) :: k)) \\ = (\tau', (\sigma' \Gamma)[x : \sigma \xi^n], \Sigma' \Sigma, ((\rightarrow \sigma' \xi^n \square) :: \sigma' k)) \end{aligned} \quad (7)$$

By [pm-arr],

$$\begin{aligned} (\tau', (\sigma' \Gamma)[x : \sigma \xi^n], \Sigma' \Sigma, ((\rightarrow \sigma' \xi^n \square) :: \sigma' k)) \\ \mapsto_{pm} ((\rightarrow \sigma' \xi^n \tau'), \sigma' \Gamma, \Sigma' \Sigma, \sigma' k) \end{aligned} \quad (8)$$

Combining (4), (5), (7), and (8) yields

$$(e, \Gamma, \Sigma, k) \mapsto_{pm}^* ((\rightarrow \sigma' \xi^n \tau'), \sigma' \Gamma, \Sigma' \Sigma, \sigma' k)$$

It remains to show that

$$\tilde{\tau} = (\rightarrow \tilde{\sigma}' \xi \tilde{\tau}') = \mathcal{E}_r((\rightarrow \sigma' \xi^n \tau'))$$

The above follows from the definition of  $\mathcal{E}_r$  over types, (6), and

$$\mathcal{E}_r(\widehat{\Sigma}' \xi^n) = (\mathcal{E}_r(\widehat{\Sigma}')) \xi = \tilde{\sigma}' \xi$$

which is true by lem. 15.

**Case  $\tilde{e} = x$ :** By assumption,  $\mathcal{W}(\tilde{\Gamma}, x) = (\tilde{\sigma}, \tilde{\tau})$  where either  $[x : \tilde{\tau}] \in \tilde{\Gamma}$  or  $[x : \forall(\vec{\alpha}) \tilde{\tau}_p] \in \tilde{\Gamma}$  ( $\tilde{\tau} = \{\vec{\alpha} \mapsto \vec{\xi}\} \tilde{\tau}_p$ ). Let  $\tilde{\rho} = \forall(\vec{\alpha}) \tilde{\tau}_p$  and  $\Gamma = \mathcal{R}(\tilde{\Gamma})$ .

**Case  $x : \tilde{\tau} \in \tilde{\Gamma}$ :** By definition of  $\mathcal{R}$ ,  $\mathcal{R}(\tilde{\Gamma}) = \mathcal{R}(\dots [x : \tilde{\tau}] \dots) = \dots [x : \tau] \dots$  where  $\mathcal{R}(\tilde{\tau}, \tilde{\Gamma}) = \tau$ . Thus,  $[x : \tau] \in \Gamma = \mathcal{R}(\tilde{\Gamma})$ .

For any  $\Sigma$  and  $k$ , by [pm-var],  $(x, \Gamma, \Sigma, k) \mapsto_{pm} (\Gamma(x), \Gamma, \Sigma, k) = (\tau, \Gamma, \Sigma, k)$ .

Therefore,  $(x, \Gamma, \Sigma, k) \mapsto_{pm}^* (\tau, \Gamma, \Sigma, k)$ .

**Case  $x : \tilde{\rho} \in \tilde{\Gamma}$ :** By definition of  $\mathcal{R}$ ,

$$\mathcal{R}(\tilde{\Gamma}) = \mathcal{R}(\dots [x : \tilde{\rho}] \dots) = \dots [x : \rho] \dots$$

$$\text{where } \mathcal{R}(\tilde{\rho}, \tilde{\Gamma}) = \rho \tag{1}$$

Thus  $[x : \rho] \in \Gamma = \mathcal{R}(\tilde{\Gamma})$ .

