Efficient ML Type Inference Using Ranked Type Variables

George Kuan David MacQueen
University of Chicago
{gkuan,dbm}@cs.uchicago.edu

Abstract

The Damas-Milner type inference algorithm (commonly known as algorithm $\mathcal{W}$) is at the core of all ML type checkers. Although the algorithm is known to have poor worst-case behavior [8], in practice well-engineered type checkers will run in approximately linear time. To achieve this efficiency, implementations need to improve on algorithm $\mathcal{W}$'s method of scanning the complete type environment to determine whether a type variable can be generalized at a let binding. Following a suggestion of Damas, most ML type checkers use an alternative method based on ranking unification variables to track their position in the type environment.

Here we formalize two such ranking systems, one based on lambda depth (used in the SML/NJ compiler), and the other based on let depth (used in OCaml, for instance). Each of these systems is formalized both with and without the value restriction, and they are proved correct relative to the classic algorithm $\mathcal{W}$. Our formalizations of the various algorithms use simple abstract machines that are similar to those derived from small-step evaluation semantics.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

General Terms Languages

Keywords Type Inference, Type Checking, Algorithm W, Standard ML, Functional Programming, Compilers

1. Introduction

ML compilers have adapted Milner's Algorithm $\mathcal{W}$ [**?**] to type-check realistically rich languages with reasonable efficiency. But to achieve efficiency and the ability to scale to very large programs, implementations of ML type inference must overcome various engineering challenges. The challenge we focus on here is the basic problem of determining which type variables can be generalized to create polymorphic types at let bindings. In algorithm $\mathcal{W}$, the criterion for generalization is that the type variable does not appear free in the "context", represented by a type environment. Since type environments can grow quite large, both in length and the sizes of types they contain, this test can be expensive (quadratic in the size of the program). More efficient algorithms have been developed that use ranking of type variables and a simple rank bound to test for generalizability. The rank associated with a type variable indicates its "position" in the type environment, where the type environment is viewed as ordered from outer to inner bindings.

ML'07, October 5, 2007, Freiburg, Germany.

	Pure	Ref
Classic $\mathcal{W}$	$\mathcal{W}$ (Fig. 4)	$\mathcal{W}_{\mathsf{V}}$ (Fig. 17)
λ-Ranking	$\mathcal{W}_{\lambda}$ (Fig. 7)	$\mathcal{W}_{\lambda\mathsf{V}}$ (Fig. 19)
let-Ranking	$\mathcal{W}_{\mathsf{L}}$ (Fig. 14)	$\mathcal{W}_{\mathsf{LV}}$ (Fig. 20)

Figure 1. Summary of Type Inference Abstract Machines

There are two main variants of this variable ranking approach. The one originally suggested by Damas [1] is based on depth of lambda bindings, and is used in SML/NJ. An equivalent alternative is based on depth of let bindings [13]. Although these implementation techniques have been used in compilers for a couple of decades, their formalization and proof of correctness has been largely neglected. Our goal is to begin to address that gap by proving that these ranked variable variants are equivalent to the original algorithm $\mathcal{W}$. We start by considering type checkers for a pure form of microML (which we will call Pure ML), and then go on to introduce side-effects (ref) and the value restriction [18], calling this language variant Impure ML.

Our approach is to formalize the various type inference algorithms that we will be studying using simple abstract machines such as those derived from small-step operational semantics for evaluation [2]. We then prove equivalences between these abstract machines, whose operation differs mainly at the point where polymorphic generalization occurs for let bindings. Since we have three variants of the type inference algorithm $\mathcal{W}$ (classic, lambda-ranked, and let-ranked) for two variants of the language (Pure and Impure ML), we have a total of six abstract machines, as shown in Fig. 1.

2. An Abstract Machine for Classic Algorithm $\mathcal{W}$

The abstract machines that we use here are simplified versions of the abstract machines used in an earlier paper [4], where we were viewing type checking and inference as a term-rewriting system that incrementally transformed expressions into their types. The abstract machines described below express type inference as a process of nonstandard evaluation of expressions, yielding their types.

The language we will start with is the usual minimal ML, presented in Fig. 3. This language can easily be extended with constant expressions (*e.g.*, numbers, booleans), and with additional basic type constructors (*e.g.*, Int, Bool, List), and we will feel free to do so in examples. Later on we will introduce references, which will force us to complicate type inference by introducing the value restriction.

Types ($\tau \in \mathsf{TYPE}$) are constructed over two forms of type variable. Polymorphic bound type variables (α, β, γ) will occur only in the bodies of polytypes (PTYPE) where they appear in the quantifier prefix – in other words, the types that occur during

$$
\begin{array}{lll}
\mathcal{W}(\Gamma, x) & = & (id, \mathcal{I}(\Gamma(x))) \\
\mathcal{W}(\Gamma, \lambda x.e) & = \text{let} & (\theta, \tau) = \mathcal{W}(\Gamma[x : \xi], e) \qquad \xi \text{ fresh} \\
& \text{in} & (\theta, \theta\xi \to \tau) \\
\mathcal{W}(\Gamma, e\ e') & = \text{let} & (\theta, \tau) = \mathcal{W}(\Gamma, e) \\
& & (\theta', \tau') = \mathcal{W}(\theta\Gamma, e') \\
& & \theta'' = \mathcal{U}(\theta'\tau, \tau' \to \xi) \qquad \xi \text{ fresh} \\
& \text{in} & (\theta''\theta'\theta, \theta''\xi) \\
\mathcal{W}(\Gamma, \mathsf{let}\ x = e\ \mathsf{in}\ e') & = \text{let} & (\theta, \tau) = \mathcal{W}(\Gamma, e) \\
& & (\theta', \tau') = \mathcal{W}(\theta\Gamma[x : \sigma], e') \\
& \text{in} & (\theta'\theta, \tau') \qquad \sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_\Gamma(\tau)\}\tau
\end{array}
$$

Figure 2. Algorithm $\mathcal{W}$ from Lee and Yi [5]

type inference will never contain free occurrences of such type variables. The other type variables (ξ, ζ, ϕ, ψ) are called *unification variables*, or *univariables* for short. They should be considered part of the internal machinery of type inference and should generally be eliminated from the final type produced.[1]

It should also be noted that although neither types nor polytypes (PTYPE) will contain free occurrences of polymorphic type variables, types and polytypes can both contain occurrences of univariables (which are necessarily free occurrences). We will use the term *monotype* for types $\tau \in \mathsf{TYPE}$ to distinguish them from polytypes $\sigma \in \mathsf{PTYPE}$, even though monotypes may contain free occurrences of univariables.

For reference, Fig. 2 shows a standard version of algorithm $\mathcal{W}$ for Pure ML, following the modern presentation given in Lee and Yi [5]. This version differs from Milner's original formulation [9] in that there are explicit generalization and generic instantiation steps that respectively introduce and eliminate polytypes.

Note that $\mathcal{W}$ returns only monotypes τ as results, while both types and polytypes occur in type environments. We see that polytypes are created and then immediately entered into the type environment while processing let bindings. Sometimes, when no univariables are generalizable, these let-bound polytypes will be degenerate in the sense that the quantifier prefix is empty.

New univariables are introduced at three points: (1) when entering a lambda abstraction, where we introduce a new univariable to represent the unknown type of the λ-bound variable, (2) when typing a variable, where we create a *generic instance* of the polytype bound to the variable in the type environment, and (3) in the application case, where we introduce a fresh univariable to represent the return type before unifying. Conversely, univariables can be eliminated (1) by being instantiated (*i.e.* replaced by substitution), or (2) by being generalized at a let binding.

Using standard methods [14], we can transform the classic algorithm of Fig. 2 into the abstract machine in Fig. 4.

The abstract machine states consist of three components, or *registers*. The control, the first register, can be either the next (sub)expression we are type checking or a monotype produced by completing the typing of a subexpression. The type environment, the second register, maps program variables to either monotypes (for λ bindings) or polytypes (for let bindings). The type environment context is an ordered sequence of λ and let-bindings that map expression variables to their types, with the outermost bindings first, so that the λ and let-nesting is reflected in its binding order. The distinction between let and lambda bindings is indicated

[1] A top level expression like $\lambda x.x$ will be assigned a type $\xi \to \xi$ containing a free univariable. We can eliminate this univariable by supplying an implicit top level let declaration context for the expression, typing it as $\mathsf{let}\ it = \lambda x.x$ where the scope is the "rest of the program". Then the univariable can be eliminated by generalization. If the value restriction applies this will not always work (*e.g.*, ref nil), and we may have to eliminate the univariables by dummy instantiation with arbitrary ground types, as is done in the SML/NJ type checker.

$$
\begin{array}{llll}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \mathsf{EXP} \\
\tau & ::= & \alpha \mid \xi \mid \tau \to \tau & \mathsf{TYPE} \\
\sigma & ::= & \forall\overline{\alpha}.\tau & \mathsf{PTYPE}
\end{array}
$$

Figure 3. Pure ML mini-language

by whether the type in the binding is a polytype (possibly degenerate) introduced by a let, or a monotype, introduced by a lambda.

The stack, the final register, represents a typing continuation that keeps track of what still needs to be done to complete the type checking. λ and let stack frames tell the machine when to pop off associated λ- and let-bindings from the type environment (*i.e.*, when the type checker is leaving the scope of the λ or let body). $(\Box\ e)$ and $(\tau\ \Box)$ frames save relevant context information when type checking the operator and operands of applications. Finally, let x in e frames tell the machine to generalize the control type, which must be the type of the definien, and to add the resulting polytype to the type environment when type checking the body e.

As in the original algorithm $\mathcal{W}$, polytypes are only introduced by [let-body], which puts them in the environment. The [var] rule looks up a variable in the type environment and immediately applies $\mathcal{I}$ to generically instantiate the result, eliminating the polytype and (possibly) introducing fresh univariables. Consequently, polytypes only exist in the typing context and are never found in the control.

The [λ-in] rule pushes a new λ-binding onto the type environment, with a fresh univariable as the type of the bound variable. Simultaneously, a λ frame is pushed onto the stack as a reminder to pop off the associated λ binding when we have finished type checking the body. The [λ-out] rule pops off the λ-binding and constructs the function type to complete the typing of an abstraction. The [let-out] rule plays an analogous role for let expressions, popping the let-binding from the typing context and returning the body type. The remaining rules essentially serve as search rules.

The [app-out] rule sets up the function-argument agreement constraint, using a fresh univariable, and solves it by unification ($\mathcal{U}$ defined in Fig. 5). In the process, the unification may eliminate some univariables. Unification fails (in a stuck state) if the function and argument cannot be made to agree, in which case a type error is signaled.

It is straightforward to show that the Algorithm $\mathcal{W}$ abstract machine of Fig. 4 is equivalent to the classical algorithm of Fig. 2, in the sense that for any closed expression e,

$$\mathcal{W}(e, \emptyset) = (\tau, \theta) \quad \Leftrightarrow \quad (e, \emptyset, \bullet) \mapsto (\tau, \emptyset, \bullet) \tag{1}$$

and we therefore take the abstract machine as our reference implementation from here on.

3. Depth Ranking Techniques

Let us make some some key observations about the effect of substitutions on the occurrence of univariables in type environments, which is the critical factor in deciding whether a univariable can be generalized.

First note that for any $\xi \in \mathsf{ftv}(\Gamma)$ we can define the the *rank* of ξ relative to the environment Γ as the index (counting from 1) of the first (*i.e.*, outermost) binding in Γ containing an occurrence of ξ. We will denote this rank by $\mathcal{R}(\xi, \Gamma)$. If $\xi \notin \mathsf{ftv}(\Gamma)$, we define $\mathcal{R}(\xi, \Gamma) = \infty$.

