

Explaining Type Inference*

Dominic Duggan[†]
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada.
dduggan@uwaterloo.ca

Frederick Bent
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada.
fwbent@uwaterloo.ca

UW TR CS-94-14. January 1994. Revised March 1994 and January 1995.

Abstract

Type inference is the compile-time process of reconstructing missing type information in a program based on the usage of its variables. ML and Haskell are two languages where this aspect of compilation has enjoyed some popularity, allowing type information to be omitted while static type checking is still performed. Type inference may be expected to have some application in the prototyping and scripting languages which are becoming increasingly popular. A difficulty with type inference is the confusing and sometimes counter-intuitive diagnostics produced by the type checker as a result of type errors. A modification of the Hindley-Milner type inference algorithm is presented, which allows the specific reasoning which led to a program variable having a particular type to be recorded for type explanation. This approach is close to the intuitive process used in practice for debugging type errors.

1 Introduction

Type inference refers to the compile-time process of reconstructing missing type information in a program based on the usage of some of its variables [10]. For example, for the ML function expression `fn x => x + 1`, since mixed mode arithmetic is not supported, the occurrence of `1` implies that the integer version of `+` is being used. This further constrains the type of `x` to be integer, so we may conclude that the type of the above expression is `int → int` (where `→` is the procedure or function type).

ML and Haskell are two popular languages currently supporting type inference [30, 21]. Since these languages do not require the types of variables to be declared, they are said to be *implicitly typed languages*. In the example above, we consider the original implicitly typed expression `fn x => x + 1` to be a syntactic abbreviation for `fn x : int => x + 1`, where the missing type annotation is recovered as a result of type checking. In fact, modulo certain restrictions concerning overloaded variables and imperative features, it is

*This work has been submitted for publication. Copyright may be transferred without further notice and this version may no longer be accessible.

[†]Supported by NSERC Operating Grant 0105568.

possible to submit a Haskell or ML program without type information to the compiler and have the compiler completely reconstruct the missing type information and determine if the program is well-typed.

For the function expression

```
fn f => fn x => f x
```

the compiler reconstructs the type

$$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

where α and β are *type variables*, and the “reconstructed” term is `fn f : $\alpha \rightarrow \beta$ => fn x : α => f x`. Type variables are used in two senses in these languages. If the above expression occurs as a subexpression in a larger expression, then α and β are *non-generic type variables*; the larger context in which this expression occurs may place further constraints on the type of the expression which requires the type variables to be instantiated further. For example, if the expression is applied to a procedure which converts integers to strings, then α is constrained to be integer and β is constrained to be string. On the other hand, if this expression is the definition for a defined variable (introduced by a `let`-statement), then α and β are generalized to *generic type variables*, meaning that they receive possibly different instantiations at each use site for the variable. In the latter case the expression is said to be *parametrically polymorphic* [10]. For example, in the expression:

```
let ap = fn f => fn x => f x in ( ap succ 1; ap size "hello" )
```

the defined variable `ap` has its type variables instantiated in different ways at the two use sites. One explanation for this polymorphism is in terms of an underlying explicit polymorphic abstraction over type variables `Fn α => e`, so type inference can be seen as the process of reconstructing the following explicitly typed program:

```
let ap = Fn  $\alpha$  => Fn  $\beta$  => fn f: $\alpha \rightarrow \beta$  => fn x: $\alpha$  => f x in
  ( ap {int} {int} succ 1; ap {string} {int} size "hello" )
```

where `ap {string} {int}` denotes the application of the polymorphic function `ap` to the types `string` and `int`. The type of this polymorphic function is then $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$, denoted by $(\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \mathbf{a} \rightarrow \mathbf{b}$ in ML and $(\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \mathbf{a} \rightarrow \mathbf{b}$ in Haskell.

Even in languages which do require the types of variables to be declared, type inference is often found to be essential, in reconstructing the instantiations for generic type variables at the use sites for polymorphic functions¹ [42, 8]. Although there is a move away from implicit typing in certain circles of the ML community, it is still popular in the community as a whole and in other language communities such as Miranda², Haskell, λ -Prolog and Gödel [51, 21, 31, 20]. Moreover we may expect implicit typing to play an increasingly important role in *scripting languages* such as TCL and Python [38, 52], where currently the demands of rapid prototyping and fast development of “throw-away” programs have been used to justify the absence of static typing. (Although these languages were intended for writing short interactive “scripts,” TCL programs containing several thousand lines of code are now common.)

