A Rewriting Semantics for Type Inference

George Kuan, David MacQueen, and Robert Bruce Findler

University of Chicago
1100 East 58th Street, Chicago, IL 60637
{gkuan,dbm,robby}@cs.uchicago.edu

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, we begin to build the theoretical underpinnings for treating type checking 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.

1 Introduction

This paper represents our first steps in exploring a completely different way to think about the type checking process. Instead of visualizing type checking as the process of constructing a proof-tree, we explore type checking as rewriting, in the spirit of Felleisen-Hieb [11].

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. We 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 [27]) is equivalent to Algorithm $\mathscr{W}$.

In addition to using the proofs in this paper to validate these systems, all of the rewriting systems have been implemented in PLT Redex [21] and have been carefully tested (except the system in figure 7, 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 λ expressions are surrounded by parenthesis, rather than suffixed with a dot.

2 Simply Typed λ-Calculus

Fig. 1 contains the grammar and the traditional presentation of the type system for the simply typed λ-calculus (STLC). Fig. 2 contains the rules that define our type checking relation, $\mapsto_t$, which rewrites expressions to their types. The typing context T dictates that type checking proceeds from left to right. Numeric constants rewrite to the type num. λ-abstractions rewrite to an arrow type whose domain is the specified type of the parameter and whose range is the body of the original λ-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. We 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 type check get stuck without producing a final type. For example, (@ 2 3) rewrites to (@ 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 ($e_h \in \text{EXP}_\text{h}$) that contain a mixture of STLC and type syntactic forms, and encompass both STLC and type expressions (i.e., STLC $\subseteq \text{EXP}_\text{h}$ and TYPE $\subseteq \text{EXP}_\text{h}$). To see how such hybrid expressions come about, consider this reduction sequence (where the redexes have been underlined):

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

We start with a λ expression whose parameter, y, has type ($\rightarrow$ num num) and whose body is another λ expression whose parameter, x, has type num. The inner λ's body is the application of y to x. The first reduction step rewrites the outer λ-abstraction into an arrow type whose domain is ($\rightarrow$ num num) and whose range is the body of the original λ-abstraction but with all the occurrences of y replaced by ($\rightarrow$ num num), producing a hybrid term. The next step is to rewrite the remaining λ expression, this time replacing x with num. The final step is a $\tau\beta$ step. It replaces the application expression with num, because the function position's domain matches the argument position.

Theorem 1 (Soundness and Completeness for $\mapsto_t$).
For any e and τ, $\emptyset \vdash e : \tau \Leftrightarrow e \mapsto_t^* \tau$.

Proof. [sketch[1]] From left to right, the proof is a straightforward induction on the derivation of $\emptyset \vdash e : \tau$. From right to left, we must first construct the CEK machine analogue of the reduction system [8, 10], 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. □

[1] All of the proofs in this paper have been carried out in the accompanying tech report [16]; proof sketches that show only the essential ideas are presented here.

$$e ::= x \mid (\lambda\ (x\ \tau)\ e) \mid (@\ e\ e) \mid number \qquad \text{STLC}$$
$$\tau ::= \mathsf{num} \mid (\rightarrow\ \tau\ \tau) \qquad \text{TYPE}$$

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

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

Fig. 1. Grammar and traditional type system for the simply typed λ-calculus

$$e_h ::= x \mid (\lambda (x\ \tau)\ e_h) \mid (@\ e_h\ e_h) \mid number \mid (\rightarrow\ \tau\ e_h) \mid \mathsf{num} \qquad \text{EXP}_\text{h}$$
$$T ::= (@\ T\ e_h) \mid (@\ \tau\ T) \mid (\rightarrow\ \tau\ T) \mid \square$$

$$T[number] \mapsto_t T[\mathsf{num}] \qquad [\text{tc-num}]$$
$$T[(\lambda\ (x\ \tau)\ e_h)] \mapsto_t T[(\rightarrow\ \tau\ \{x \mapsto \tau\} e_h)] \qquad [\text{tc-lam}]$$
$$T[(@\ (\rightarrow\ \tau_1\ \tau_2)\ \tau_1)] \mapsto_t T[\tau_2] \qquad [\text{tc-}\tau\beta]$$

Fig. 2. Grammar and rewriting type system for simply typed λ-calculus (TC)

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

Fig. 3. Evaluation Rewriting Semantics

Although Theorem 1 guarantees that the type checker is sensible, we might wish to relate it directly to evaluation, bypassing the traditional type system. Fig. 3 gives the standard evaluation contexts and rewrite rules for call-by-value STLC.