So at a let binding, a univariable is generalizable if and only if its rank relative to the current type environment is infinite:

$$\xi \in \mathcal{G}_\Gamma(\tau) \ \Leftrightarrow\ \xi \in \mathsf{ftv}(\tau) \text{ and } \mathcal{R}(\xi, \Gamma) = \infty$$

Another useful observation is that at a let binding, if we generalize relative to the current environment Γ to get a polytype σ and add the polytype binding $[x : \sigma]$ to Γ, then any univariable $\xi \in \mathsf{ftv}(\sigma)$

$$
\begin{array}{rcll}
\Gamma & ::= & \emptyset \mid \Gamma[x:\tau] \mid \Gamma[x:\sigma] & \text{Tyenv} \\
f & ::= & \lambda \mid (\Box\ e) \mid (\tau\ \Box) \mid \mathsf{let}\ x\ \mathsf{in}\ e \mid \mathsf{let} & \text{Frames} \\
k & ::= & \bullet \mid f :: k & \text{Stack} \\
c & ::= & e \mid \tau & \text{Control} \\
s & ::= & (c, \Gamma, k) & \text{States}
\end{array}
$$

Initial States $s_0 = (e, \emptyset, \bullet)$

$$
\begin{array}{rcll}
(x, \Gamma, k) & \mapsto & (\mathcal{I}(\Gamma(x)), \Gamma, k) & [\mathsf{var}] \\
(\lambda x.e, \Gamma, k) & \mapsto & (e, \Gamma[x:\xi], \lambda :: k) & [\lambda\text{-}\mathsf{in}] \\
 & & \xi \text{ fresh} & \\
(\tau, \Gamma[x:\tau'], \lambda :: k) & \mapsto & (\tau' \to \tau, \Gamma, k) & [\lambda\text{-}\mathsf{out}] \\
(e_1\ e_2, \Gamma, k) & \mapsto & (e_1, \Gamma, (\Box\ e_2) :: k) & [\mathsf{app\text{-}l}] \\
(\tau, \Gamma, (\Box\ e_2) :: k) & \mapsto & (e_2, \Gamma, (\tau\ \Box) :: k) & [\mathsf{app\text{-}r}] \\
(\tau, \Gamma, (\tau'\ \Box) :: k) & \mapsto & (\theta\xi, \theta\Gamma, \theta k) & [\mathsf{app\text{-}out}] \\
 & & \text{where } \theta = \mathcal{U}(\tau', \tau \to \xi), \xi \text{ fresh} & \\
(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \Gamma, k) & \mapsto & (e_1, \Gamma, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) & [\mathsf{let\text{-}def}] \\
(\tau, \Gamma, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) & \mapsto & (e_2, \Gamma[x:\sigma], \mathsf{let} :: k) & [\mathsf{let\text{-}body}] \\
 & & \text{where } \sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_\Gamma(\tau)\}\tau, \overline{\alpha} \text{ fresh} & \\
(\tau, \Gamma[x:\sigma], \mathsf{let} :: k) & \mapsto & (\tau, \Gamma, k) & [\mathsf{let\text{-}out}]
\end{array}
$$

$\mathcal{G}_\Gamma(\tau) = \mathsf{ftv}(\tau) - \mathsf{ftv}(\Gamma)$ — Generalization

$\mathcal{I}(\forall\overline{\alpha}.\tau) = \{\overline{\xi}/\overline{\alpha}\}\tau$ where $\overline{\xi}$ is fresh
$\mathcal{I}(\tau) = \tau$ — Generic Instantiation

Figure 4. Algorithm $\mathcal{W}$ Abstract Machine

$$
\begin{array}{rcl}
\mathcal{U}(\tau, \tau) = \varepsilon \quad \mathcal{U}(\tau, \xi) = \mathcal{U}(\xi, \tau) & & \\
\text{where } \tau \text{ is not a univariable} & & \\
\mathcal{U}(\xi, \tau) = \{\tau/\xi\} \text{ if } \xi \notin \mathsf{ftv}(\tau) & & \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \mathcal{U}(\theta(\tau_2), \theta(\tau_4)) \circ \theta & & \\
\text{where } \theta = \mathcal{U}(\tau_1, \tau_3) & &
\end{array}
$$

Figure 5. Unification

already appeared in Γ, *i.e.*, $\xi \in \mathsf{ftv}(\Gamma)$. This is because if ξ did not appear in Γ, it would have been generalized and replaced by a polymorphic bound type variable. Thus, at the point where it is added, a let binding introduces no new univariables into the environment. It also means that the first occurrence of a univariable in the environment is always at a λ binding. These properties are obviously preserved if a type substitution is applied to a type environment. So we have the invariant (for Pure ML):

1. *The first occurrence of a univariable in an environment is always at a λ binding.*

Now let us consider the effect of a substitution on the rank of a univariable in an environment. Consider a univariable ξ, an environment Γ, and a type substitution θ. What is the relation between $\mathcal{R}(\xi, \Gamma)$ and $\mathcal{R}(\xi, \theta\Gamma)$? If $\xi \in \mathsf{dom}(\theta)$, then ξ is eliminated by θ (remember that all our substitutions are idempotent because of the occurrence check), so $\mathcal{R}(\xi, \theta\Gamma) = \infty$. But in this case, since θ is applied uniformly to all components of the abstract machine, ξ is eliminated everywhere, and its rank is no longer relevant since we will never ask whether ξ can be generalized.

Next assume that $\xi \notin \mathsf{dom}(\theta)$. Then applying θ to Γ will not remove any occurrences of ξ, so the rank of ξ in Γ will not increase, but it could decrease if ξ occurs in the range of θ and new occurrences of ξ are added in $\theta(\Gamma)$ (including the case where $\xi \notin \mathsf{ftv}(\Gamma)$ but $\xi \in \mathsf{ftv}(\theta(\Gamma))$. Hence

$$\mathcal{R}(\xi, \theta(\Gamma)) \leq \mathcal{R}(\xi, \Gamma) \text{ if } \xi \notin \mathsf{dom}(\theta)$$

Thus, except for the case where a univariable is eliminated by a substitution, a substitution can only decrease the rank of a univariable or leave it the same. This means that a substitution may "promote" a univariable to an earlier binding and thus make it "less generalizable".

As we have defined it, computing the rank of a univariable relative to an environment is no less expensive than computing whether it occurs in the environment. But the main idea of the ranked variable algorithms we will present below is that we can avoid scanning the environment by making the rank an attribute of the univariable and managing that attribute incrementally so that it is always equal to the intrinsic rank defined above, relative to the current environment in the machine state.

Since in the $\mathcal{W}$ machine for Pure ML, only the λ bindings *count*, our first version of univariable ranking will assign ranks based on the depth of λ nesting (*i.e.* counting only λ bindings). We will then look at a variant for Pure ML that counts only let definien marks (which demarcate the scopes of nested let definiens). When we introduce references and the consequent requirement for the value restriction, things become more complicated, because the invariant that univariables first occur in λ bindings will no longer be the case, and non-value let bindings will act like λ bindings.

4. λ-Depth Ranking

In Figs. 6 and 7, we modify the Pure ML inference machine by adding ranks to the univariables. The rank associated with a univariable is either a natural number or ∞, and if a univariable occurs in the current type environment its rank is meant to correspond to the index of the earliest λ-binding in the environment in which it occurs. A rank greater than the number of λ bindings in the current environment will indicate that the univariable is generalizable, *i.e.* its intrinsic rank is ∞.

For convenience, we add a new register to the machine states containing the current λ nesting depth d. This register starts at 0 and counts how many λ-abstractions we have passed through to reach to the current control. Its value will always be equal to both the number of λ-bindings in the current environment and the number of λ frames in the context stack.

The $\mathcal{W}$ machine rules are easily adapted to incorporate λ-depth ranking, and we will call this new machine $\mathcal{W}_\lambda$. The new [λ-in] rule increments the depth, introduces a fresh univariable whose initial rank is the new λ nesting depth, and pushes a new λ binding onto the environment. The [λ-out] rule now also decrements the depth.

We can categorize bindings in type environments according to whether they are introduced by the [let-body] rule, in which case the variable is bound to a polytype σ, or by the [λ-in] rule, in which case the variable is bound to a monotype τ (and this distinction is preserved under substitution). This distinction on bindings is expressed in the definition of environments in Fig. 7.

As noted in the previous section, if a univariable occurs in an environment, the first binding it occurs in will always be a λ-binding, and let-bindings play a secondary role. So we can modify the notion of the occurrence index of a univariable ξ in an environment Γ, which we called the intrinsic rank $\mathcal{R}(\xi, \Gamma)$, by counting only λ-bindings from left to right when we calculate this index. The univariable ranks in the $\mathcal{W}_\lambda$ machine are assigned and managed so that they always correspond to the intrinsic rank determined by the location of the univariable in the current environment (if any).

Note that because applying unification substitutions in [app-out] can "promote" univariables to earlier λ-bindings, the first λ-binding containing a univariable may not correspond to the program point where it was introduced. A substitution that promotes a univariable to an earlier binding must compensate for the change in the intrinsic rank of the univariable by adjusting the explicit rank of the univariable to reflect its new earliest occurrence in the environment.

The $\mathcal{L}$ limit substitution defined in Fig. 8 performs this necessary adjustment to univariable ranks. When a unification substitu-

$$
\begin{array}{rcll}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \text{EXP} \\
\tau & ::= & \xi^d \mid \tau \to \tau \mid \alpha & \text{TYPE} \\
d & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \infty & \text{Ranks} \\
\sigma & ::= & \forall \overline{\alpha}.\tau & \text{PTYPE}
\end{array}
$$

Figure 6. λ-Ranking Language

$$
\begin{array}{rcll}
\Gamma & ::= & \emptyset \mid \Gamma[x:\tau] \mid \Gamma[x:\sigma] & \text{Tyenv} \\
f & ::= & \lambda \mid (\square\ e) \mid (\tau\ \square) \mid \mathsf{let}\ x\ \mathsf{in}\ e \mid \mathsf{let} & \text{Frames} \\
k & ::= & \bullet \mid f :: k & \text{Stack} \\
c & ::= & e \mid \tau & \text{Control} \\
s & ::= & (c, \Gamma, d, k) & \text{States}
\end{array}
$$

Initial State $s_0 = (e, \emptyset, 0, \bullet)$

$$
\begin{array}{rcll}
(x, \Gamma, d, k) & \mapsto_\lambda & (\mathcal{I}(\Gamma(x)), \Gamma, d, k) & [\mathsf{var}] \\
(\lambda x.e, \Gamma, d, k) & \mapsto_\lambda & (e, \Gamma[x : \xi^{d+1}], d+1 & [\lambda\text{-}\mathsf{in}] \\
 & & \quad \lambda :: k) & \\
(\tau, \Gamma[x:\tau'], d, \lambda :: k) & \mapsto_\lambda & (\tau' \to \tau, \Gamma, d-1, k) & [\lambda\text{-}\mathsf{out}] \\
(e_1\ e_2, \Gamma, d, k) & \mapsto_\lambda & (e_1, \Gamma, d, (\square\ e_2) :: k) & [\mathsf{app\text{-}l}] \\
(\tau, \Gamma, d, (\square\ e_2) :: k) & \mapsto_\lambda & (e_2, \Gamma, d, (\tau\ \square) :: k) & [\mathsf{app\text{-}r}] \\
(\tau, \Gamma, d, (\tau'\ \square) :: k) & \mapsto_\lambda & (\theta\xi^\infty, \theta\Gamma, d, \theta k) & [\mathsf{app\text{-}out}] \\
 & & \quad \theta = \mathcal{U}(\tau', \tau \to \xi^\infty) & \\
(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \Gamma, d, k) & \mapsto_\lambda & (e_1, \Gamma, d, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) & [\mathsf{let\text{-}def}] \\
(\tau, \Gamma, d, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) & \mapsto_\lambda & (e_2, \Gamma[x:\sigma], d, \mathsf{let} :: k) & [\mathsf{let\text{-}body}] \\
 & & \quad \sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_d(\tau)\}\tau \text{ where } \overline{\alpha} \text{ is fresh} & \\
(\tau, \Gamma[x:\sigma], d, \mathsf{let} :: k) & \mapsto_\lambda & (\tau, \Gamma, d, k) & [\mathsf{let\text{-}out}]
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{G}_d(\tau) & = & \{\xi^m \in \mathsf{ftv}(\tau) \mid m > d\} & \text{Generalizable} \\
\mathcal{I}(\forall\overline{\alpha}.\tau) & = & \{\overline{\xi^\infty}/\overline{\alpha}\}\tau & \text{Generic Instantiation} \\
\mathcal{I}(\tau) & = & \tau &
\end{array}
$$