For any  $\Sigma$  and  $k$ , by [pm-var],

$$(x, \Gamma, \Sigma, k) \mapsto_{pm} (\Gamma(x), \Gamma, \Sigma, k) \tag{2}$$

By the definition of  $\Gamma(x)$ ,

$$(\Gamma(x), \Gamma, \Sigma, k) = (\rho, \Gamma, \Sigma, k) \tag{3}$$

By [pm-poly],

$$(\rho, \Gamma, \Sigma, k) \mapsto_{pm} (\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p, \Gamma, \Sigma, k) \tag{4}$$

Combining (2) - (4) yields

$$(x, \Gamma, \Sigma, k) \mapsto_{pm}^* (\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p, \Gamma, \Sigma, k)$$

Let  $\tau = \{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p$ . By (1) and the definition of  $\mathcal{R}$ ,  $\mathcal{R}(\vec{\rho}, \vec{\Gamma}) = \mathcal{R}(\forall(\vec{\alpha}) \tilde{\tau}_p) = \forall(\vec{\alpha})(\mathcal{R}(\tilde{\tau}_p)) = \rho$ . By definition of the nonterminal  $\rho$ ,  $\rho = \forall(\vec{\alpha}) \tau_p$  where  $\mathcal{R}(\tilde{\tau}_p, \vec{\Gamma}) = \tau_p$ . By lem. 15,  $\mathcal{E}_r(\{\vec{\alpha} \mapsto \vec{\xi}^\infty\} \tau_p) = (\mathcal{E}_r(\{\vec{\alpha} \mapsto \vec{\xi}^\infty\}))(\mathcal{E}_r(\tau_p)) = \{\vec{\alpha} \mapsto \vec{\xi}\} \tilde{\tau}_p$ .

$\mathcal{E}_r(\tau) = \tilde{\tau}$  by definition of  $\mathcal{E}_r$ .

There were no substitutions applied in this reduction sequence, hence  $\Sigma' = \bullet$ . By definition,  $\mathcal{R}(\bullet) = \mathcal{R}(\varepsilon) = \varepsilon$ .

**Case  $\tilde{e} = (@ \tilde{e}_1 \tilde{e}_2)$ :** By assumption,  $\mathcal{W}(\vec{\Gamma}, (@ \tilde{e}_1 \tilde{e}_2)) = (\vec{\sigma}, \vec{\tau})$  and  $\mathcal{R}(\vec{\Gamma}) = \Gamma$ .

It follows from the definition of  $\mathcal{W}$  that

$$\mathcal{W}(\vec{\Gamma}, \tilde{e}_1) = (\vec{\sigma}_1, \tilde{\tau}_1) \quad (*)$$

and

$$\mathcal{W}(\vec{\sigma}_1 \vec{\Gamma}, \tilde{e}_2) = (\vec{\sigma}_2, \tilde{\tau}_2) \quad (**)$$

$$\text{where } \vec{\sigma}_3 = \mathcal{U}(\vec{\sigma}_2 \tilde{\tau}_1, (\rightarrow \tilde{\tau}_2 \xi)), \tilde{\tau} = \vec{\sigma}_3 \xi, \text{ and } \vec{\sigma} = \vec{\sigma}_3 \vec{\sigma}_2 \vec{\sigma}_1 \quad (1)$$

Let  $e = d_\lambda(\tilde{e}, \mathcal{D}(\vec{\Gamma}))$ . By definition of  $d_\lambda$ ,  $e = (@ e_1 e_2)$  such that  $e_1 = d_\lambda(\tilde{e}_1, \mathcal{D}(\vec{\Gamma}))$  and  $d_\lambda(\tilde{e}_2, \mathcal{D}(\vec{\Gamma}))$ . Let  $\Gamma = \mathcal{R}(\vec{\Gamma})$ .

For any  $\Sigma$  and  $k$ , by [pm-app],

$$((@ e_1 e_2), \Gamma, \Sigma, k) \mapsto_{pm} (e_1, \Gamma, \Sigma, (@ \square e_2) :: k) \quad (2)$$

Applying the induction hypothesis to (\*), I get that there must exist  $\tau_1$  and  $\Sigma_1$  such that

$$(e_1, \Gamma, \Sigma, (@ \square e_2) :: k) \mapsto_{pm}^* (\tau_1, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, \widehat{\Sigma}_1 ((@ \square e_2) :: k)) \quad (3)$$

$$\text{where } \mathcal{E}_r(\widehat{\Sigma}_1) = \vec{\sigma}_1 \text{ and } \mathcal{E}_r(\tau_1) = \tilde{\tau}_1 \quad (4)$$