A vexing problem for type inference is the need to explain typing errors to programmers. As a preliminary indication of the problems, consider the following ML example (which we will use again later in the paper):

¹This special case of type inference is referred to as *type reconstruction*. Unfortunately the terminology is not universal in the literature, so “type reconstruction” sometimes refers to type inference.

²Miranda is a trademark of Research Associates, Ltd.

```
... F y; ... y=(3,x); ... F (z,4.5);
```

In this example, the type-checker will determine that the type of `x` is `real`. Why? The first application (`F y`) constrains `F` to be a function, with domain type equated to the type of `y`. The equality (`y=(3,x)`) constrains `y` to be a product (record) type, whose second component type is the same as the type for `x`. Finally the third application constrains `F`'s domain type to be a product type whose second component type is `real`. Now since `F`'s domain type is also equated with `y`'s type, this transitively constrains `x`'s type to be `real`.

In our experience with using ML and with teaching ML in a fourth year programming languages course, such convoluted reasoning about the type inference process becomes necessary fairly quickly when programming in an implicitly typed language. Unfortunately the support provided by current type-checkers for this reasoning is found wanting. Rather than trying to automatically discover the source of type errors (we do not believe there is a general solution to this which scales up), our objective is to provide better explanations for why particular types were inferred for program variables, to help the programmer in tracking down the source of type errors.

There have been a few approaches to providing explanations for type errors with type inference. Nikhil [32] describes a practical realization of type inference, where the main consideration for reporting type errors is to report the application site where the type error arose (this is now standard in most type checkers). In our experience, type errors often arise when the type-checker infers an erroneous type for a function based on its use in a (possibly quite large) mutually recursive function definition; but this error only manifests itself in a confusing type error when the function definition itself is subsequently type checked. For example (in ML):

```
fun f x = g 3
...
and g (x::xs) = x (* type error! *)
```

Wand [54] gives an algorithm for collecting the reasons for bindings of type variables, accumulating these reasons while traversing two type trees in parallel, and reporting the two accumulated lists of source expressions (as potential sources of the error) when a type error is discovered. Although conceptually superior to the current practice, Wand's approach does not scale up to examples of reasonable size, eventually overwhelming the programmer with possible error sites. For our purposes Wand's underlying algorithm is deficient in several respects, including repeated explanations and aliasing of type variables (which is fairly ubiquitous in practice, but for which Wand's algorithm would give incomplete and sometimes incorrect explanations); nor does he consider practical aspects of the implementation of unification (such as path compression [1, 33]). Nevertheless his approach is a good starting point for understanding our approach to explaining the types of individual program variables. The underlying unification algorithm in Wand's work is essentially the standard implementation unchanged; we identify several changes which are necessary in the usual implementation of unification [1, Chapter 6] in order to support type explanation.

Johnson and Walz [24, 53] use weightings on bindings and flow analysis to determine the most likely correct types for variables in reporting type errors. Although this approach appears more likely to scale, it can lead to counter-intuitive results. Consider for example changing the formal parameters of a function definition but neglecting to update all the call sites which follow that function's definition; under this approach, conceivably the type checker would report an error in the function definition. Soosaipillai [47] provides a type explainer which provides an explanation of the type of a function by walking over an explicit type derivation tree for that function. However it is not clear from her thesis what explanation is provided for the actions of the unification algorithm. As the example above demonstrates, it is here that the real issues in explaining

VAR	$\frac{(x \in \text{dom}(A))}{A \vdash x : A(x)}$
GEN	$\frac{A \vdash e : \sigma \quad (\alpha \notin \text{FV}(A))}{A \vdash e : \forall \alpha. \sigma}$
INST	$\frac{A \vdash e : \forall \alpha. \sigma}{A \vdash e : \{\tau/\alpha\}\sigma}$
ABS	$\frac{A, x : \tau \vdash e : \tau'}{A \vdash \mathbf{fn} \ x \Rightarrow e : \tau \rightarrow \tau'}$
APP	$\frac{A \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash e_1 \ e_2 : \tau_1}$
LET	$\frac{A \vdash e_1 : \sigma \quad A, x : \sigma \vdash e_2 : \tau}{A \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau}$
REC	$\frac{A, x : \tau \vdash e : \tau}{A \vdash \mathbf{rec} \ x \Rightarrow e : \tau}$

Figure 1: Type Rules for Mini-language

type inference arise. Furthermore what we seek is some facility for isolating the actions of the type checker which lead to a particular type being inferred for a variable.