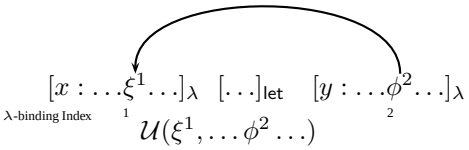
Figure 7. Algorithm $\mathcal{W}$ Machine extended with λ-depth Ranking ($\mathcal{W}_\lambda$)

$$
\begin{array}{rl}
\mathcal{U}(\tau,\tau) = \varepsilon \quad \mathcal{U}(\tau, \xi^m) = \mathcal{U}(\xi^m, \tau) & \\
\qquad \text{where } \tau \text{ is not a univariable} & \\
\mathcal{U}(\xi^m, \tau) = \mathcal{L}_m(\tau) \circ \{\tau/\xi^m\} \text{ if } \xi^m \notin \mathsf{ftv}(\tau) & \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \mathcal{U}(\theta(\tau_2), \theta(\tau_4)) \circ \theta & \\
\qquad \text{where } \theta = \mathcal{U}(\tau_1, \tau_3) & \\
\mathcal{L}_d(\tau) = \{\xi^d/\xi^m \mid \xi^m \in \mathsf{ftv}(\tau) \wedge m > d\} &
\end{array}
$$

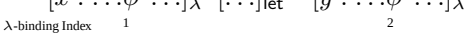
Figure 8. Unification with λ-rank Maintenance

tion maps a univariable ξ to a type τ, univariables occurring in τ may be introduced to λ-bindings to the left of their original first occurrences, if ξ occurs earlier in the environment. Thus, if the first occurrence of ξ was to the left of the first occurrence of some univariable $\phi \in \mathsf{ftv}(\tau)$, then after unification, the first occurrence of ϕ is now where the first occurrence of ξ was before as shown in Fig. 9. If the first occurrence of ξ was to the right of the first occurrence of ϕ, then the first occurrence of ϕ does not move after unification as shown in Fig. 10.

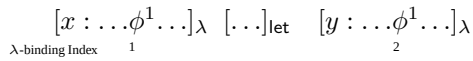
As a simple example of this issue, consider the expression $\lambda x.\mathsf{let}\ y = \lambda z.xz\ \mathsf{in}\ \ldots$. The λ bindings for x and z produce an environment $\Gamma = [x : \xi^1_x][z : \xi^2_z]$ for the typing of the application xz. The typing of this application involves a unification producing the substitution $\theta = \xi^1_x \mapsto (\xi^2_z \to \xi^\infty_r)$. But if θ is applied to the typing environment it would produce $\Gamma' = [x : \xi^2_z \to \xi^\infty_r][z : \xi^2_z]$ in which the ranks of the univariables in the x binding no longer correspond to its index in the environment, which is 1. So an additional rank limiting substitution is required to adjust the ranks of ξ_z and ξ_r downward: $(\xi^2_z \mapsto \xi^1_z, \xi^\infty_r \mapsto \xi^1_r)$. When this is applied we get a properly ranked environment $\Gamma' = [x : \xi^1_z \to \xi^1_r][z : \xi^1_z]$. Now the adjusted rank of ξ^1_z will prevent the type of y, namely $\xi^1_z \to \xi^1_z$ from being generalized at the let binding.



(a) If ξ (a contextual variable) instantiates to y's type, then ϕ (a local variable) propagates into x's λ-binding.

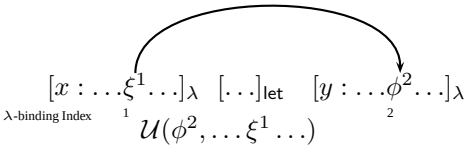


(b) The rank of ϕ must now be limited to that of ξ because the first occurrence of ϕ has shifted left, thus making it contextual.

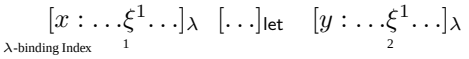


(c) After Applying the Limit Substitution

Figure 9. Limit Substitution ($\mathcal{L}$) Example: Subscripts on bindings indicate the kind of binding.



(a) If ϕ instantiates to x's type, then ξ propagates into y's λ-binding.



(b) ξ's rank is fine because its first occurrence has not shifted. It remains contextual.

Figure 10. Example of when the limit substitution is unnecessary

Definition 1 (Valid Machine States ($\mathcal{VMS}$))
A machine state (c, Γ, d, k) is valid if it is reachable from some valid initial machine state $(e, \emptyset, 0, \bullet)$ (i.e., $(e, \emptyset, 0, \bullet) \mapsto^* (c, \Gamma, d, k)$ where $\mapsto$ is the machine transition relation).

From this point on, a metavariable with a tilde ($\widetilde{\ }$) on top denotes the corresponding instance of that construct where the ranks of any univariables that occur are erased or ignored.

Definition 2
$|\Gamma| =$ *the length of* Γ, *and similarly for* $|\widetilde{\Gamma}|$.

Definition 3
$\mathcal{D}(\Gamma) =$ *# λ bindings in* Γ, *and similarly for* $\mathcal{D}(\widetilde{\Gamma})$.

Definition 4
Γ_λ *denotes the list of λ-bindings in* Γ, *and similarly for* $\widetilde{\Gamma}_\lambda$.

Definition 5 (Intrinsic Univariable Rank)

$$\begin{array}{rcl}\widetilde{\mathcal{R}}(\xi,\emptyset) & = & \infty \\ \widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}[x:\widetilde{\tau}]) & = & \min(\widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}),\mathcal{D}(\widetilde{\Gamma})+1) \textit{ if } \xi \in \mathsf{ftv}(\widetilde{\tau}) \\ & = & \infty \textit{ otherwise} \\ \widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}[x:\widetilde{\sigma}]) & = & \widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma})\end{array}$$

This computes the index of the first λ-binding containing an occurrence of a univariable, ignoring let-bindings. If a univariable is not in the environment, it is assigned the intrinsic rank ∞, which ensures that such a univariable should be generalized because it is always greater than any let expression's finite λ-depth.

Definition 6
$\mathcal{R}(\xi^m,\Gamma) = \widetilde{\mathcal{R}}(\xi,\mathcal{E}_r(\Gamma))$
where $\mathcal{E}_r$ erases the ranks of univariables in Γ.

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

Definition 7
$\widetilde{\Gamma}'$ is an **extension** *of $\widetilde{\Gamma}$ if and only if there exists $\widetilde{\Gamma}''$ such that $\widetilde{\Gamma}'$ is the concatenation of $\widetilde{\Gamma}$ and $\widetilde{\Gamma}''$ (in that order).*

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 1 ($\widetilde{\mathcal{R}}$ Invariant Under $\widetilde{\Gamma}$ Extension)
If $\xi \in \mathsf{ftv}(\widetilde{\Gamma})$ and $\widetilde{\Gamma}'$ is an extension of $\widetilde{\Gamma}$, then $\widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}') = \widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma})$.

Proof: By induction on the length of $\widetilde{\Gamma}$ ■

Because $\widetilde{\mathcal{R}}$ computes the first occurrence index for a univariable in an environment, if a univariable does occur in the environment $\widetilde{\mathcal{R}}$ will return the same index for any extension of that environment.

By definition, $\widetilde{\mathcal{R}}$ counts the number of λ-bindings up to a univariable's first occurrence. Naturally, this number cannot exceed the total number of λ-bindings on $\widetilde{\Gamma}$. We observe this fact in Lemma 2.

Lemma 2
If $\xi \in \mathsf{ftv}(\widetilde{\Gamma}_\lambda)$, then $\widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}) \leq \mathcal{D}(\widetilde{\Gamma})$.

Proof: By induction on the length of $\widetilde{\Gamma}$ ■

We determined by construction that the rank of a univariable that does not occur in the environment is ∞. This is formally stated in Lemma 3.

Lemma 3
If $\xi \notin \mathsf{ftv}(\widetilde{\Gamma}_\lambda)$ then $\widetilde{\mathcal{R}}(\xi,\widetilde{\Gamma}) = \infty$.

Proof: By induction on the structure of $\widetilde{\Gamma}$ ■

To formalize the notion of first occurrence of a univariable, we appeal to Lemma 4.

Lemma 4
If ξ^m occurs in the n-th λ-binding in Γ (counting from the left) and ξ^m occurs in no λ-binding to the left of that n-th λ-binding, then $\mathcal{R}(\xi^m,\Gamma) = n$.

Proof: By induction on the length of Γ and Lemma 3 ■

We also make a simple observation about substitutions and environments that simplifies the proofs in Lemma 5.

Lemma 5 (Substitution Range Preservation)
If $\tau \in \mathrm{rng}\,\Gamma$, then $\sigma\tau \in \mathrm{rng}\,\sigma\Gamma$.

Proof: This follows from definition of type variable substitution on type environments. ■

Definition 8 (Properties for $\mathcal{W}_\lambda$)
The following are invariant properties on any valid $\mathcal{W}_\lambda$ machine state $s = (c,\Gamma,d,k)$:

Prop$_1$ $\forall \xi^m \in \mathsf{ftv}(\Gamma), \mathcal{R}(\xi^m,\Gamma) = m$
Prop$_2$ $\forall \xi^m \in \mathsf{ftv}(c)$ *such that* $m \leq \mathcal{D}(\Gamma)$, *then* $\xi^m \in \mathsf{ftv}(\Gamma)$
Prop$_3$ $\mathcal{D}(\Gamma) = \#$ *of* λ*s on* k
Prop$_4$ $\forall \xi^m \in \mathsf{ftv}(k)$ *such that* $m \leq r$ *where* r *is the number of* λ *frames below the frame with the occurrence of* ξ^m *in the stack* k, $\xi^m \in \mathsf{ftv}(\Gamma)$

Prop$_1$ asserts that a univariable's rank corresponds to the index of the λ-binding containing the first occurrence of that univariable in the environment. This is a precursor to Corollary 1 below that provides a notion of completeness for the λ-ranking scheme. All univariables in the environment must be given a low rank.

Prop$_2$ says that all the "contextual" univariables occurring in the control are bound in the environment. Contextual refers to having a rank no greater than $\mathcal{D}(\Gamma)$. This specifically excludes univariables that would run off the end of the environment at the current scope because their ranks are too great, including univariables with rank ∞. Prop$_2$ asserts that all low ranked univariables must be in the environment. Therefore it provides a notion of the soundness of the ranking scheme.

Prop$_3$ tells us that the number of λ-bindings in the environment corresponds to the λ-nesting depth. This simple observation is necessary for establishing Prop$_1$ when only λ-depth information is available.