By [pm-app-left],

$$(\tau_1, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, \widehat{\Sigma}_1 ((@ \square e_2) :: k)) \mapsto_{pm} (e_2, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, (@ \tau_1 \square) :: k) \quad (5)$$

By lem. 16 and (4),  $\mathcal{R}(\widetilde{\sigma}_1 \widetilde{\Gamma}) = \widehat{\Sigma}_1 \Gamma$ . Applying the induction hypothesis to (\*\*), I get that there must exist  $\tau_2$  and  $\Sigma_2$  such that

$$(e_2, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, (@ \tau_1 \square) :: k) \mapsto_{pm}^* (\tau_2, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 ((@ \tau_1 \square) :: \widehat{\Sigma}_1 k)) \quad (6)$$

$$\text{where } \mathcal{E}_r(\widehat{\Sigma}_2) = \widetilde{\sigma}_2 \text{ and } \mathcal{E}_r(\tau_2) = \widetilde{\tau}_2$$

By [pm-app-right],

$$(\tau_2, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 ((@ \tau_1 \square) :: \widehat{\Sigma}_1 k)) \mapsto_{pm} ((@ \tau_1 \tau_2), \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \quad (7)$$

By [pm- $\tau\beta$ ], for some fresh  $\xi^\infty$ ,

$$\begin{aligned} & ((@ \tau_1 \tau_2), \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \\ & \mapsto_{pm} ((\mathbf{unify} \tau_1 (\rightarrow \tau_2 \xi^\infty) \xi^\infty), \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \end{aligned} \quad (8)$$

By thm. B.14,

$$\begin{aligned} & ((\mathbf{unify} \tau_1 (\rightarrow \tau_2 \xi^\infty) \xi^\infty), \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \\ & \mapsto_{pm}^* (\widehat{\Sigma}_3 \xi^\infty, \widehat{\Sigma}_3 \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_3 \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_3 \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \end{aligned} \quad (9)$$

Combining (2), (3), (5) - (9),

$$((@ e_1 e_2), \Gamma, \Sigma, k) \mapsto_{pm}^* (\widehat{\Sigma}_3 \xi^\infty, \widehat{\Sigma}_3 \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_3 \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_3 \widehat{\Sigma}_2 \widehat{\Sigma}_1 k)$$

By lem. 15,  $\mathcal{E}_r(\widehat{\Sigma}_3 \xi^\infty) = \widetilde{\sigma}_3 \xi = \widetilde{\tau}$  and  $\mathcal{E}_r(\widehat{\Sigma}_3 \widehat{\Sigma}_2 \widehat{\Sigma}_1) = \widetilde{\sigma}_3 \widetilde{\sigma}_2 \widetilde{\sigma}_1 = \widetilde{\sigma}$ .

**Case  $\tilde{e} = (\mathbf{let} (x \tilde{e}_1) \tilde{e}_2)$ :** By assumption,  $\mathcal{W}(\widetilde{\Gamma}, (\mathbf{let} (x \tilde{e}_1) \tilde{e}_2)) = (\widetilde{\sigma}, \widetilde{\tau})$ .

It follows from the definition of  $\mathcal{W}$  that

$$\mathcal{W}(\widetilde{\Gamma}, \widetilde{e}_1) = (\widetilde{\sigma}_1, \widetilde{\tau}_1) \quad (*)$$

and

$$\mathcal{W}(\Gamma[x : \mathcal{P}_{\widetilde{\sigma}_1 \widetilde{\Gamma}}(\widetilde{\tau}_1)], \widetilde{e}_2) = (\widetilde{\sigma}_2, \widetilde{\tau}) \quad (**)$$

$$\text{where } \widetilde{\sigma} = \widetilde{\sigma}_2 \widetilde{\sigma}_1 \quad (\diamond)$$