A first cut at a direct statement of type soundness for our rewriting type system is to simply take the union of the the evaluation relation $\mapsto_e$ and the type checking 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. 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, type checking rules apply, eventually reducing to the type num. Moving down from the original term, the rule for function application applies, producing 42, which also type checks to num.

Unfortunately, the union of the relations is not confluent in general. Consider the example in Fig. 5. It is an ill-typed term, but after a single application becomes well-typed. Accordingly, the type checking rewrite rules detect the error in the original term,

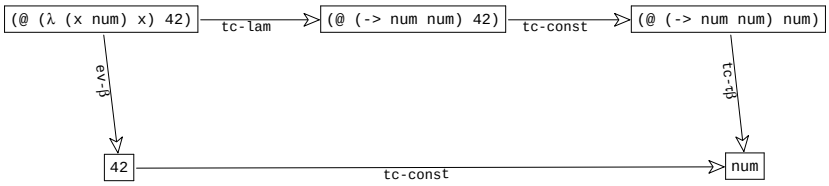


Fig. 4. Confluent examples for combined rewrite relation

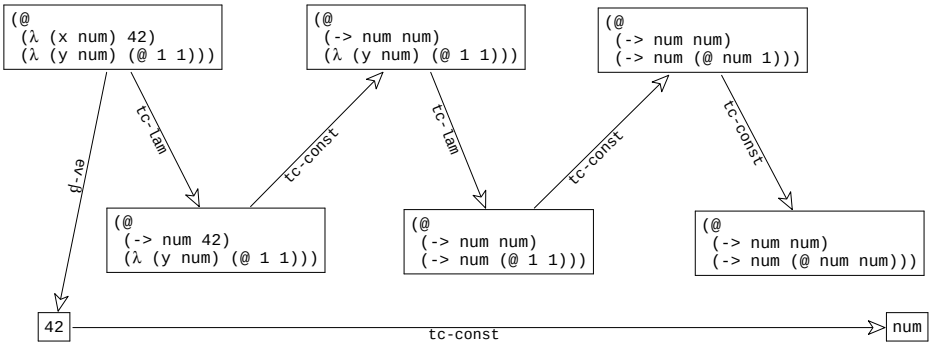


Fig. 5. Non-confluent counterexample for combined rewrite relation

but produce type num for the term after β-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 (@ (λ (x num) 42) (λ (y num) (@ 1 1))) rewrites to both num and an expression that decomposes into a type checking context with the stuck state (@ num num) in the hole. □

Nevertheless, we 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. □

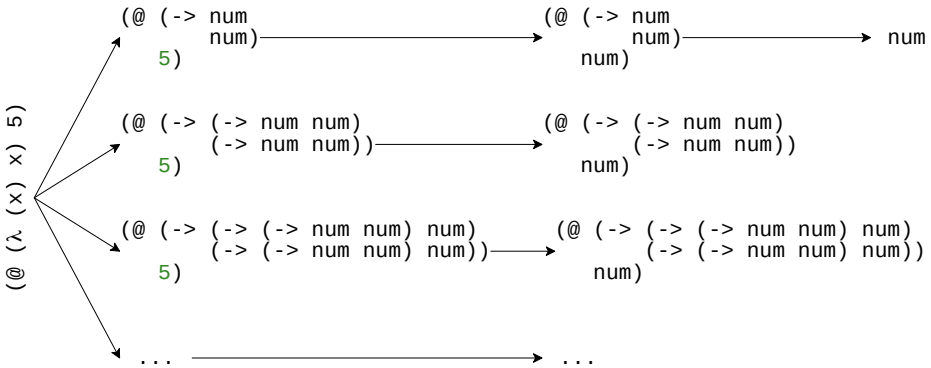


Fig. 6. Rewriting nondeterministically

$e ::= x \mid (\lambda\ (x)\ e) \mid (@\ e\ e)$ UTLC
$e_n ::= x \mid (\lambda\ (x)\ e_n) \mid (@\ e_n\ e_n) \mid \mathit{number} \mid (\rightarrow\ \tau\ e_n) \mid \mathsf{num}$ $\mathrm{EXP_n}$
$T_n ::= (@\ T_n\ e_n) \mid (@\ \tau\ T_n) \mid (\rightarrow\ \tau\ T_n) \mid \square$
$\tau ::= \mathsf{num} \mid (\rightarrow\ \tau\ \tau)$