Prop$_4$ is a rather subtle property. It ensures that the a modified version of Prop$_2$ holds for those univariables occurring on the stack in $(\tau\ \square)$ frames. Because the type τ of the operator is in a sense "parallel" to the current operand expression that the machine is working on, it may contain some seemingly low ranked univariables (rank $<$ current λ-depth) that are not in the current environment for the operand. These univariables are not truly low ranked because at the point when $(\tau\ \square)$ was pushed onto the stack, τ was in the control, so by Prop$_2$ any univariables not in the environment at that point had rank higher than the $\mathcal{D}(\Gamma')$, where Γ' was the then-current environment. But $\mathcal{D}(\Gamma')$ was also equal to the number of λ frames on the stack before the $(\tau\ \square)$ frame was pushed. Once the machine pops all the frames sitting on top of the $(\tau\ \square)$ frame, the environment will once again have its original λ-depth.

For example, in $(\lambda x.x)(\lambda y.\lambda z.z)$, before the machine reaches the body of $\lambda z.z$, $\lambda x.x$ is given the type $\xi^1 \to \xi^1$. When typing the body of $\lambda z.z$, the λ-depth is 2. It seems that ξ^1 has a low rank. However, at the point we were just finished with $\lambda x.x$ and when we pop out of the operand, the λ-depth is 0. Therefore ξ^1 is actually a high ranked univariable that we do not expect to find in the environment when we get back to it.

We proceed by first showing that these properties are invariant under the unification that results from [app-out].

Lemma 6 (Invariants under unification)
For all Γ, if the properties from def. 8 hold for $s = (\tau, \Gamma, d, (\tau' \square) :: k) \in \mathcal{VMS}$ and $\theta = \mathcal{U}(\tau', \tau \to \psi^\infty)$ (where ψ^∞ is fresh), then the properties must also hold for $s' = (\theta\psi^\infty, \theta\Gamma, d, \theta k)$.

Proof (sketch): This proof is by induction on sizes of τ' and the other type argument of $\mathcal{U}$. The instantiation case is the interesting part of this proof. In that case, by definition of unification, $\theta = \mathcal{U}(\phi^n, \tau) = \mathcal{L}_n(\tau) \circ \{\tau/\phi^n\}$.

The interesting cases for $\text{Prop}_1(s')$ are illustrated in Figs. 10 and 9. In the first case, assume $\xi^m \in \mathsf{ftv}(\theta\Gamma)$ was not limited, hence it must have a rank that is lower or equal to n. Because ϕ^n was substituted away by θ, it must occur in Γ. By, Lemma 2, $n \leq \mathcal{D}(\Gamma)$. Thus, $m \leq \mathcal{D}(\Gamma)$. By $\text{Prop}_2(s)$, ξ^m must occur in Γ. By $\text{Prop}_1(s)$, $\mathcal{R}(\xi^m, \Gamma) = m$. In the case where ξ^m was limited, the first occurrence of ξ^m shifted left to the former first occurrence of ϕ^n. Thus, it should also receive rank n. Lem. 4 helps establish this.

For $\text{Prop}_2(s')$, the unlimited case is simple. The case where ξ^m was limited is somewhat tricky. If the univariable to be instantiated ϕ^n has a rank $n > \mathcal{D}(\theta\Gamma)$, the case is vacuous, so assume that $m \leq \mathcal{D}(\theta\Gamma)$. Note that by the definition of $\mathcal{L}$, $m = n$. We first have to show that ϕ^n must be in Γ. ϕ^n can come from τ, τ', or ψ^∞. The ψ^∞ case is vacuous because $\infty > \mathcal{D}(\Gamma)$. The τ case is handled by $\text{Prop}_2(s)$. The τ' case takes a little more work because $\text{Prop}_2(s)$ did not say anything about τ'. This is the reason why we need Prop_4. $\text{Prop}_4(s)$ gives us what we need. Assuming that $\phi^n \in \mathsf{ftv}(\Gamma)$, it is straightforward to show that $\xi^m \in \mathsf{ftv}(\Gamma)$ with the help of Lemma 5.

$\text{Prop}_3(s')$ is easily to establish because substitution does not change the length of Γ or k. Prop_4 is also relatively straightforward. ■

The properties trivially hold for the initial states. Now we proceed to show that the properties are indeed invariant under all machine transition rules. This would establish that the properties are invariants over all valid machine states.

Lemma 7 (Properties invariant over all machine transitions)
If the props. in def. 8 hold for $s \in \mathcal{VMS}$ and $s \mapsto_\lambda s'$, then they must hold for s'.

Proof (sketch): We need Prop_3 to push through Prop_1's [λ-in] case, thus showing that the ranks do measure λ nesting depth. This fact combined with Lemma 3 establish that the ranks correspond to the λ-binding index of the first occurrence (actually the only occurrence) of the fresh univariable representing this λ's parameter type in the environment.

For the [λ-out], [let-body], and [let-out] cases, we use Lemma 1 to show Prop_1. For [app-r], $\text{Prop}_4(s')$ simply inherits the property from $\text{Prop}_2(s)$ because τ merely shifted from the control to the top of the stack. Prop_1 and Prop_2 for the [app-out] case follow from Lemma 6.

For Prop_1 in the [let-body] case, we use Prop_2's prop. that we assume holds before transitioning by [let-body] to show that any univariable ξ in $\mathsf{ftv}(\Gamma[x : \sigma])$ must also occur in $\mathsf{ftv}(\Gamma)$. That is to say, first occurrences of univariables in environments are never in let-bindings. We can then conclude by assumption and Lemma 1 that $\mathcal{R}(\xi^m, \Gamma[x : \sigma]) = m$.

For Prop_2, we note that $\xi^m \in \mathsf{ftv}(\tau' \to \tau)$ in [λ-out] cannot have a first occurrence in the last binding in $\Gamma[x : \tau']$ because by assumption, $m \leq \mathcal{D}(\Gamma)$. If this were not the case, then we would be seeing unbound yet non-generic univariables after [λ-out] transitions.

The rest of the cases are fairly straightforward case analyses assuming that the properties holds before each transition rule. ■

From Prop_1, we can easily obtain a completeness corollary.

Corollary 1 (Ranking is Complete)
For all Γ from $s = (c, \Gamma, d, k) \in \mathcal{VMS}$
$\xi^m \in \mathsf{ftv}(\Gamma) \Rightarrow m \leq \mathcal{D}(\Gamma)$

Proof (sketch): By Prop_1 from Lemma 7, $\mathcal{R}(\xi^m, \Gamma) = m$. Since $\mathcal{D}(\Gamma) = |\Gamma_\lambda|$, $m \leq \mathcal{D}(\Gamma)$ by definition of $\mathcal{R}$ and Lemma 2. ■

We can put together Corollary 1 and Prop_2 to establish that the ranking corresponds to searching through the environment for all valid machine states.

Lemma 8
For any $(\tau, \Gamma, d, k) \in \mathcal{VMS}$ and $\forall \xi^m \in \mathsf{ftv}(\Gamma)$,
$m \leq d$ *iff* $\xi^m \in \mathsf{ftv}(\Gamma)$.

Proof:

$\Rightarrow$: This direction follows from Prop_2 of Lemma 7.

$\Leftarrow$: This direction follows from Corollary 1. $m \leq \mathcal{D}(\Gamma) = d$

■

The generalizability criteria are equivalent. Consequently, we always generalize the same set of univariables using the classical and λ-ranking algorithms.

Lemma 9 (Equivalence of Generalizability Criteria)
If $(\tau, \Gamma, d, \mathsf{let}\ x = \square\ \mathsf{in}\ e :: k) \mapsto (e, \Gamma[x : \sigma], d, \mathsf{let} :: k)$ where $\sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_d(\Gamma)\}\tau$, then $\mathcal{G}_d(\Gamma) = \mathcal{G}_\Gamma(\Gamma)$.

Proof: This proof falls out from the definitions of $\mathcal{G}_d$ and $\mathcal{G}_\Gamma$, and Lemma 8. ■

Finally, we put everything together. The classical algorithm $\mathcal{W}$ and the λ-ranking algorithms are equivalent.

Theorem 1 (Equivalence to Classical Algorithm $\mathcal{W}$)
$(e, \emptyset, 0, \bullet) \mapsto^* (\tau, \emptyset, 0, \bullet)$ *iff* $(e, \emptyset, \bullet) \mapsto^* (\tau, \emptyset, \bullet)$.

Proof: We only inspect the depth register in [let-body]. In all other cases, the λ-ranking and classical machines are identical. The depth register is only along for the ride. Because of Lemma 9, the [let-body] behaviors are also identical. ■

5. Let-Depth Ranking

The let-depth ranking scheme is similar to the λ-depth ranking except for a few significant adjustments. We can reuse the language and the same notation for ranks. The main observation that the let-depth ranking relies on is that the locality of univariables at different λ-depths is the same as long as we are in the same let definien's scope. In contrast, λ-ranks distinguish among λ's at different λ-nestings even though they may all be in the same let definien scope. We have to be careful, though, because a let expression only extended the type environment when the type checker reaches the body. At that point, generalization of the definien type would have already occurred. So we cannot use let-bindings to derive a ranking for determining generalization. We have to determine a rank before generalization occurs. So, we turn to another implicit structure of the environment, the partitioning of bindings into contiguous

$\lambda x.\mathsf{let}\ a = \lambda y_1.\lambda y_2.\mathsf{let}\ b = \mathsf{let}\ c = \bullet$
$\mathsf{in}\ \ldots$
$\mathsf{in}\ \ldots$
$\mathsf{in}\ \ldots$

Environment at $\bullet$: $[x : \xi^0][y_1 : \zeta^1][y_2 : \phi^1]$

Figure 11. When we reach $\bullet$, we pass into three let definiens but added no let-bindings to the environment. The type of x is less generalizable than that of the y's. The type of the y's should be generalizable at the same point.

$\mathsf{let}\ a = \lambda x.\mathsf{let}\ b = \ldots$
$\mathsf{in}\ \ \mathsf{let}\ c = \ldots$
$\mathsf{in}\ (\lambda y.y)\bullet$
$\mathsf{in}\ \ldots$

Environment at $\bullet$: $[x : \xi^1][b : \sigma_b][c : \sigma_c][y : \zeta^1]$

Figure 12. When we reach $\bullet$, we are in no let definiens but added two let-bindings to the environment. The types of x and y are both generalizable at the let a.

blocks according to the definien scope. We can extend the machine introduced in $\mathcal{W}_\lambda$ to make this let definien partitioning manifest.

We cannot naively replace λ's with let's in the development from the previous section to get a working let-ranking scheme. In particular, λ's were simple. We only traverse a λ-abstraction in one pass. First, we extend the environment with the bound λ-variable. Then we traverse the body of the λ using this extended environment. Once we are done, we pop out of the λ never to revisit it. In contrast, for let's, we have to first traverse the definien without extending the environment, pop out of the definien, extend the environment with the generalized definien type, and only then traverse the body of the let with this extended environment. We have to increment the depth when we pass into a definien, as for instance in the expression $\lambda x.\mathsf{let}\ y = x\ \mathsf{in}\ \ldots$, where we need to prevent the type of x from being generalized where y is bound. On the other hand, the body should have the depth that held before we entered into the definien. This permits the type of $\lambda y.y$ to generalize in $\mathsf{let}\ x = \mathsf{let}\ \ldots\ \mathsf{in}\ \lambda y.y\ \mathsf{in}\ \ldots$. The complication is that there is no simple connection between the number of let-bindings in the environment and the number of let definiens we have passed into (but have not finished typing). Figs. 11 and 12 give examples of this problem. In the environments for the two examples, we label the univariables with the let definien depth at the point of introduction. Although in Fig. 11 ξ has a lower let definien depth than ζ and ϕ, there is nothing in the environment to demarcate the two groups of bindings. In Fig.12, ξ and ζ have the same depth and yet they are separated by two let-bindings.