For any  $\Sigma$  and  $k$ , by [pm-let],

$$((\mathbf{let}^n (x e_1) e_2), \Gamma, \Sigma, k) \mapsto_{pm} (e_1, \Gamma, \Sigma, (\mathbf{let}^n (x \square) e_2) :: k) \quad (1)$$

Applying the induction hypothesis to (\*), there must exist  $\tau_1$  and  $\Sigma_1$  such that

$$(e_1, \Gamma, \Sigma, (\mathbf{let}^n (x \square) e_2) :: k) \mapsto_{pm}^* (\tau_1, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, \widehat{\Sigma}_1 k) \quad (2)$$

$$\text{where } \mathcal{E}_r(\widehat{\Sigma}_1) = \widetilde{\sigma}_1 \text{ and } \mathcal{E}_r(\tau_1) = \widetilde{\tau}_1 \quad (\star)$$

By equivalence of generalization criteria, lem. 11, and the definition of  $\mathcal{P}_n$  and  $\mathcal{P}_{\widetilde{\sigma}_1 \widetilde{\Gamma}}$ ,  $\mathcal{P}_n(\tau_1) = \mathcal{P}_{\mathcal{E}_r(\widehat{\Sigma}_1 \Gamma)}(\widetilde{\tau}_1)$ . By lem. 15,  $\mathcal{E}_r(\widehat{\Sigma}_1 \Gamma) = \widetilde{\sigma}_1 \widetilde{\Gamma}$ . Thus,  $\mathcal{P}_n(\tau_1) = \mathcal{P}_{\widetilde{\sigma}_1 \widetilde{\Gamma}}(\widetilde{\tau}_1)$ .

By lem. 16,

$$\mathcal{R}(\widetilde{\sigma}_1 \widetilde{\Gamma}[x : \mathcal{P}_{\widetilde{\sigma}_1 \widetilde{\Gamma}}(\widetilde{\tau}_1)]) = \widehat{\Sigma}_1 \Gamma[x : \mathcal{P}_n(\tau_1)] \quad (***)$$

By [pm-let-def],

$$(\tau_1, \widehat{\Sigma}_1 \Gamma, \Sigma_1 \Sigma, \widehat{\Sigma}_1 k) \mapsto_{pm} (e_2, \widehat{\Sigma}_1 \Gamma[x : \mathcal{P}_n(\tau_1)], \Sigma_1 \Sigma, (\mathbf{let}^n \square) :: \widehat{\Sigma}_1 k) \quad (3)$$

Because of (\*\*\*), the induction hypothesis applies to (\*\*). Applying induction hypothesis to (\*\*), there must exist  $\tau$  and  $\Sigma_2$  such that

$$\begin{aligned} & (e_2, \widehat{\Sigma}_1 \Gamma[x : \mathcal{P}_n(\tau_1)], \Sigma_1 \Sigma, (\mathbf{let}^n \square) :: \widehat{\Sigma}_1 k) \\ & \mapsto_{pm}^* (\tau, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma[x : \mathcal{P}_n(\tau_1)], \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 ((\mathbf{let}^n \square) :: \widehat{\Sigma}_1 k)) \end{aligned} \quad (4)$$

$$\text{where } \mathcal{E}_r(\widehat{\Sigma}_2) = \widetilde{\sigma}_2 \text{ and } \mathcal{E}_r(\tau) = \widetilde{\tau} \quad (\dagger)$$

By [pm-let-body],

$$\begin{aligned} & (\tau, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma[x : \mathcal{P}_n(\tau_1)], \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2((\mathbf{let}^n \square) :: \widehat{\Sigma}_1 k)) \\ & \mapsto_{pm} (\tau, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k) \end{aligned} \quad (5)$$

By combining (1) - (5), I get that

$$((\mathbf{let}^n (x e_1) e_2), \Gamma, \Sigma, k) \mapsto_{pm}^* (\tau, \widehat{\Sigma}_2 \widehat{\Sigma}_1 \Gamma, \Sigma_2 \Sigma_1 \Sigma, \widehat{\Sigma}_2 \widehat{\Sigma}_1 k)$$

By  $(\dagger)$ ,  $\mathcal{E}_r(\tau) = \widetilde{\tau}$ . By lem. 15,  $(\star)$ , and  $(\dagger)$ ,  $\mathcal{E}_r(\widehat{\Sigma}_2 \widehat{\Sigma}_1) = \widetilde{\sigma}_2 \widetilde{\sigma}_1$ . By  $(\diamond)$ ,  $\widetilde{\sigma}_2 \widetilde{\sigma}_1 = \widetilde{\sigma}$ .