We consider a *type explanation* to be a record of the subexpressions in a program which led the type checker to infer a particular type for a program variable. In this paper, we consider a modification of the type inference algorithm which allows this form of explanatory information to be saved as part of type unification, forming the basis for automatically providing an explanation such as given earlier to explain the actions of the type-checker. This approach is intuitively close to the process we have used when manually debugging ML type errors, and as such appears more likely to scale than earlier approaches.

In the next section we provide an overview of type inference, giving a very simple ML implementation which forms a basis for discussing the implementation of type explanation. In Sect. 3 through Sect. 5 we describe the algorithm for collecting type explanation information during type inference. In Sect. 6 we formalize our notion of a type explanation. In Sect. 7 we demonstrate how our approach to type explanation can be extended to other unification algorithms besides the graph-based Robinson unification algorithm used in the paper. Finally Sect. 8 provides our conclusions.

2 Type Inference

In this section we review the main results in type inference. A good discussion of type inference, polymorphic types and unification algorithms may be found in the Aho, Sethi and Ullman compiler text [1, Chapter 6] and in Cardelli's tutorial paper [10]. We give a traditional abstract presentation of the algorithms; we then give a concrete implementation in ML of unification. This concrete implementation forms the basis for the remainder of the paper, where we explain our approach to type explanation.

We use a mini-language which amounts to the extended λ -calculus underlying most functional languages

(and many Algol derivatives):

$$\begin{aligned}
e \in \text{Terms} & ::= x \mid \mathbf{fn} \ x \Rightarrow e \mid (e_1 \ e_2) \mid \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \mid \mathbf{rec} \ x \Rightarrow e \\
\tau \in \text{Monotypes} & ::= \alpha \mid \mathbf{t}(\tau_1, \dots, \tau_m) \mid \tau_1 \rightarrow \tau_2 \\
\sigma \in \text{Polytypes} & ::= \tau \mid \forall \alpha. \sigma
\end{aligned}$$

$\mathbf{rec} \ x \Rightarrow e$ defines the least fixed point of the recursive equation $x = e$, and is used to define recursive functions. Figure 1 gives the type rules for our mini-language. Type variables 'a are denoted by Greek letters α in the type rules; all type variables in the type rules are generic. As usual we use $A \vdash e : \sigma$ to denote both the corresponding sequent in the type rules, and the derivability of such a sequent from the type rules, relying on the reader's wits to distinguish them. A *substitution* θ is a mapping from type variables to monotypes which is the identity on all but a finite number of variables, sometimes denoted $\{\tau_1/\alpha_1, \dots, \tau_m/\alpha_m\}$. We define the domain and composition of substitutions as:

$$\begin{aligned}
\text{dom}(\theta) & \stackrel{\text{def}}{=} \{\alpha \mid \theta(\alpha) \neq \alpha\} \\
\theta_1 \circ \theta_2 & \stackrel{\text{def}}{=} \{\theta_1(\theta_2(\alpha))/\alpha \mid \alpha \in \text{dom}(\theta_1) \cup \text{dom}(\theta_2)\}
\end{aligned}$$

We define the homomorphic extension of substitutions to types:

$$\begin{aligned}
\theta(\mathbf{t}(\overline{\tau_m})) & = \mathbf{t}(\overline{\theta(\tau_m)}) \\
\theta(\forall \alpha. \sigma) & = \forall \beta. \theta(\{\beta/\alpha\}\sigma) \text{ where } \beta \notin \text{dom}(\theta)
\end{aligned}$$

Define the following instance ordering on polytypes, and instantiation ordering on type environments:

Definition 2.1 *Define:*

1. $\sigma_1 \sqsubseteq \sigma_2$ if and only if there exists some substitution θ such that $\sigma_1 = \theta(\sigma_2)$.
2. $A \leq A'$ if and only if $\text{dom}(A) = \text{dom}(A')$ and, for all $\alpha \in \text{dom}(A)$, $A(\alpha) = \forall \overline{\beta_m}. \tau$ and $A'(\alpha) = \forall \overline{\gamma_n}. \theta(\tau)$ for some θ such that $\text{dom}(\theta) \subseteq \{\beta_i\}_i, \{\gamma_j\}_j \cap FV(A') = \{\}$.