To overcome this problem, we rank by the number of let definien letd markers in an environment instead of λ-bindings. We want to identify contiguous blocks of λ-bindings in the environment that are within the same let definien scope. Thus, every time we pass into a let definien, we mark the (rightmost) end of the environment with a letd marker. This marker tells us that any λ to left of the marker will be less generalizable than any λ that will be to the right. We extend the definition of Γ with these letd marks in Fig. 14. The letd marked versions of the environments in Figs. 11 and 12 would be $[x : \xi^0][\mathsf{letd}][y_1 : \zeta^1][y_2 : \phi^1][\mathsf{letd}][\mathsf{letd}]$ and $[\mathsf{letd}][x : \xi^1][b : \sigma_b][c : \sigma_c][y : \zeta^1]$ respectively.

Fig. 13 shows both the λ and let ranking of the following program:

$\lambda s.\mathsf{let}\ x = \lambda t.\ldots$
$\mathsf{in}\ \mathsf{let}\ y = \lambda u.\ldots$
$\mathsf{in}\ \lambda v.\quad \mathsf{let}\ z = \lambda w.\ldots$
$\mathsf{in}\ \ldots$

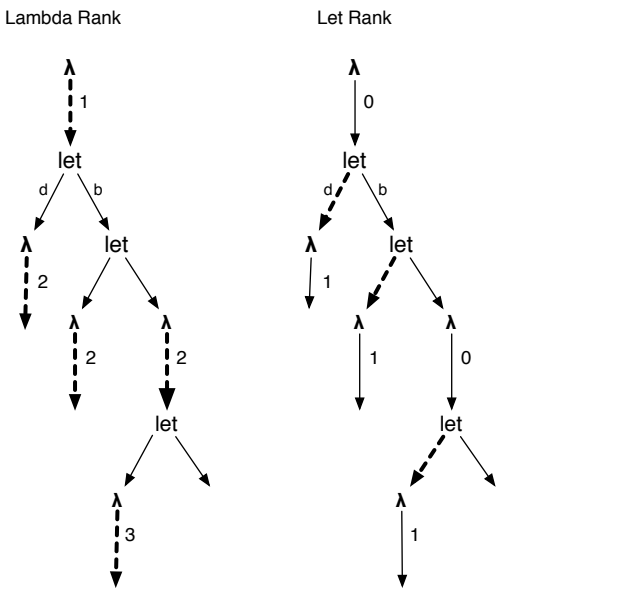


Figure 13. Comparison of λ and let ranks in expression structure

The left branch of each let node represents its definien (d) and the right branch its body (b). The dashed lines indicate where we increment the rank. In this example, there are no let definiens nested in another let definien, hence generalizability at a let definien is determined solely by whether a univariable is local to that let definien or outside of any let definien. Univariables at that let definien could not have been introduced in another let definien because they would have already been generalized at that earlier (to the left) let definien or we would have not reached that let definien yet. Notice that the λ ranking method increments the rank much more frequently. In fact, the distinction between ranks on the right spine of the program does not make a difference with respect to generalizability. For example, the types of s and v are in the same scope of generalization (*i.e.*, they are both outside of all let definiens). The let ranking method only increments the rank at the let definiens where the rank distinction may make a difference with respect to generalizability.

Definition 9 (Let Depth)
$\mathcal{D}_{\mathsf{let}}(\widetilde{\Gamma}) = \#\ \textit{of}\ \mathsf{letd}\ \textit{markers in}\ \widetilde{\Gamma}$

Definition 10 (Univariable Let-Rank)

$$\begin{array}{rcl}
\widetilde{\mathcal{R}}_{\mathsf{let}}(\xi, \emptyset) & = & \infty \\
\widetilde{\mathcal{R}}_{\mathsf{let}}(\xi, \widetilde{\Gamma}[x : \widetilde{\tau}]) & = & \min(\widetilde{\mathcal{R}}_{\mathsf{let}}(\xi, \widetilde{\Gamma}), m) \\
 & & \textit{where}\ m = \mathcal{D}_{\mathsf{let}}(\widetilde{\Gamma})\ \textit{if}\ \xi \in \mathsf{ftv}(\widetilde{\tau}), \\
 & & \textit{otherwise}\ m = \infty \\
\widetilde{\mathcal{R}}_{\mathsf{let}}(\xi^m, \widetilde{\Gamma}[x : \widetilde{\sigma}]) & = & \widetilde{\mathcal{R}}_{\mathsf{let}}(\xi^m, \widetilde{\Gamma})
\end{array}$$

Definition 11
$\mathcal{R}_{\mathsf{let}}(\xi^m, \Gamma) = \widetilde{\mathcal{R}}_{\mathsf{let}}(\xi, \mathcal{E}_r(\Gamma))$

$\Gamma ::= \Gamma[\mathsf{letd}] \mid \ldots$ — Γ with let definien marks

$s = (c, \Gamma, d, k)$ — states

Initial Machine State $s_0 = (e, \emptyset, 0, \bullet)$

$$(x, \Gamma, d, k) \mapsto_{\mathsf{L}} (\mathcal{I}(\Gamma(x)), \Gamma, d, k) \quad [\mathsf{wl\text{-}var}]$$

$$(\lambda x.e, \Gamma, d, k) \mapsto_{\mathsf{L}} (e, \Gamma[x : \xi^d], d \quad \lambda :: k) \quad [\mathsf{wl\text{-}\lambda\text{-}in}]$$

$$(\tau, \Gamma[x : \tau'], d, \lambda :: k) \mapsto_{\mathsf{L}} (\tau' \to \tau, \Gamma, d, k) \quad [\mathsf{wl\text{-}\lambda\text{-}out}]$$

$$(e_1\ e_2, \Gamma, d, k) \mapsto_{\mathsf{L}} (e_1, \Gamma, d, (\square\ e_2) :: k) \quad [\mathsf{wl\text{-}app\text{-}l}]$$

$$(\tau, \Gamma, d, (\square\ e_2) :: k) \mapsto_{\mathsf{L}} (e_2, \Gamma, d, (\tau\ \square) :: k) \quad [\mathsf{wl\text{-}app\text{-}r}]$$

$$(\tau, \Gamma, d, (\tau'\ \square) :: k) \mapsto_{\mathsf{L}} (\theta\xi, \theta\Gamma, d, \theta k) \quad [\mathsf{wl\text{-}app\text{-}out}]$$
$$\theta = \mathcal{U}(\tau', \tau \to \xi^{\infty})$$

$$(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \Gamma, d, k) \mapsto_{\mathsf{L}} (e_1, \Gamma[\mathsf{letd}], d+1, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) \quad [\mathsf{wl\text{-}let\text{-}def}]$$

$$(\tau, \Gamma[\mathsf{letd}], d, \mathsf{let}\ x\ \mathsf{in}\ e_2 :: k) \mapsto_{\mathsf{L}} (e_2, \Gamma[x : \sigma], d-1, \mathsf{let} :: k) \quad [\mathsf{wl\text{-}let\text{-}body}]$$
$$\sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_d^{\mathsf{let}}(\tau)\}\tau, \overline{\alpha} \text{ is fresh}$$

$$(\tau, \Gamma[x : \sigma], d, \mathsf{let} :: k) \mapsto_{\mathsf{L}} (\tau, \Gamma, d, k) \quad [\mathsf{wl\text{-}let\text{-}out}]$$

$$\mathcal{G}_d^{\mathsf{let}}(\tau) = \{\xi^m \in \mathsf{ftv}(\tau) \mid m \geq d\}$$

Generic instantiation $\mathcal{I}$, language, stack register k, and frames f remain the same as in the $\mathcal{W}_\lambda$ machine in Fig. 7.

Figure 14. Algorithm $\mathcal{W}$ Machine with let-Ranking ($\mathcal{W}_{\mathsf{L}}$)

The univariable rank no longer indicates the exact λ-binding that has the first occurrence in the environment. Instead, the rank is the index of the nearest letd marker to the left of the λ-binding with the first occurrence. We can call this the *left bounding* letd *marker for the first occurrence*. Multiple, distinct univariables can share the same left bounding letd mark. These univariables have the same generalizability. As before, generic univariables are rank ∞. Univariables that are bound outside of any let definiens are rank 0. If the environment was $[x : \xi][\mathsf{letd}_1][y_1 : \phi_1][y_2\phi_2][\mathsf{letd}_2][z : \zeta]$, then ξ, ϕ_1, ϕ_2, and ζ would be ranked 0, 1, 1, and 2 respectively. The left bounding letd's for the ϕ's and ζ are letd_1 and letd_2 respectively. Also, similar to the $\mathcal{W}_\lambda$ rank function $\widetilde{\mathcal{R}}$, $\widetilde{\mathcal{R}}_{\mathsf{let}}$ still ignores let-bindings when computing the rank. As we have shown before, let-bindings have nothing to do with rank and generalizability.

Fig. 14 gives an abstract machine for a type inference algorithm that uses let-ranking. The [wl-λ-in] and [wl-λ-out] rules do change the depth counter. This is now the responsibility of the [wl-let-def] and [wl-let-body] rules that introduce and eliminate the let x in e definien binder frame on the stack respectively.

We can reuse the definition of unification from the λ-ranking system because it turns out that we can use $\mathcal{L}$ limit substitution with no changes except for using the let-ranks in place of the λ-ranks. Since instantiation during unification still shifts around the first occurrence in exactly the same way it did before, the limit substitution is still necessary in those cases where a univariable of low rank is instantiated with a type with univariables of high rank. Because it is easier for univariables to share the same let-rank than to share λ-ranks, we do not expect the limit substitution to apply as often as it did with the λ-ranking.

The [wl-let-body] rule decrements the let-depth, pops the let x in e_2 frame off the stack and the letd frame off the environment, and pushes a let onto the stack. When type checking the body of a let-expression, that let will introduce no additional polymorphism because it already generalized the types for the definien. Thus, the [wl-let-body] pops out from the definien's scope of generalization. This is why [wl-let-body] decrements the let depth. In contrast to $\mathcal{W}_\lambda$, the type environment is augmented but the depth is decremented. The let frame on the stack is a reminder to pop off this new let-binding on the environment.

As in $\mathcal{W}_\lambda$, we generalize univariables of rank greater than the depth at the point of generalization where we have popped out of the definien (and thus decremented the depth). Any univariables that we introduced inside the scope of the definien must have rank greater than or equal to the index of the letd mark we just popped off. Rémy's let-ranking scheme generalizes only those univariables that have rank exactly equal to the current depth [13]. This is due to a difference between how his and our systems rank generic univariables. His system ranks generic univariables with the current depth instead of ∞.

Many of the lemmas about the construction of $\widetilde{\mathcal{R}}$ hold for $\widetilde{\mathcal{R}}_{\mathsf{let}}$.

Lemma 10 (Rank is ∞ if not in Γ)
If $\xi \notin \mathsf{ftv}(\widetilde{\Gamma}_\lambda)$, then $\widetilde{\mathcal{R}}_{\mathsf{let}}(\xi, \widetilde{\Gamma}) = \infty$.

Proof (sketch): By straightforward induction on $\widetilde{\Gamma}$ ■

Lemma 11
If ξ^m occurs in a λ-binding in Γ that is left bounded by the nth letd *mark and ξ^m occurs in no λ-binding to the left of that* letd*, then $\mathcal{R}_{\mathsf{let}}(\xi^m, \Gamma) = n$.*

Proof (sketch): By induction on Γ and lem. 10 ■

The properties also carry over with minimal changes.