$T_n[\mathit{number}] \mapsto_n T_n[\mathsf{num}])$ [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-$\tau\beta$]

Fig. 7. Grammar and rewrite rules for nondeterministic (ND) inference calculus

---

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 λ-calculus (UTLC), 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. 6. It begins with the application of the identity function to 5, which decomposes into a type checking context with $(\lambda\ (x)\ x)$ in the hole. Since there is no longer any constraint on the bound variable, the λ-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, we 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. 7. The [nd-lam] rule has a τ 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.

We can relate the nondeterministic system with the original one via the function $\mathscr{E}$, that maps EXP_h to EXP_n by erasing the type annotations in λ-expressions. In particular, the nondeterministic choice does not keep us from type checking terms that type checked before:

Theorem 4 (Completeness of nondeterministic reduction). *For any STLC expression e and type τ,*

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

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. □

But the implication in the reverse direction does not hold. In particular, the erasure of the term (@ (λ (x ($\rightarrow$ num num)) x) 1) has type 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 5 (Soundness of nondeterministic reduction). *For any UTLC expression e and type τ, if $e \mapsto_n^* \tau$ then there exists a STLC expression e' such that $\mathscr{E}(e') = e$ 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. □

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 (λ (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, 14]) to coping with this problem is to use unification, and so we add unification to our model, as shown in Fig. 8.

The language in Fig. 8 is the same as the one in Fig. 7, except that expressions may be wrapped with **unify** and types may be type variables (ξ). This system uses type variables and **unify** to enforce constraints between types whereas the nondeterministic system guesses types. In particular, a λ-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 type checking reductions occur does not contain **unify**, the **unify**s must be reduced before any other reductions occur. The last four reductions in Fig. 8 cover the reductions of the **unify** expressions[2]. The first removes the unification of two identical types. The second distributes the unification of two different arrow

[2] Martelli and Montanari introduced a rewriting method for performing unification[20]. Our unification system is related to theirs. Instead of explicitly transforming equation sets, we work on **unify** prefixes, each of which represents one equation.

$p ::= (\textbf{unify}\ \tau_u\ \tau_u\ p) \mid e_u$
$e_u ::= x \mid (\lambda\ (x)\ e_u) \mid (@\ e_u\ e_u) \mid number \mid (\rightarrow\ \tau_u\ e_u) \mid \mathsf{num} \mid \xi$
$T_u ::= (@\ T_u\ e_u) \mid (@\ \tau_u\ T_u) \mid (\rightarrow\ \tau_u\ T_u) \mid \Box$
$\tau_u ::= \mathsf{num} \mid (\rightarrow\ \tau_u\ \tau_u) \mid \xi$ $\quad$ $\mathrm{TYPE_u}$
$\xi ::=$ type variables $\quad$ TVAR

$T_u[number] \mapsto_u T_u[\mathsf{num}]$ $\quad$ [ch-num]
$T_u[(\lambda\ (x)\ e_u)] \mapsto_u T_u[(\rightarrow\ \xi\ \{x \mapsto \xi\}\ e_u)]$ $\quad$ [ch-lam]
ξ fresh
$T_u[(@\ \tau_{u_1}\ \tau_{u_2})] \mapsto_u (\textbf{unify}\ \tau_{u_1}\ (\rightarrow\ \tau_{u_2}\ \xi)\ T_u[\xi])$ $\quad$ [ch-$\tau\beta$]
ξ fresh
$(\textbf{unify}\ \tau_{u_1}\ \tau_{u_1}\ p) \mapsto_u p$ $\quad$ [ch-u-eq]
$(\textbf{unify}\ (\rightarrow\ \tau_{u_1}\ \tau_{u_2})\ (\rightarrow\ \tau_{u_3}\ \tau_{u_4})\ p) \mapsto_u (\textbf{unify}\ \tau_{u_1}\ \tau_{u_3}\ (\textbf{unify}\ \tau_{u_2}\ \tau_{u_4}\ p))$ $\quad$ [ch-u-dist]
$(\rightarrow\ \tau_{u_1}\ \tau_{u_2}) \neq (\rightarrow\ \tau_{u_3}\ \tau_{u_4})$
$(\textbf{unify}\ \tau_u\ \xi\ p) \mapsto_u (\textbf{unify}\ \xi\ \tau_u\ p)$ $\quad$ [ch-u-orient]
$\tau_u \neq \xi$
$(\textbf{unify}\ \xi\ \tau_u\ p) \mapsto_u \{\xi \mapsto \tau_u\}p$ $\quad$ [ch-u-inst]
$\xi \notin \mathrm{ftv}(\tau_u)$
$\mathrm{ftv}(\tau)$ = set of type variables occurring in τ.