The most important result we require here is:

Lemma 2.1 (Substitution Lemma) *If $A \vdash e : \sigma$, then for any substitution θ , we have $\theta A \vdash e : \theta \sigma$.*

Figure 2 gives the Hindley-Milner algorithm for reconstructing the type of a program based on the usage of its variables. The algorithm walks over the abstract syntax tree of the program, introducing a new non-generic type variable for the type of each program variable, collecting constraints on these type variables which are imposed by the use of the program variables, and using unification to resolve these constraints and compute most general unifiers. Figure 3 gives the standard Robinson unification algorithm [44].

Once the definition in a \mathbf{let} -expression has been type checked, any remaining type variables which are not constrained by the program variables in the environment are generalized to generic type variables. At each use site of the variable bound to this definition, these generic type variables are instantiated with new non-generic type variables, thus allowing the definition to be used polymorphically. The algorithm makes use of the following metafunction:

$$\forall_A(\tau) = \forall \alpha_1 \dots \forall \alpha_m. \tau \text{ where } \{\alpha_1, \dots, \alpha_m\} = FV(\tau) - FV(A)$$

which corresponds to lifting the uninstantiated non-generic type variables in an inferred type to generic type variables.

The correctness of these algorithms is provided by the following [17]:

$$\begin{aligned}
\mathcal{W}'_A(x) &= \langle \theta\tau, \theta \rangle \\
&\text{where } \begin{cases} \forall \alpha_1 \cdots \forall \alpha_m \cdot \tau = A(x) \\ \theta = \{\overline{\beta_m / \overline{\alpha_m}}\} \text{ where the } \beta_i \text{'s are new type variables} \end{cases} \\
\mathcal{W}'_A(\mathbf{fn } x \Rightarrow e) &= \langle \tau \rightarrow \tau', \theta \rangle \\
&\text{where } \begin{cases} \langle \tau', \theta \rangle = \mathcal{W}'_{A,x:\alpha}(e) \text{ where } \alpha \text{ is a new type variable} \\ \tau = \theta\alpha \end{cases} \\
\mathcal{W}'_A(e_1 e_2) &= \langle \theta_3\alpha, \theta_3 \circ \theta_2 \circ \theta_1 \rangle \\
&\text{where } \begin{cases} \langle \tau_1, \theta_1 \rangle = \mathcal{W}'_A(e_1) \\ \langle \tau_2, \theta_2 \rangle = \mathcal{W}'_{\theta_1 A}(e_2) \\ \alpha \text{ is a new type variable} \\ \theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \rightarrow \alpha) \end{cases} \\
\mathcal{W}'_A(\mathbf{let } x = e_1 \mathbf{ in } e_2) &= \langle \tau_2, \theta_2 \circ \theta_1 \rangle \\
&\text{where } \begin{cases} \langle \tau_1, \theta_1 \rangle = \mathcal{W}'_A(e_1) \\ \sigma = \forall_{\theta_1 A}(\tau_1) \\ \langle \tau_2, \theta_2 \rangle = \mathcal{W}'_{\theta_1 A, x:\sigma}(e_2) \end{cases} \\
\mathcal{W}'_A(\mathbf{rec } x \Rightarrow e) &= \langle \theta'\tau, \theta' \circ \theta \rangle \\
&\text{where } \begin{cases} \alpha \text{ is a new type variable} \\ \langle \tau, \theta \rangle = \mathcal{W}'_{A,x:\alpha}(e) \\ \theta' = \mathcal{U}(\theta\alpha, \tau) \end{cases}
\end{aligned}$$

Figure 2: Type-checking algorithm \mathcal{W}

Theorem 1 (Correctness of \mathcal{U})

1. $\mathcal{U}(\tau_1, \tau_2)$ always terminates.
2. If $\mathcal{U}(\tau_1, \tau_2) = \theta$ then $\theta(\tau_1) = \theta(\tau_2)$.
3. Furthermore, for any θ' such that $\theta'(\tau_1) = \theta'(\tau_2)$, there exists some θ'' such that $\theta' = \theta'' \circ \theta$.

Theorem 2 (Correctness of \mathcal{W})

1. $\mathcal{W}'_A(e)$ always terminates.
2. If $\mathcal{W}'_A(e) = \langle \tau, \theta \rangle$ then $\theta A \vdash e : \tau$.
3. Suppose $\theta A' \vdash e : \sigma$ for some $\theta, A' \leq A, e$ and σ . Then $\mathcal{W}'_A(e) = \langle \tau, \theta' \rangle$ for some τ, θ', θ'' such that $\theta(\alpha) = (\theta'' \circ \theta')(\alpha)$ for all $\alpha \in FV(A)$ and $\sigma \sqsubseteq \theta''(\forall_{\theta' A}(\tau))$.