Definition 12 (Properties on $\mathcal{W}_{\mathsf{L}}$ Machine States)
The following are the properties on valid $\mathcal{W}_{\mathsf{L}}$ machine states (c, Γ, d, k):

Prop$_1$ *For all $\xi^m \in \mathsf{ftv}(\Gamma)$. $\mathcal{R}(\xi^m, \Gamma) = m$*

Prop$_2$ *For all $\xi^m \in \mathsf{ftv}(c) \cup \mathsf{ftv}(k)$ such that $m \leq \mathcal{D}_{\mathsf{let}}(\Gamma)$, $\xi^m \in \mathsf{ftv}(\Gamma)$*

Prop$_3$ *$\mathcal{D}_{\mathsf{let}}(\Gamma) =$ # of* let x in e*'s (*let *definien frames) on k*

Prop$_4$ *For all $\xi^m \in \mathsf{ftv}(k)$ such that $m \leq r$ where r is the number of* let x in e *frames up to the frame with the occurrence of ξ^m on the stack k, $\xi^m \in \mathsf{ftv}(\Gamma)$.*

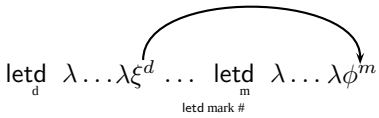
Lemma 12 (Properties Invariant Under Unification)
If the def. 12 hold for machine state valid $\mathcal{W}_{\mathsf{L}}$ machine state (τ, Γ, d, k) and $\theta = \mathcal{U}(\tau', \tau \to \zeta^{\infty})$ and τ' occurs in k and ζ is fresh, then those properties hold for $(\theta c, \Gamma, d, k)$.

Proof (sketch): This proof is very similar to that of Lemma 6. As the example in Fig. 15 shows, the limited univariable case also plays out as it did for $\mathcal{W}_\lambda$. The difference is that we look for left bounding letd marks for λ-bindings with first occurrences instead of the λ-bindings themselves. ■

Again, the properties hold trivially for the initial $\mathcal{W}_{\mathsf{L}}$ states, hence we only have to show that they hold under all the transition rules.

Lemma 13 (Properties Are Invariant For $\mathcal{W}_{\mathsf{L}}$)
For all valid $\mathcal{W}_{\mathsf{L}}$ machine states s, if the def. 12 properties hold for s and $s \mapsto_{\mathsf{L}} s'$, then the properties must hold for s'.

Proof (sketch): This proof is similar to lem. 7. Lemma 12 provides the proof for the [wl-app-out] case. The two proofs differ in that some of the work for the [λ-out] case is shifted to the [wl-let-body] since now it does the rank decrementing. The intuition is that the environment and the stack are synchronized. Furthermore, letd and let x in e (let definien frames) partition the environment



(a) Instantiating ξ^d to ϕ^m

(b) ϕ^m now has a bad rank. The first occurrence of ϕ^m is left bounded by the dth let.

(c) Limited rank of ϕ to d

Figure 15. The limit substitution for unification instantiation works similar to $\mathcal{W}_\lambda$

and stack respectively into the definien scopes that determine generalization. ■

Having shown that the properties are invariants under $\mathcal{W}_\mathsf{L}$, we can proceed to show that the generalizability criteria are equal at the point of generalization.

Lemma 14 (Equivalence of Generalizability Criteria)
If $(\tau, \Gamma[\mathsf{letd}], d, \mathsf{let}\ x\ \mathsf{in}\ e :: k) \mapsto_\mathsf{L} (e, \Gamma[x : \sigma], d-1, \mathsf{let} :: k)$ *where* $\sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_{d-1}(\Gamma)\}\tau$, *then* $\mathcal{G}_{d-1}(\Gamma) = \mathcal{G}_\Gamma(\Gamma)$.

Proof (sketch): This goes through similar to how the analogous $\mathcal{W}_\lambda$ lemma goes through. ■

We can now wrap up by showing the $\mathcal{W}_\mathsf{L}$ to be equivalent to the classic $\mathcal{W}$ machine.

Theorem 2 (Equivalence of $\mathcal{W}_\mathsf{L}$ Machine to Classic $\mathcal{W}$ Machine)
$(e, \emptyset, 0, \bullet) \mapsto^*_\mathsf{L} (\tau, \emptyset, 0, \bullet)$ *iff* $(e, \emptyset, \bullet) \mapsto^*_\mathcal{W} (\tau, \emptyset, \bullet)$

Proof (sketch): Except for the [let-body] rule and let depths that are just carried around in the rest of the machine, the $\mathcal{W}_\mathsf{L}$ machine is identical to the classic $\mathcal{W}$ machine. Lemma 14 shows that even the [wl-let-body] and [let-body] rules are equivalent. ■

6. The Value Restriction

The previous sections have considered only a pure object language. In this section, we introduce references into the object language, which complicates inference. Using Algorithm $\mathcal{W}$ without modification on a language with references admits the possibility of unsound generalization. The case in point is given in example 1. Although ref nil has a seemingly unconstrained type ξ **list ref**, ξ cannot be generalized because there is only one ref cell, which can only have one monomorphic type.

Example 1
let $x =$ ref nil
in $(x :=$ cons unit nil; hd$(!x)$ 3)

To eliminate this unsoundness, we introduce the value restriction, a particularly simple solution to type checking imperative programs. The tradeoff is that some perfectly sound pure programs no longer type check under the value restriction. In particular, this precludes returning polymorphic data structures [3]. The value restriction eliminates the need for the usually heavy type machinery necessary to prevent generalization of only the imperative univariables. An early approach, Tofte's imperative variables [17], revealed the internal use of state in types and thus did not provide implementation abstraction. Wright showed that the value restriction did not unduly limit expressiveness in practice [18]. Since then, the revised version of the Standard ML language (SML 97) adopted the value restriction in place of the earlier, more complex mechanisms for inference with imperative type variables, and production compilers such as SML/NJ have implemented it.

In Fig. 16, we introduce references and the standard imperative primitives into the language. All the notation is taken from Standard ML. We distinguish between primitive operators and types by sans-serif and **bold face** respectively. Thus, ref is the operator and **ref** is the type. We also add the conventional list type, list primitives (cons and nil), and unit to the language to make it more convenient to construct examples that use two easily distinguishable types.

Fig. 17 introduces an extension of the $\mathcal{W}$ abstract machine that directly implements the value restriction. Wright's value restriction distinguished value let's and non-value let's, a distinction reminiscent of Leroy's call-by-name and call-by-value let's [6] albeit in a much more restrictive form. Our abstract machine also distinguishes between the two syntactic forms of let by having two sets of let-rules, one for the value let's and the other for the non-value let's. The abstract machine treats value let's the same as the pure language let-ranking ($\mathcal{W}_\mathsf{L}$) machine did. To distinguish non-value let definiens, we add a new stack frame of the form $\mathsf{let_n}\ x\ \mathsf{in}\ e$. The rules [letv-def] and [letn-def] push on a value definien frame $\mathsf{let}\ x\ \mathsf{in}\ e$ and a non-value definien $\mathsf{let_n}\ x\ \mathsf{in}\ e$ frame respectively when entering the definien of the respective kind of let. After we get a type for the definien, we only generalize this type if we see a normal (value) $\mathsf{let}\ x\ \mathsf{in}\ e$ frame on top of the stack as per rule [letv-body]. Otherwise, we see a $\mathsf{let_n}\ x\ \mathsf{in}\ e$ and therefore do not generalize the definien type when adding a new binding for this definien on the environment in the [letn-body] rule. In example 1, [letn-def] will push into the outer let and [letn-body] will add the appropriate binding to the environment without generalizing ξ in the type of x.

Note that non-value let-bindings may contain first occurrences of univariables. In this sense, non-value let-bindings are the same as λ-bindings in the environment. In example 2, if the type of the definien x is ξ **list ref**, then ξ only appears in the binding introduced the non-value let in the environment. Consequently, we do not distinguish between these two kinds of bindings. Both λ's and non-value let's may put bindings that restrict generalization into the environment. They are for all intents and purposes, the same kind of binding. When the abstract machine reaches the inner let in example 2, the let_n binding in the environment contains the univariable ξ ($[x : \xi\ \mathbf{list\ ref}]$), hence ξ would not be erroneously generalized for the type of the definien y, ξ **list ref**. In order to pop off the monomorphic bindings introduced by [letn-body], we use $\mathsf{let_n}$ frames. The $\mathsf{let_n}$ frames serve the same role as λ frames except in the service of non-value let's.

Example 2
let $x =$ ref nil
in let $y = x$
in $(y :=$ cons unit nil$)(y := [\lambda x.x])$

$$
\begin{array}{rcll}
e & ::= & x \mid \lambda x.t \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{ref} \mid ! & \text{Terms} \\
 & \mid & e_1 := e_2 \mid \mathsf{cons} \mid \mathsf{nil} \mid \mathsf{unit} & \\
\tau & ::= & \alpha \mid \xi \mid \tau \to \tau \mid \tau\ \mathbf{list} \mid \tau\ \mathbf{ref} \mid \mathbf{unit} & \text{Types} \\
v & ::= & x \mid \lambda x.e \mid \mathsf{ref} \mid ! \mid \mathsf{unit} \mid \mathsf{cons} \mid \mathsf{nil} & \text{Values} \\
n & ::= & e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid t_1 := t_2 & \text{Non-values}
\end{array}
$$

Figure 16. Reference ML

Machine Stack Frames

$$
\begin{array}{rcll}
f & ::= & \mathsf{let}\ x\ \mathsf{in}\ e & \text{Value let definien} \\
 & \mid & \mathsf{let_n}\ x\ \mathsf{in}\ e & \text{Non-value let definien} \\
 & \mid & \mathsf{let_n} & \text{Non-value let body} \\
 & \mid & \ldots & \text{Frames from classical } \mathcal{W} \text{ machine}
\end{array}
$$

Value Restriction Let Rules

$$
\begin{array}{rcll}
(\mathsf{let}\ x = v\ \mathsf{in}\ e, \Gamma, k) & \mapsto_{\mathsf{V}} & (v, \Gamma, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & [\mathsf{letv\text{-}def}] \\
(\tau, \Gamma, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & \mapsto_{\mathsf{V}} & (e, \Gamma[x : \sigma], \mathsf{let} :: k) & [\mathsf{letv\text{-}body}] \\
\multicolumn{4}{r}{\sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_\Gamma(\tau)\}\tau \text{ where } \overline{\alpha} \text{ is fresh}} \\
(\mathsf{let}\ x = n\ \mathsf{in}\ e', \Gamma, k) & \mapsto_{\mathsf{V}} & (n, \Gamma, \mathsf{let_n}\ x\ \mathsf{in}\ e' :: k) & [\mathsf{letn\text{-}def}] \\
(\tau, \Gamma, \mathsf{let_n}\ x\ \mathsf{in}\ e' :: k) & \mapsto_{\mathsf{V}} & (e', \Gamma[x : \tau], \mathsf{let_n} :: k) & [\mathsf{letn\text{-}body}] \\
(\tau, \Gamma[x : \tau'], \mathsf{let_n} :: k) & \mapsto_{\mathsf{V}} & (\tau, \Gamma, k) & [\mathsf{letn\text{-}out}]
\end{array}
$$

Imperative Features

$$
\begin{array}{rcll}
(\mathsf{ref}\ e, \Gamma, k) & \mapsto_{\mathsf{V}} & (e, \Gamma, \mathsf{ref} :: k) & [\mathsf{ref\text{-}in}] \\
(\tau, \Gamma, \mathsf{ref} :: k) & \mapsto_{\mathsf{V}} & (\tau\ \mathsf{ref}, \Gamma, k) & [\mathsf{ref\text{-}out}] \\
(!e, \Gamma, k) & \mapsto_{\mathsf{V}} & (e, \Gamma, ! :: k) & [\mathsf{deref\text{-}in}] \\
(\tau, \Gamma, ! :: k) & \mapsto_{\mathsf{V}} & (\theta\xi, \theta\Gamma, \theta k) & [\mathsf{deref\text{-}out}] \\
\multicolumn{4}{r}{\xi \text{ is fresh} \qquad \theta = \mathcal{U}(\tau, \xi\ \mathsf{ref})} \\
(x := e, \Gamma, k) & \mapsto_{\mathsf{V}} & (e, \Gamma, x := \square :: k) & [\mathsf{set\text{-}in}] \\
(\tau, \Gamma, x := \square :: k) & \mapsto_{\mathsf{V}} & (\mathbf{unit}, \theta\Gamma, \theta k) & [\mathsf{set\text{-}out}] \\
\multicolumn{4}{r}{\theta = \mathcal{U}(\Gamma(x), \tau\ \mathsf{ref})}
\end{array}
$$

List Rules

$$
\begin{array}{rcll}
(\mathsf{nil}, \Gamma, k) & \mapsto_{\mathsf{V}} & (\xi\ \mathbf{list}, \Gamma, k) & [\mathsf{nil}] \\
\multicolumn{4}{r}{\xi \text{ is fresh}} \\
(\mathsf{cons}, \Gamma, k) & \mapsto_{\mathsf{V}} & (\xi\ \mathbf{list} \to \xi \to \xi\ \mathbf{list}, \Gamma, k) & [\mathsf{cons}] \\
\multicolumn{4}{r}{\xi \text{ is fresh}}
\end{array}
$$

All the rules other than [let-def] and [let-body] carry over from the classical $\mathcal{W}$ machine (Fig. 4).