Fig. 8. Grammar and rewrite rules for Curry/Hindley calculus

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 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. 6:

$$\begin{array}{ll} & (@\ (\lambda\ (x)\ x)\ 5) \\ \mapsto_u & (@\ (\rightarrow\ \xi\ \xi)\ 5) \\ \mapsto_u & (@\ (\rightarrow\ \xi\ \xi)\ \mathsf{num}) \\ \mapsto_u & (\textbf{unify}\ (\rightarrow\ \xi\ \xi)\ (\rightarrow\ \mathsf{num}\ \xi')\ \xi') \\ \mapsto_u^* & \mathsf{num} \end{array}$$

The first step generates a fresh type variable and replaces the λ-expression with an arrow type whose domain and range are that type variable. As in Fig. 6, 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, we may encounter a number of these reductions that generate multiple next states. For any particular combination of choices of next states, we can construct a ground type substitution γ: TVAR $\rightarrow$ TYPE that instantiates the type variables in the CH reduction sequence to those types chosen in the ND reduction sequence. We can exploit this correspondence to make the relation-

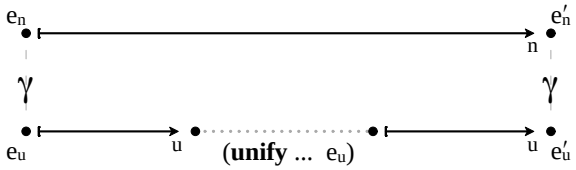
ship between the two type checking relations precise, via the ground type substitution γ for a complete ND reduction sequence $e \mapsto_n^* \tau$.

Theorem 6 (Nondeterministic & Curry/Hindley typing relationship).
Let $e \in$ UTLC.

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

Soundness *If* $e \mapsto_u^* \tau_u$ *then for all ground types* τ *that are instantiations of* τ_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 γ 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, we 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 τ_i. For each [nd-lam] step, associate a fresh type variable ξ_i. The ground type substitution maps each ξ_i to τ_i. The [ch-lam] steps should use ξ_i for the fresh type variable when reducing the λ-binder for x_i.

For the soundness part, the ground type substitution is the composition of two substitutions. First, we 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, we 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 ξ introduced by the corresponding [ch-lam] reduction.

Three other essential facts must be established. First, we 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 γT_u is a T_n and γe_u is an e_n. Second, that the composition of two most general unifiers is also a most general unifier (due to Robinson [28]) and finally that the **unify** reductions perform unifications consistent with a most general unifier. □

$e ::= x \mid (\lambda\ (x)\ e) \mid (@\ e\ e) \mid \mathit{number} \mid (\textbf{let}\ (x\ e)\ e)$ UTLC$_\ell$
$e_p ::= x \mid (\lambda\ d\ (x)\ e_p) \mid (@\ e_p\ e_p) \mid \mathit{number} \mid (\textbf{let}\ d\ (x\ e_p)\ e_p) \mid \tau \mid \sigma$
$T_p ::= (\textbf{let}\ d\ (x\ T_p)\ e_p) \mid (@\ T_p\ e_p) \mid (@\ \tau\ T_p) \mid (\rightarrow\ \tau\ T_p) \mid \Box$
$p ::= (\textbf{unify}\ \tau\ \tau\ p) \mid e_p$
$\tau ::= \mathsf{num} \mid \xi^d \mid (\rightarrow\ \tau\ \tau)$
$\sigma ::= (\forall\ \overline{\alpha}\ (\{\overline{\xi} \mapsto \overline{\alpha}\}\tau))$
$d ::= 0 \mid 1 \mid \ldots \mid \infty$

Fig. 9. Grammar for Hindley/Milner calculus

4 Hindley/Milner inference

As a practical matter, it is important to add a **let**-form to our language so that programmers can bind a single value and use it with multiple types. A **let** expression has the form (**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 λ expression: $((\lambda\ (x)\ e_2)\ e_1)$.