Practical implementations of unification do not pass around explicit substitutions. Instead variables are represented as updateable references directly in the terms. Substitution is implemented by assigning pointers to subtrees to these references, thus leading to an *acyclic digraph* term representation (acyclic because

$$\begin{aligned}
\mathcal{U}(\tau, \tau) &= \{\} \\
\mathcal{U}(\alpha, \beta) &= \{\alpha/\beta\} \text{ where } \alpha \neq \beta \\
\mathcal{U}(\alpha, \mathfrak{t}(\overline{\tau_m})) &= \begin{cases} \{\mathfrak{t}(\overline{\tau_m})/\alpha\} & \text{if } \alpha \notin FV(\tau_i) \text{ for all } i \in [m] \\ - & \text{otherwise} \end{cases} \\
\mathcal{U}(\mathfrak{t}(\overline{\tau_m}), \alpha) &= \begin{cases} \{\mathfrak{t}(\overline{\tau_m})/\alpha\} & \text{if } \alpha \notin FV(\tau_i) \text{ for all } i \in [m] \\ - & \text{otherwise} \end{cases} \\
\mathcal{U}(\mathfrak{t}(\overline{\tau_{1,m}}), \mathfrak{t}(\overline{\tau_{2,m}})) &= \theta_m \circ \dots \circ \theta_1 \\
\text{where } \theta_i &= \mathcal{U}((\theta_{i-1} \circ \dots \circ \theta_1)(\tau_{1,i}), (\theta_{i-1} \circ \dots \circ \theta_1)(\tau_{2,i})) \\
&\quad \text{for all } i = 1, \dots, m \\
\mathcal{U}(\mathfrak{t}_1(\overline{\tau_{1,m_1}}), \mathfrak{t}_2(\overline{\tau_{2,m_2}})) &= - \text{ if } \mathfrak{t}_1 \neq \mathfrak{t}_2
\end{aligned}$$

Figure 3: Unification algorithm \mathcal{U}

of the occurs check in unification). Appendix A gives a declaration for such a representation in ML, and an implementation in ML of the unification algorithm. Now unifying two different variables leads to one reference being aliased to another, so that each call to `Unify` begins by performing any necessary dereferencing of an alias chain. For efficiency this dereferencing operation also performs path compression on the alias chain [1, 33].

The choice between applicative and destructive substitutions is not one to be taken casually. Sansom [45] provides some data on the use of applicative substitutions, based on profiling of the Glasgow Haskell compiler [40]:

45% of [the compiler's] time [was] spent in the typechecker. . . [N]early 36% of the entire compilation time [was] spent extending the substitution (a routine consisting of only 30 lines of code), with an additional 5% of the execution time spent searching the association list for a type variable's substitution.

Based on this experience, the applicative implementation of substitution in the Glasgow Haskell compiler has been replaced by a destructive implementation, as described in App. A. The Glasgow Haskell implementation is based on the use of monadic mutable arrays [41]:

The results . . . show quite spectacular speedups. Making the substitution representation non-idempotent improved the performance of the substitution algorithm by a factor of 5. . . [T]he introduction of a mutable array as the underlying data structure provided a further 10 times speedup. Overall the performance of the substitution algorithm was improved by a factor of more than 50!

Sansom further notes that the time spent in type-checking dropped from 45% of compilation time down to 10%, with overall compilation time reduced by 75% (by these and other optimizations to the compiler data structures).

The algorithm in App. A, is the algorithm used in GHC and almost all other ML and Haskell compilers; we refer to this in Sect. 7 as *naive graph unification*. Our approach to type explanation is based on modifying this algorithm. Section 3 provides the basis of our approach to type explanation. With this approach, the

naive graph unification algorithm is modified to record the sequence of steps in type-checking which lead to a type variable being instantiated. In Sect. 4 we consider the very practical consideration of how to properly handle this type explanation information with variable aliasing. This section demonstrates that simply augmenting the result of unification with type explanations is insufficient, the algorithm itself must be modified to preserve the correctness of explanations. Path compression, the most important operation used in practical unification algorithms to improve efficiency with variable aliasing, must be carefully restricted in order to preserve the correctness of type explanations. Section 5 considers how to properly handle aliasing between type variables which are introduced by the instantiation of polymorphic types. This consideration is absolutely crucial to the practical usability of any type explanation facility, as explained that section. In Sect. 6 we provide a formalization of type explanation which is independent of the particular unification algorithm used. In Sect. 7 we use this formal framework to examine how our algorithm could be incorporated into asymptotically superior algorithms. Essentially the results of Sect. 3 and Sect. 5 are relevant to all of these algorithms, while the results of Sect. 4 are relevant to all of the algorithms whose use in type inference is remotely practical.