Figure 17. Classical $\mathcal{W}$ with References and Value Restriction

$$
\begin{array}{rcll}
e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{ref} \mid ! & \text{Terms} \\
 & \mid & e_1 := e_2 \mid \mathsf{cons} \mid \mathsf{nil} \mid \mathsf{unit} & \\
\tau & ::= & \alpha \mid \xi^d \mid \tau \to \tau \mid \tau\ \mathbf{list} \mid \tau\ \mathbf{ref} \mid \mathbf{unit} & \text{Types} \\
v & ::= & x \mid \lambda x.e \mid \mathsf{ref} \mid ! \mid \mathsf{unit} \mid \mathsf{cons} \mid \mathsf{nil} & \text{Values} \\
n & ::= & e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid e_1 := e_2 & \text{Non-values}
\end{array}
$$

Figure 18. Reference ML with Ranked Univariables

7. Value Restriction with λ-Ranking

We can also adapt the efficient ranking schemes to work with the value restriction. Because both λ and non-value let-introduced bindings may restrict generalization, our notion of nesting depth must count them both. Fortunately, because we can introduce monomorphic bindings into the environment for non-value let's, we can count them as λ-bindings and therefore reuse $\mathcal{D}$ and $\mathcal{R}$ from the $\mathcal{W}_\lambda$ machine. The basic idea is to treat non-value let bodies as we do λ binding bodies in the foregoing machines and to distinguish non-value let definiens so that we do not confuse them with value let definiens whose types *do* generalize. However, there is one last complication in that non-value let definiens may leak out univariables that might be unsoundly generalized in the body. We deal with this situation by limiting the ranks of non-value definien types.

Value Restriction Let Rules

$$
\begin{array}{rcll}
(\mathsf{let}\ x = v\ \mathsf{in}\ e, \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (v, \Gamma, d, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & [\lambda\mathsf{v\text{-}letv\text{-}def}] \\
(\tau, \Gamma, d, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & \mapsto_{\lambda\mathsf{V}} & (e, \Gamma[x : \sigma], d, \mathsf{let} :: k) & [\lambda\mathsf{v\text{-}letv\text{-}body}] \\
\multicolumn{4}{r}{\sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_d(\tau)\}\tau \text{ where } \overline{\alpha} \text{ is fresh}} \\
(\mathsf{let}\ x = n\ \mathsf{in}\ e', \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (n, \Gamma, d, \mathsf{let_n}\ x\ \mathsf{in}\ e' :: k) & [\lambda\mathsf{v\text{-}letn\text{-}def}] \\
(\tau, \Gamma, d, \mathsf{let_n}\ x\ \mathsf{in}\ e :: k) & \mapsto_{\lambda\mathsf{V}} & (e, \Gamma'[x : \tau'], d', \mathsf{let_n} :: k') & [\lambda\mathsf{v\text{-}letn\text{-}body}] \\
\multicolumn{4}{r}{d' = d + 1 \quad \theta = \mathcal{L}_d(\tau) \quad \Gamma' = \theta(\Gamma) \quad \tau' = \theta(\tau)} \\
\multicolumn{4}{r}{k' = \theta k} \\
(\tau, \Gamma[x : \tau'], d, \mathsf{let_n} :: k) & \mapsto_{\lambda\mathsf{V}} & (\tau, \Gamma, d - 1, k) & [\lambda\mathsf{v\text{-}letn\text{-}out}]
\end{array}
$$

Imperative Features

$$
\begin{array}{rcll}
(\mathsf{ref}\ e, \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (e, \Gamma, d, \mathsf{ref} :: k) & [\lambda\mathsf{v\text{-}ref\text{-}in}] \\
(\tau, \Gamma, d, \mathsf{ref} :: k) & \mapsto_{\lambda\mathsf{V}} & (\tau\ \mathsf{ref}, \Gamma, d, k) & [\lambda\mathsf{v\text{-}ref\text{-}out}] \\
(!e, \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (e, \Gamma, d, ! :: k) & [\lambda\mathsf{v\text{-}deref\text{-}in}] \\
(\tau, \Gamma, d, ! :: k) & \mapsto_{\lambda\mathsf{V}} & (\theta\xi^\infty, \theta\Gamma, d, \theta k) & [\lambda\mathsf{v\text{-}deref\text{-}out}] \\
\multicolumn{4}{r}{\xi^\infty \text{ is fresh} \qquad \theta = \mathcal{U}(\tau, \xi^\infty\ \mathsf{ref})} \\
(x := e, \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (e, \Gamma, d, x := \square :: k) & [\lambda\mathsf{v\text{-}set\text{-}in}] \\
(\tau, \Gamma, d, x := \square :: k) & \mapsto_{\lambda\mathsf{V}} & (\mathbf{unit}, \theta\Gamma, d, \theta k) & [\lambda\mathsf{v\text{-}set\text{-}out}] \\
\multicolumn{4}{r}{\theta = \mathcal{U}(\Gamma(x), \tau\ \mathsf{ref})}
\end{array}
$$

List Rules

$$
\begin{array}{rcll}
(\mathsf{nil}, \Gamma, d, k) & \mapsto_{\lambda\mathsf{V}} & (\xi^\infty\ \mathbf{list}, \Gamma, d, k) & [\lambda\mathsf{v\text{-}nil}] \\
\multicolumn{4}{r}{\xi^\infty \text{ is fresh}} \\
(\mathsf{cons}, \Gamma, d, k) & & & \\
\multicolumn{4}{r}{\mapsto_{\lambda\mathsf{V}} (\xi^\infty\ \mathbf{list} \to \xi^\infty \to \xi^\infty\ \mathbf{list}, \Gamma, d, k) \quad [\lambda\mathsf{v\text{-}cons}]} \\
\multicolumn{4}{r}{\xi^\infty \text{ is fresh}}
\end{array}
$$

All the other rules, machine state, initial state, registers, $\mathcal{G}_d$, and $\mathcal{I}$ carry over from $\mathcal{W}_\lambda$.

Figure 19. λ-Ranking $\mathcal{W}$ with References and Value Restriction ($\mathcal{W}_{\lambda\mathsf{V}}$)

λ-ranking does nothing special with value let's. The rules look the same as in the classical $\mathcal{W}$ with value restriction except now they carry around the λ depth. In contrast, there is a slight complication for non-value let's. If we leave high ranked univariables in the non-value let definien alone, we admit the possibility that such contextual univariables may be generalized at a later point. In example 2, although the we did not generalize the type of x, $\xi\ \mathbf{list}\ \mathbf{ref}$, at the outer let, a non-value let, it may erroneously generalize at the inner let, a value let. The [λv-letn-body] avoids this unsound generalization by limiting the ranks in the non-value definien types. This measure ensures that no high rank univariable would be generalized in the body of the non-value let. For example 2, the [λv-letn-body] rule would limit the rank of ξ to 0 where the type of x is $\xi^1\ \mathbf{list}\ \mathbf{ref}$, even though [$\lambda$v-letn-def] previously incremented the λ-depth to 1 . When the machine reaches the inner let, the rank of ξ is too low to generalize. Using the intuition that non-value let $x = e_1$ in e_2 act like expressions of the form $(\lambda x.e_2)e_1$, [λv-let-body] increments the depth when entering the body (and extending the environment with a monomorphic (λ-) binding). The [λv-let-out] decrements the depth because it pops off that monomorphic binding from the environment when popping out from the body.

In the remaining rules, we simply add the depth register to the analogous classical $\mathcal{W}$ with value restriction rules. In [λv-deref-out] and [λv-nil], we introduce fresh generic univariables that should therefore be ranked ∞.

To account for the non-value let-introduced bindings in the environment, Prop_3 and Prop_4 must count both λ and $\mathsf{let_n}$ frames on the stack because nesting depth now includes both. Def. 13 shows the Prop_3 and Prop_4 updated to reflect this. This is another reason why we need [λv-letn-body] to introduce $\mathsf{let_n}$ frames as distinct from regular let frames.

Definition 13 (Properties for Value Restriction Machines)
Prop$_1$ $\forall \xi^m \in \mathsf{ftv}(\Gamma), \mathcal{R}(\xi^m, \Gamma) = m$
Prop$_2$ $\forall \xi^m \in \mathsf{ftv}(c)$ *such that* $m \leq \mathcal{D}(\Gamma)$, $\xi^m \in \mathsf{ftv}(\Gamma)$
Prop$_3$ $\mathcal{D}(\Gamma) =$ *# of* λ*'s* **and** $\mathsf{let_n}$*'s on* k
Prop$_4$ $\forall \xi^m \in \mathsf{ftv}(k)$ *such that* $m \leq r$ *where* r *is the number of* λ *and* $\mathsf{let_n}$ *frames up to the frame with the occurrence of* ξ^m *on the stack* k, $\xi^m \in \mathsf{ftv}(\Gamma)$.

We now have three cases where we use unification ([λv-deref-out], [λv-set-out], and the app-out rule), hence the properties must be invariant under all of them.

Lemma 15 (Invariants under unification)
If the properties hold for $s = (\tau, \Gamma, d, (\tau' \ \square) :: k)$, $s = (\tau, \Gamma, d, ! :: k)$, *or* $s = (\tau, \Gamma, d, x := \square :: k) \in \mathcal{VMS}$ *and* $\theta = \mathcal{U}(\tau', \tau \to \psi^\infty)$, $\mathcal{U}(\tau, \psi^\infty)$, *or* $\mathcal{U}(\Gamma(x), \tau\ \mathbf{ref})$ *respectively, then they must also hold for* $s' = (\theta\psi^\infty, \theta\Gamma, d, \theta k)$ *for the first two cases or* $s' = (\mathbf{unit}, \theta\Gamma, d, \theta k)$.

Proof (sketch): The definition of $\mathcal{U}$ has not changed at all. The other cases for s and θ are simple adaptations of the proof for lem. 6. They are actually simpler than the general [app-out] case. ■

Next, we need to show that the properties are invariant under the new rules, especially the non-value let rules.

Lemma 16 (Properties invariant over all $\mathcal{W}_{\lambda\mathsf{V}}$ rules)
If the properties hold for $\mathcal{W}_{\lambda\mathsf{V}}$ *state* $s \in \mathcal{VMS}$ *and* $s \mapsto_{\lambda\mathsf{V}} s'$, *then they hold for* s'.