Type checking a **let** expression by replacing it with such an application, however, yields a type checker 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 type check, even though the original program certainly is safe.

A naive type checker 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. β-reducing the redex that the let expression abbreviates). Then each resulting occurrence of the argument expression could be type checked separately in its own context within the body, allowing the type checker to infer different types for different uses of the bound variable.

Unfortunately, such a scheme involves redundant work in the type checker and possibly duplicated type error messages. To avoid this redundancy, Milner developed a type checking algorithm [4, 24] that achieves the same result as the substitution by splitting the type checking 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 type checking the definiens in the context of the whole **let** expression, and then partitioning the unconstrained type variables in the result into two sets: *polymorphic* variables that can be safely instantiated to different types at each occurrence of the **let**-bound vaiable, and those that cannot because they are constrained by the outer context. The result is represented as a polymorphic type $\forall\overline{\alpha}.\tau$, where the type variables in $\overline{\alpha}$ are the polymorphic, or generalizable, variables.

If we 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

$E[(@\ (\lambda\ (x)\ e_p)\ v)] \mapsto_e E[\{x \mapsto v\}e_p]$ [ev-beta]

$E[(\textbf{let}\ d\ (x\ e_{p1})\ e_{p2})] \mapsto_e E[\{x \mapsto e_{p1}\}e_{p2}]$ [ev-let]

$T_p[number] \mapsto_p T_p[\mathsf{num}]$ [tcp-num]

$T_p[(\lambda\ d\ (x)\ e_p)] \mapsto_p T_p[(\rightarrow \xi^d\ (\{x \mapsto \xi^d\}e_p))]$ [tcp-lam]
ξ^d fresh

$T_p[(@\ \tau_1\ \tau_2)] \mapsto_p (\textbf{unify}\ \tau_1\ (\rightarrow \tau_2\ \xi^\infty)\ T_p[\xi^\infty])$ [tcp-$\tau\beta$]
ξ^∞ fresh

$T_p[(\textbf{let}\ d\ (x\ \tau)\ e_p)] \mapsto_p T_p[\{x \mapsto (\forall\ \overline{\alpha}\ \tau_1)\}\ e_p]$ [tcp-let]
$\overline{\alpha}$ fresh and $\tau_1 = \{\mathscr{G}(\tau\ ,\ d) \mapsto \overline{\alpha}\}\tau$

$T_p[(\forall\ \overline{\alpha}\ \tau)] \mapsto_p T_p[\{\overline{\alpha} \mapsto \overline{\xi^\infty}\}\tau]$ [tcp-poly]
$\overline{\xi^\infty}$ fresh

$(\textbf{unify}\ \xi^d\ \tau\ p) \mapsto_p (\mathscr{L}(\tau\ ,\ d))(\{\xi^d \mapsto \tau\}p)$ [tcp-u-inst]
$\xi^d \notin \mathrm{ftv}(\tau)$

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.

$\mathscr{G} : \tau \times depth \rightarrow \xi\ list$
$\mathscr{G}(\tau, d) = \{\xi^{d'} \in \mathrm{ftv}(\tau) \mid d < d'\}$
$\mathscr{L} : \tau \times depth \rightarrow \xi\ depth\ substitution$
$\mathscr{L}(\tau, d) = \{\xi^{d'} \mapsto \xi^d \mid \xi^{d'} \in \mathrm{ftv}(\tau) \text{ and } d < d'\}$

Fig. 10. Rewrite rules for Hindley/Milner Type Inference

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 *maximal 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 ξ^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 τ is substituted for ξ^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 bindings at even lower depths. Thus substitution for a variable of rank d entails globally resetting depths of type veriables found the substituted type to have rank at most d to reflect 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, then it does not appear in the context and thus can be considered polymorphic.[3] The fresh type variables used to generically instantiate the polymorphic type of a let-bound variable occurrence start with rank ∞, reflecting the fact that they initially are not free in the type of any lambda-bound variables.