3 Type Explanation

In this section we describe our approach to type explanation. We denote generic type variables by 'a, 'b, 'c, ... and non-generic type variables by 'A0, 'A1, ..., 'B0, 'B1, The type inference algorithm \mathcal{W} itself is essentially unchanged, since this algorithm simply walks over the abstract syntax tree for a program collecting equality constraints on the types of the program variables. The true heart of the algorithm is in the unification algorithm \mathcal{U} which is used to solve these constraints (aborting if no solution is possible).

Initially a program variable x is given type 'A0 for some new (non-generic) type variable 'A0. \mathcal{W} collects constraints on 'A0 due to the usage of x in the program, and \mathcal{U} solves these constraints, possibly instantiating 'A0 via substitution in the process. "Instantiation" in the algorithm of App. A means assigning to an unbound type variable. To record information for type explanation, we record the program fragment which gave rise to that constraint, with the instantiation. For example, for the program fragment:

```
fn x => x + 1
```

x is initially assumed to have type 'A0. Type checking the application $+(x, 1)$ constrains ('A0 * int) to be equal to (int * int) (the domain type of +). Unification produces the substitution {int/'A0}, which in practical terms means the reference cell for 'A0 is assigned int. Assuming the type checker actually annotates the abstract syntax tree with type information as it walks over it, the modified type inference algorithm stores the abstract syntax for $+(x, 1)$ as an explanation for why the type of x is int, with the type annotation for x .

The following example (repeated from Section 1) shows why this alone is insufficient:

```
... F y; ... y=(3,x); ... F (z,4.5)
```

Using the simple approach just outlined, we get the explanation:

```
x : real
F (z,4.5) ..... gives x : real
```

Figure 4 shows the graph for the type of **F** created by type inference. Vertices in this graphical representation are labelled by type expressions. directed edge from a type variable vertex to another vertex represents the instantiation of the type variable, labelled by the program fragment which gave rise to the constraint which caused that instantiation. Since such an edge may be directed from or to a subexpression of the type expression labelling a vertex, we distinguish this subexpression graphically by a subvertex embedded in the original vertex. In the example in Fig. 4, there is an instantiation edge from 'A4 to **int**, both of which are subexpressions of compound type expressions. The label on this edge records that the instantiation was caused by the unification triggered by the type-checking of the application **F (z,4.5)**.

In this figure, the dashed edge represents the instantiation of the type of **x** (the type variable 'A3) to **real**. The program fragment which gives rise to this instantiation causes the function type at the bottom of the diagram to be unified with the type of **F**. The path in this type graph from 'A0 to 'A3 follows edges which connect a component of **F**'s type to the type of **x**; the label on each one of these edges records the program fragment which created that link in the path of connections. The unification algorithm traces this connection path in the process of walking over the two type graphs. The modified unification algorithm accumulates these explanations in the process of recursively walking over the graph, and when it performs an instantiation at the end of such a connection path, it records this accumulated explanation with the program fragment which led to unification being called.

Using this information, the explanation for the type of **x** is:

```
x : real
F y ..... gives F : 'A0 = 'A1 -> 'A2
= (y,(3,x)) .... gives y : 'A1 = int * 'A3
F (z,4.5) ..... gives x : 'A3 = real
```

On the other hand, the explanation for **z**'s type is:

```
z : int
F (z,4.5) ..... gives z : 'A4 = int
```

Whereas the explanation for **x** needs to explain the connection between **x**'s type and that of **F**, in the case of **z** the connection is immediate. Further explanation of the type of **F** is obtained by lazily expanding the frontier of the type digraph rooted at 'A0, printing an explanation with each expansion of an interior type variable to its instantiation:

```
F : 'A1 -> 'A2
F y ..... gives F : 'A0 = 'A1 -> 'A2

F : (int * 'A3) -> 'A2
= (y,(3,x)) .... gives y : 'A1 = int * 'A3

F : (int * real) -> 'A2
F (z,4.5) ..... gives x : 'A3 = real
```