□

**Lemma 18 (Machine and Rewrite System Equivalence (Danvy)).**

$e \mapsto_p^* \tau_p$  iff  $(e, \Gamma, \Sigma, \bullet) \mapsto_{pm}^* (\tau_p, \Sigma' \Gamma, \Sigma' \Sigma, \bullet)$

*Proof.* See Danvy's paper [10]. Danvy's refocusing machine does not have a  $\Sigma$  register, but  $\Sigma$  is only an accumulator of substitutions and thus does not affect the operation of the rest of the machine. □

**Theorem 1 (Soundness/Completeness of Hindley-Milner Reduction).**

For all closed, well-annotated ULCL-A expressions  $e$ ,  $e \mapsto_p^* \tau \Leftrightarrow \mathcal{W}(\bullet, \mathcal{E}_a(e)) = (\theta, \tau)$ .

*Proof.* This result follows from lemma 12 (for soundness  $\Rightarrow$ ) which strengthens the induction to work with any type environment, lem. 17 (for completeness  $\Leftarrow$ ), and lem. 18. □

## APPENDIX D

### DEPTH RANKING SYSTEM

The Hindley-Milner rewrite system assumes that the expression to be typechecked is pre-ranked with  $\lambda$ -depths. I define  $\lambda$ -depth of a  $\lambda$  abstraction intuitively as the number of enclosing  $\lambda$ 's on the path to that  $\lambda$  abstraction, inclusive of that  $\lambda$  abstraction. The following functions in def. 1 annotate a ULCL expression with appropriate depth ranks. The helper function  $d_\lambda$  annotates a ULCL expression given a current depth rank  $n$ , i.e. all  $\lambda$ s will be given annotations  $\geq n + 1$ . The ranking function  $\mathcal{A}$  ranks a ULCL expression using  $d_\lambda$ .  $\mathcal{A}$  is defined so as to give the shallowest  $\lambda$ s rank 1 and any **lets** that are not enclosed by  $\lambda$ s a rank of 0 so that all  $\lambda$ -bound type variables would be considered generalizable.

**Definition 1 (Depth Ranking and Helper Functions).**

$$\begin{aligned}
 d_\lambda &: ULCL \times depth \rightarrow ULCL - A \\
 d_\lambda(x, n) &= x \\
 d_\lambda((@ e e'), n) &= (@ d_\lambda(e, n) d_\lambda(e', n)) \\
 d_\lambda((\lambda(x) e), n) &= (\lambda^{n+1}(x) d_\lambda(e, n+1)) \\
 d_\lambda((\mathbf{let} (x e) e'), n) &= (\mathbf{let}^n (x d_\lambda(e, n)) d_\lambda(e', n)) \\
 \mathcal{A} &: ULCL \rightarrow ULCL - A \\
 \mathcal{A}(e) &= d_\lambda(e, 0)
 \end{aligned}$$

I use the  $\dot{e}$  notation to indicate that the given ULCL expression is annotated.

**Lemma 1 ( $d_\lambda$  Ranking Correctness).** *For all  $(\lambda^m(x) \dot{e}')$  and  $(\mathbf{let}^m(x) \dot{e}')$  in  $d_\lambda(e, n) = \dot{e}$ , there are  $m - n$   $\lambda$ s enclosing  $(\lambda^m(x) \dot{e}')$  inclusive in  $\dot{e}$ .*

*Proof.* **Case**  $e = x$ : There are no  $\lambda$ s (nor **lets**) in  $d_\lambda(x, n) = x$ , so this case vacuously holds.