[3] Some rank systems, like the one described here and the one used in the SML/NJ type checker, are based on nesting levels of lambda bindings. Other closely related systems, such Rémy's [27] and McAllester's [23], are based on nesting levels of let bindings.

Our rewriting system for Hindley/Milner inference is presented in Figs. 9 and 10. We 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.

$$(\lambda\ (x)\ (\lambda\ (y)\ (\textbf{let}\ (z\ (\lambda\ (u)\ y))\ (@\ (@\ z\ x)\ (@\ z\ 1)))))$$
$$(\lambda\ 1\ (x)\ (\lambda\ 2\ (y)\ (\textbf{let}\ 2\ (z\ (\lambda\ 3\ (u)\ y))\ (@\ (@\ z\ x)\ (@\ z\ 1)))))$$

The rewriting system operates on such labeled expressions from the e_p grammar. When a type variable ξ^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 λ-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 ξ_x^1, ξ_y^2, and ξ_u^3 respectively. Fresh type variables used to create a generic instance of a polytype in rule [tcp-poly] are given a depth of ∞, 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 τ that may contain other type variables. This rule enforces the maximal rank property discussed above by applying a depth-adjustment substitution $\mathscr{L}(\tau, d)$. 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

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

The type rewriting of this expression proceeds as follows:

$$(\lambda\ 1\ (x)\ (\textbf{let}\ 1\ (f\ (\lambda\ 2\ (y)\ (@\ x\ y)))\ (@\ f\ 5))) \tag{1}$$
$$\mapsto_p^*\ \ (\rightarrow\ \xi_x^1\ (\textbf{let}\ 1\ (f\ (\rightarrow\ \xi_y^2\ (@\ \xi_x^1\ \xi_y^2)))\ (@\ f\ 5))) \tag{2}$$
$$\mapsto_p\ \ (\textbf{unify}\ \xi_x^1\ (\rightarrow\ \xi_y^2\ \xi_3^\infty)\ (\rightarrow\ \xi_x^1\ (\textbf{let}\ 1\ (f\ (\rightarrow\ \xi_y^2\ \xi_3^2)))\ (@\ f\ 5)))) \tag{3}$$
$$\mapsto_p\ \ (\rightarrow\ (\rightarrow\ \xi_y^1\ \xi_3^1)\ (\textbf{let}\ 1\ (f\ (\rightarrow\ \xi_y^1\ \xi_3^1)))\ (@\ f\ 5)))) \tag{4}$$
$$\mapsto_p\ \ (\rightarrow\ (\rightarrow\ \xi_y^1\ \xi_3^1)\ (@\ (\rightarrow\ \xi_y^1\ \xi_3^1)\ 5)))) \tag{5}$$
$$\mapsto_p^*\ \ (\rightarrow\ (\rightarrow\ \mathsf{num}\ \xi_4^1)\ \xi_4^1) \tag{6}$$

The expression at line (2) is obtained by two applications of [tcp-lam] to rewrite the λx and λy binders, introducing the rank 1 type variable ξ_x^1 and the rank 2 type variable ξ_y^2. At line (3), the application in the definiens of f is rewritten using [tcp-$\tau\beta$], introducing the fresh type variable ξ_3^∞ 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 ξ_y^1 is accompanied by the reduction of the ranks of ξ_y^2 and ξ_3^∞ 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 $\mathscr{G}((\rightarrow\ \xi_y^1\ \xi_3^1), 1)$ 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 ($\forall\alpha\ (\rightarrow\ \alpha\ \alpha)$).

We prove the correctness of this typing rewrite system for let-polymorphism, which we will call HM, by showing that it is equivalent to a slightly modernized variant [17] of Milner's Algorithm $\mathscr{W}$ [24]. We assume this Algorithm $\mathscr{W}$ defines a function $\mathscr{W}(\Gamma, e)$, where Γ is a type assignment mapping variables to types, returning a pair (θ, τ), where θ is a type substitution mapping type variables to types (either monomorphic or polymorphic). $\mathscr{W}$ has the property that:

$$\mathscr{W}(\Gamma, e) = (\theta, \tau) \implies \theta(\Gamma) \vdash e : \tau$$

Theorem 7 (HM Rewrite Soundness and Completeness relative to $\mathscr{W}$).
For any closed UTLC$_\ell$ expression e, let e_l be the depth-labeled version of e.
Then $e_l \mapsto_p^* \tau$ iff $\mathscr{W}(\emptyset, e) = (\theta, \tau)$.