In the final phase of this explanation, it is necessary for the explainer to recognize that part of the explanation on the edge rooted at 'A3 has already been encountered and does not need to be repeated. As another example, the type for **y** is explained as:

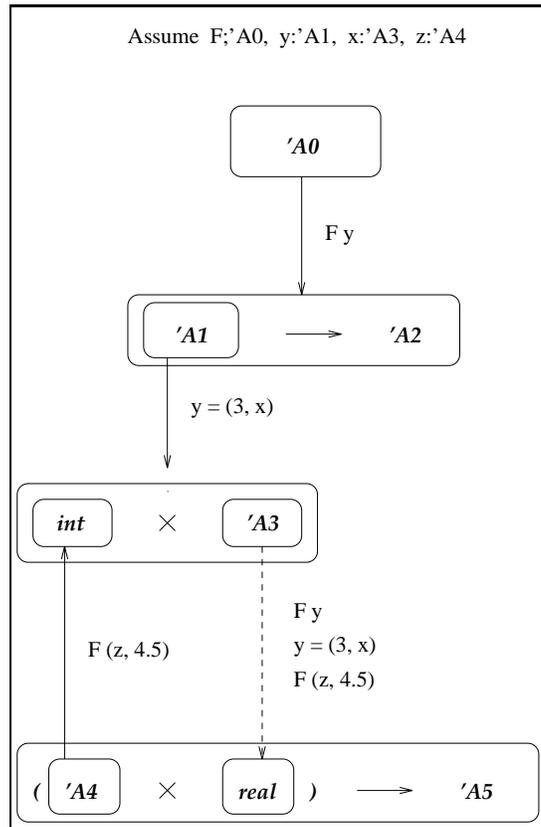


Figure 4: Type of F after: F y; y = (3,x); F (z,4.5)

```
y : int * 'A3
= (y,(3,x)) .... gives y : 'A1 = int * 'A3
```

```
y : int * real
F y ..... gives F : 'A0 = (int * 'A3) -> 'A2
F (z,4.5) ..... gives x : 'A3 = real
```

In the next example the type is a graph but not a tree. Figure 5 shows the type graph for the types of F and G after the following program fragment has been type checked:

```
F y; y=(w,x); F(v,4.5); G y; G (3,u)
```

For example, the explanation for y's type is:

```
y : 'A5 * 'A6
= (y,(w,x)) .... gives y : 'A1 = 'A5 * 'A6
```

```
y : int * 'A6
G y ..... gives G : 'A3 = ('A5 * 'A6) -> 'A4
```