**Case**  $e = (@ e_1 e_2)$ :  $d_\lambda((@ e_1 e_2), n) = (@ d_\lambda(e_1, n) d_\lambda(e_2, n))$  by definition of  $d_\lambda$  (def. 1).

Any  $(\lambda^m(x) e')$  in  $(e)$  must be either in  $d_\lambda(e_1, n)$  or  $d_\lambda(e_2, n)$ . By induction, there are  $m - n$   $\lambda$ s enclosing  $(\lambda^m(x) e')$  inclusive. Since application is not a  $\lambda$ , there are also exactly  $m - n$   $\lambda$ s enclosing  $(\lambda^m(x) e')$  inclusive in  $e$ . Same reasoning holds for **lets**.

**Case**  $e = (\lambda(y) e_1)$ :  $d_\lambda((\lambda(y) e_1), n) = (\lambda^{n+1}(y) d_\lambda(e_1, n+1))$ . Either  $(\lambda^m(x) e') = (\lambda^{n+1}(y) d_\lambda(e_1, n+1))$  or  $(\lambda^m(x) e')$  is in  $d_\lambda(e_1, n+1)$ .

**Case**  $(\lambda^m(x) e') = (\lambda^{n+1}(y) d_\lambda(e_1, n+1))$ :  $m - n = n + 1 - n = 1$ . This  $\lambda$  is the shallowest in  $e$ , hence there is exactly 1 enclosing  $\lambda$  inclusive, namely itself.

**Case**  $(\lambda^m(x) e') = d_\lambda(e_1, n+1)$ : By induction, there are  $m - (n+1)$   $\lambda$ s enclosing  $\lambda^m(x)$  inclusive in  $d_\lambda(e_1, n+1)$ . In  $e$ ,  $\lambda^{n+1}(y)$  is one additional  $\lambda$  enclosing  $\lambda^m(x)$ , hence there is a total of  $m - (n+1) + 1 = m - n$   $\lambda$ s enclosing  $\lambda^m(x)$  inclusive.

The latter subcase applies to **lets**.

**Case**  $e = (\mathbf{let} (y e_1) e_2)$ :  $d_\lambda((\mathbf{let} (y e_1) e_2), n) = (\mathbf{let}^n (y d_\lambda(e_1, n)) d_\lambda(e_2, n))$  by definition of  $d_\lambda$ .  $(\lambda^m(x) e')$  must be in either  $d_\lambda(e_1, n)$  or  $d_\lambda(e_2, n)$ . By induction, there are  $m - n$   $\lambda$ s enclosing  $\lambda^m(x)$  in  $d_\lambda(e_1, n)$  or  $d_\lambda(e_2, n)$  respectively. The **let** is not a  $\lambda$ , hence there are  $m - n$   $\lambda$ s enclosing  $\lambda^m(x)$  in  $e$ .

The same reasoning applies if  $\mathbf{let}^m (x e'')$  is in either  $d_\lambda(e_1, n)$  or  $d_\lambda(e_2, n)$ . If  $\mathbf{let}^m (x \dots) \dots = \mathbf{let}^n (y d_\lambda(e_1, n)) d_\lambda(e_2, n)$ , then  $n = m$ . Certainly  $m - n = m - m = 0$   $\lambda$ s enclose this shallowest **let** in  $e$ .

□

**Theorem 1 (Ranking Correctness).** *For any ULCL expression  $e$ , let  $\dot{e} = \mathcal{A}(e)$ . For any  $M = (\lambda^m(x) e')$  or  $M = (\mathbf{let}^m(x) e')$  in  $\dot{e}$ ,  $n =$  the number of  $\lambda$ s enclosing  $M$  (inclusive).*

*Proof.* By definition of  $\mathcal{A}$ ,  $\mathcal{A}(e) = d_\lambda(e, 0)$ . By lem. 1,  $m - n = m$   $\lambda$ s inclusive enclose  $M$  in  $\dot{e}$ . □