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 $\mathscr{W}$ and the rewriting system in Fig. 10 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 (e_p, Γ, Σ, K) where e_p is the control (C), Γ is an environment mapping program variables to types (E), Σ (the extra register) is a list of substitutions that map type variables to types, and K is the type checking context. The Σ register is used to maintain a correspondance between the machine's states and the substitutions that recursive calls in Algorithm $\mathscr{W}$ 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. 10. For example, this rule:

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

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 Σ register is maintained by the unification rules, and the rules that pop contexts. The complete set of rules are given in the first author's master's paper [16].

An important technical element for relating the rewrite system, the abstract machine, and Algorithm $\mathscr{W}$ 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 ξ^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 ξ^d appears in the binding prefix derived from the context of the **let**. □

5 Related Work

Prior approaches to type checking 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 type checking for a simple arithmetic language. Pašalić et al [26] give a graph rewriting system for the original formulation of Hindley/Milner inference without explicit generalization. In contrast, our 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; our work has a concrete, operational flavor where his is more denotational. Kahrs [15] has a different formulation of type checking via abstract interpretations; while his is more operational, like ours, it is based on a translation to a machine-like language; ours operates directly on the program text.

There have been several alternative presentations of type inference algorithms. Wand's algorithm [31] 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 $\mathscr{W}$ type inference focuses on retaining information from intermediate steps of the process. Soosaipillai [29] maintains a list of the types inferred for each subexpression. Duggan and Bent [9] as well as Wand [30] 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. Instead of retaining specific information during inference, we present the entire process in terms of simple rewrite steps. We 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, 12, 13, 17, 18, 19, 22, 25, 32, 33]. Like that work, we too are motivated by the desire to make type checking easier to understand. Generally speaking, that work augments existing type checking algorithms with more information or improves existing type checking algorithms in order to improve the error messages produced by the type checker. Our work, in contrast, is an entirely different way to think about the behavior of a type checker.

6 Conclusion

Our vision is that this work forms the technical foundation for a more ambitious program to make type checkers easier to understand. Our 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 type checkers are based directly on the program text and the rewrite rules are relatively straightforward counterparts to evaluation, we believe students can more easily gain an intuition for how a type checker behaves by studying them. Similarly, we expect to be able to exploit the operational flavor of the type checking rewrite rules to build debuggers to help more experienced programmers understand why ill-typed programs fail to type check.

Although we believe this work succeeds in providing an accessible model for typing the simply typed λ-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, we have managed to produce a correct version of Hindley-Milner polymorphism and to provide, to the best of our knowledge, the first proof that an algorithm based on depth-numbering is equivalent to Algorithm $\mathscr{W}$.

Looking to the future, we 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, 1982.

[8] 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`.

[9] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[10] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Revision of 1989 edition, 2003.

[11] 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.

[12] Christian Haack and Joseph B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[13] 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.

[14] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–40, 1969.

[15] Stefan Kahrs. Polymorphic type checking by interpretation of code. Technical Report ECS-LFCS-92-238, University of Edinburgh, 1992.

[16] George Kuan. A rewriting semantics for type inference. Technical Report TR-2007-01, University of Chicago, 2007.

[17] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[18] 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.

[19] Xavier Leroy. Programmation du système Unix en Caml Light. Technical report 147, INRIA, 1992.

[20] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[21] 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.

[22] Bruce McAdam. How to repair type errors automatically. *Trends in functional programming*, pages 87–98, 2002.

[23] David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications*, 2003.

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

[25] 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.

[26] Emir Pašalić, Jeremy G. Siek, and Walid Taha. Concoqtion: Mixing indexed types and Hindley-Milner type inference. In *POPL '07: Conference record of the 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007.

[27] 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.

[28] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[29] Helen Soosaipillai. An explanation based polymorphic type checker for Standard ML. Master's Thesis, 1990.

[30] 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.

[31] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[32] 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.

[33] Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, 2001.