Proof (sketch): Again, the [app-out] case is handled by lem. 15. All rules that were present in $\mathcal{W}_\lambda$ are handled in the same way. ■

At this point, we have everything we need to prove the analogs of Corollary 1, lemmas 8, and 9. The proofs are identical to those in $\mathcal{W}_\lambda$. We are now ready to connect $\mathcal{W}_{\lambda\mathsf{V}}$ to $\mathcal{W}_\mathsf{V}$.

Theorem 3 (Equivalence of $\mathcal{W}_{\lambda\mathsf{V}}$ and $\mathcal{W}_\mathsf{V}$)
$(t, \emptyset, 0, \bullet) \mapsto^*_{\lambda\mathsf{V}} (\tau, \emptyset, 0, \bullet)$ *iff* $(t, \emptyset, \bullet) \mapsto^*_\mathsf{V} (\tau, \emptyset, \bullet)$

Proof (sketch): This proof proceeds in the same way as thm. 1. $\mathcal{W}_{\lambda\mathsf{V}}$ and $\mathcal{W}_\mathsf{V}$ are identical except for the generalization criteria and innocuous carrying around of ranks. ■

8. Value Restriction with Let-Ranking

We can carry over much of the $\mathcal{W}_\mathsf{L}$ machine, so Fig. 20 only lists the new or changed rules in the $\mathcal{W}$ with value restriction and let-ranking machine, $\mathcal{W}_{\mathsf{LV}}$. The rules for let expressions are now also responsible for letd marking. However, it turns out that only the rules for value let's do the letd marking just as they did in $\mathcal{W}_\mathsf{L}$ and the rules for non-value let's are responsible for the definien type limitation as they were in $\mathcal{W}_{\lambda\mathsf{V}}$. Thus, for value let's, the let depth is incremented only while inside the definien ([lv-letv-def]) and it is decremented once we start the body ([lv-letv-body]) just as it is in $\mathcal{W}_\mathsf{L}$. Non-value let's, however, do not increment the let-depth because non-value let definiens introduce no scope of generalization. A non-value let $x = n$ in e is treated in the same way as would $(\lambda x.e)n$. The [lv-letn-def] rule enters the definien after pushes on a $\mathsf{let_n}$ x in e frame onto the stack. Once we get a type for the definien, the [lv-letn-body] pops off the $\mathsf{let_n}$ x in e in order to start the body under the environment extended with the limited but monomorphic definien type. We still need to limit the definien type ranks for the same reasons as in the previous section.

$$
\begin{array}{ll}
(\mathsf{let}\ x = v\ \mathsf{in}\ e, \Gamma, d, k) & \\
\qquad \mapsto_{\mathsf{LV}} (v, \Gamma[\mathsf{letd}], d+1, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & \text{[lv-letv-def]} \\
(\tau, \Gamma[\mathsf{letd}], d, \mathsf{let}\ x\ \mathsf{in}\ e :: k) & \\
\qquad \mapsto_{\mathsf{LV}} (e, \Gamma[x : \sigma], d-1, \mathsf{let} :: k) & \text{[lv-letv-body]} \\
\qquad \sigma = \forall\overline{\alpha}.\{\overline{\alpha}/\mathcal{G}_{d-1}(\tau)\}\tau \text{ where } \overline{\alpha} \text{ is fresh} & \\
(\mathsf{let}\ x = n\ \mathsf{in}\ e, \Gamma, d, k) & \\
\qquad \mapsto_{\mathsf{LV}} (n, \Gamma, d, \mathsf{let_n}\ x\ \mathsf{in}\ e :: k) & \text{[lv-letn-def]} \\
(\tau, \Gamma, d, \mathsf{let_n}\ x\ \mathsf{in}\ e :: k) & \\
\qquad \mapsto_{\mathsf{LV}} (e, (\theta\Gamma)[x : \theta(\tau)], d, \mathsf{let_n} :: \theta k) & \text{[lv-letn-body]} \\
\qquad \theta = \mathcal{L}_d(\tau) & \\
(\tau, \Gamma[x : \tau'], d, \mathsf{let_n} :: k) & \\
\qquad \mapsto_{\mathsf{LV}} (\tau, \Gamma, d, k) & \text{[lv-letn-out]}
\end{array}
$$

All imperative features, list rules, and generic instantiation $\mathcal{I}$ carry over from the $\mathcal{W}_{\lambda\mathsf{V}}$, Fig. 19. All the remaining rules, machine state, and initial state carry over from $\mathcal{W}_\mathsf{L}$, Fig. 14.

Figure 20. Let-Ranking $\mathcal{W}$ with References and Value Restriction ($\mathcal{W}_{\mathsf{LV}}$)

Non-value let's also do not add letd definien marks to the environment. Everything in the non-value let definien is no more generalizable than anything outside of that non-value let. Thus, the depth ranking here refers to the value let nesting depth only. Non-value let's are ignored just as λ's are with respect to determining the depth. So, unlike in $\mathcal{W}_{\lambda\mathsf{V}}$, [lv-letn-body] and [lv-letn-out] do not change the current depth. The [lv-letn-out] is relegated to the small role of popping off the monomorphic definien binding once we are done with the body.

Because non-value let's do not affect the let-ranking scheme, we can reuse the properties in def. 12 ($\mathcal{W}_\mathsf{L}$) and consequently all the lemmas for the $\mathcal{W}_\mathsf{L}$ machine. The new cases to the proofs are straightforward because everything, including the non-value let's can be encoded in terms of the $\mathcal{W}_\mathsf{L}$ language.

Theorem 4 (Equivalence of $\mathcal{W}_{\mathsf{LV}}$ and $\mathcal{W}_\mathsf{V}$)
$(e, \emptyset, 0, \bullet) \mapsto^*_{\mathsf{LV}} (\tau, \emptyset, 0, \bullet)$ *iff* $(e, \emptyset, \bullet) \mapsto^*_\mathsf{V} (\tau, \emptyset, \bullet)$.

Proof (sketch): This proof follows the form of the proof for $\mathcal{W}_\mathsf{L}$. The new cases are straightforward. ■

9. Related Work

Most of the literature on Algorithm $\mathcal{W}$ and the value restriction fall into two categories. The first category focuses on general frameworks for understanding $\mathcal{W}$ and the value restriction. Rémy [13] formalizes a ranked variable system using ranks based on let-depth. Pottier and Rémy [12] offer a model implementation using the let-ranking technique. As mentioned earlier, Rémy's let-ranking differs somewhat from ours when it instantiates generic univariables to the current let-depth rather than rank ∞. He also proves a principal typing result. The work relies on a complex theory based on unificands, thus providing a very general framework that can be extended to other equational theories. It does not address the value restriction. It is not clear whether this development is equivalent to our $\mathcal{W}_\mathsf{L}$ abstract machine, but we conjecture that they are similar. In contrast to Rémy's work, this paper is concerned primarily about the engineering of inference algorithms using ranked type variables. This is reflected by the presentations of the machines that more closely follow implementations.

Pottier [11] and Skalka and Pottier [15] describe a variant of the Hindley-Milner type system (HM(X)) with the value restriction. They prove its soundness. Wright provides empirical evidence that the value restriction is practical [18]. His work relies on the pure Algorithm $\mathcal{W}$ for inference after translating non-value let's away. Thus, implicitly, soundness for the value restriction is obtained by treating non-value let's as equivalent to corresponding beta-

redexes, where the bindings become λ bindings (*e.g.*, let $x = n_1$ in e_2 translates to $(\lambda x.e_2)n_1$). Pottier, Skalka and Pottier, and Wright do not address efficient algorithms for inference.

The second category focuses on expanding the set of well-typed programs by either refining the value restriction or by developing a type discipline for imperative type variables, neither of which are considered in this paper. Garrigue uses a subtyping relation to relax the value restriction and thereby expand the set of safe, well-typed programs while simultaneously preserving the implementation abstraction property of the value restriction [3]. His technique types some programs involving the handling of polymorphic data that would not type check under the ordinary value restriction. Tofte provides a soundness proof for the imperative type system of SML 90 [17]. The soundness of the value restriction is the limiting case of Tofte's proof where all type variables are imperative type variables. There are also a number of other contributions that offer more powerful and complex alternatives to the value restriction [6, 7, 16]. These treatments do not address using type variable ranks to make generalization more efficient.

10. Future Work

Both the $\mathcal{W}_{\lambda\mathsf{V}}$ and $\mathcal{W}_{\mathsf{LV}}$ machines use a rank-limiting coercion on the univariables of the let-definien type to prevent unsound generalization of definien univariables in the body. However, for efficiency, we can avoid computing and applying this limit substitution if we introduced generic univariables with more precise initial ranks.

We believe that there is an alternative to this rank coercion approach. Another version of the value restriction machines can initialize generic type variables to the current depth rather than infinity. This setup would essentially pre-limit the ranks of generic univariables in a non-value let definien. In particular, we observe that the only requirement on the rank given to a generic univariable is that the rank must be greater than or equal to $\mathcal{D}_{\mathsf{let}}(\Gamma)$. In fact, $\mathcal{D}_{\mathsf{let}}(\Gamma)$ is a perfectly sensible rank for generic univariables. The [var] rule can use a modified generic instantiation that assigns the current let depth as the rank for the new generic univariables. Similarly, the [nil] rule introduces a univariable ranked at the current depth. This technique may be related to Rémy's variant of the let-ranking scheme. We plan to investigate these alternative machines and their connection to Rémy's technique.

The abstract machine approach to formalizing inference can also be extended to other type systems and inference approaches. Local type inference offers partial inference for a variant of System $\mathrm{F}_{\leq}$ [10]. One alternative approach to inference for the Hindley-Milner type system is Lee and Yi's formalization of the folklore top-down alternative to $\mathcal{W}$, algorithm $\mathcal{M}$ [5]. The abstract machine approach may offer some further insight into these inference algorithms.

11. Conclusion

We provide a direct and accessible approach to formalizing the various versions of type inference with ranked variables. This particular formalism is also very close to implementation. Using $\mathcal{W}_{\mathsf{LV}}$ as a reference, we implemented a let-ranked machine for type checking programs under value restriction semantics. This implementation can be easily adapted to follow the other machines. This formalism also exposes some interesting properties of the type environment that we rely upon for use with the various ranking techniques.

Using this abstract machine type inference formalism, we have proved these systems correct with respect to the original Algorithm $\mathcal{W}$ (for a pure version of ML), and with respect to a version with the value restriction for the more realistic impure ML. From this experience, we believe that the formulation of type inference algorithms as abstract machines is an effective basis for reasoning about such systems and relating them to each other.

References

[1] Luís Damas. unpublished note, 1984.

[2] 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.

[3] Jacques Garrigue. Relaxing the value restriction. *Functional and Logic Programming*, 2004.

[4] George Kuan, David MacQueen, and Robert Bruce Findler. A rewriting semantics for type inference. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*, volume 4421, March 2007.

[5] 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.

[6] Xavier Leroy. Polymorphism by name for references and continuations. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–231, New York, NY, USA, 1993. ACM Press.

[7] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–302, New York, NY, USA, 1991. ACM Press.

[8] Harry G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *POPL*, pages 382–401, 1990.

[9] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[10] Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 252–265, New York, NY, USA, 1998. ACM Press.

[11] François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report RR-4150, INRIA, 2001.

[12] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[13] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[14] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

[15] Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Proceedings of the Workshop on Types in Programming (TIP'02)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, Dagstuhl, Germany, July 2002.

[16] Geoffrey Smith and Dennis Volpano. Polymorphic typing of variables and references. *ACM Trans. Program. Lang. Syst.*, 18(3):254–267, 1996.

[17] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.

[18] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.