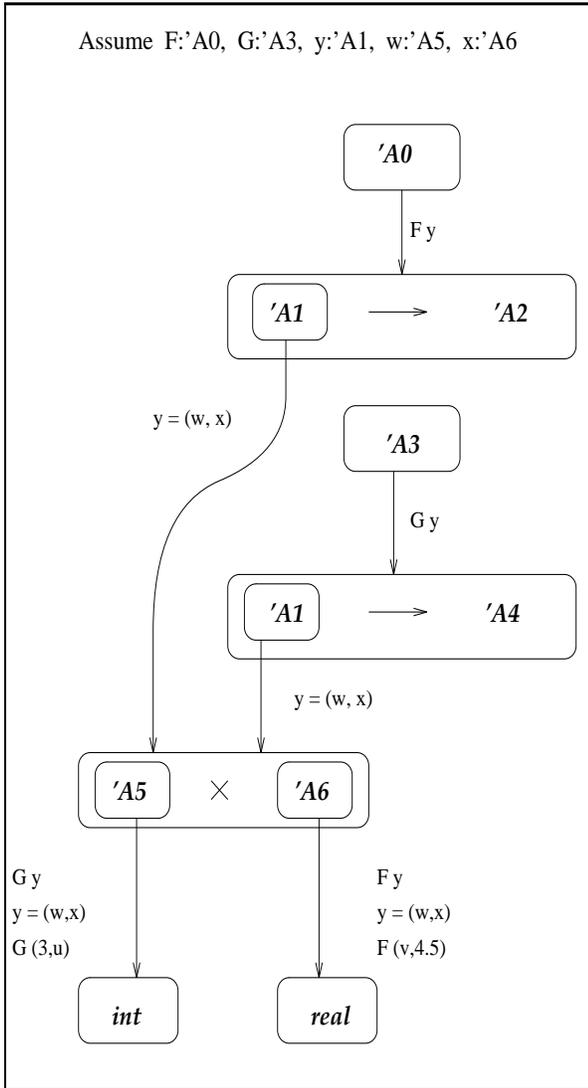


Figure 5: Type of F and G after: $F\ y; y=(w,x); F(v,4.5); G\ y; G(3,u)$

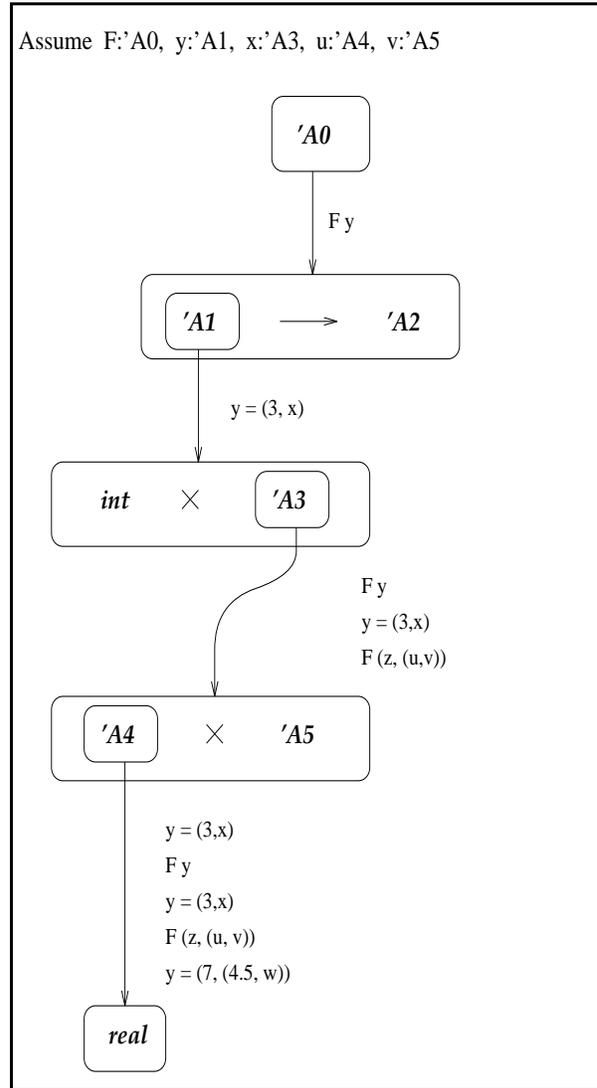


Figure 6: Type of F after: $F\ y; y=(3,x); F(z,(u,v)); y = (7,(4.5,w))$

$G(3,u) \dots \dots \dots$ gives $w : 'A5 = int$

$y : int * real$

$F\ y \dots \dots \dots$ gives $F : 'A0 = ('A5 * 'A6) \rightarrow 'A2$

$F(z,4.5) \dots \dots \dots$ gives $x : 'A6 = real$

As another example, the explanation for F 's type is:

$F : 'A1 \rightarrow 'A2$

$F\ y \dots \dots \dots$ gives $F : 'A0 = 'A1 \rightarrow 'A2$

```

F : ('A5 * 'A6) -> 'A2
= (y,(w,x)) .... gives y : 'A1 = 'A5 * 'A6

F : (int * 'A6) -> 'A2
G y ..... gives G : 'A3 = ('A5 * 'A6) -> 'A4
G (3,u) ..... gives w : 'A5 = int

F : (int * real) -> 'A2
F (z,4.5) ..... gives x : 'A6 = real

```

The next example (Figure 6) demonstrates where a compound explanation may occur on an edge besides the last edge in a path:

```
F y; y = (3,x); F (z,(u,v)); y = (7,(4.5,w))
```

In this case the explanation for the type of `u` is (again the explainer needs to remember instantiations for type variables which have already been explained):

```

u : real
= (y,(3,x)) .... gives y : 'A1 = int * 'A3
F y ..... gives F : 'A0 = (int * 'A3) -> 'A2
F (z,(u,v)) .... gives x : 'A3 = 'A4 * 'A5
= (y,(7,(4.5,w))) gives u : 'A4 = real

```

All of the examples so far have been based on unification being called in the type checking of applications. This approach extends straightforwardly to unification being called in the type checking of recursive function definitions. For example:

```
fun f x y = if true then "hello" else (f 3 y; y)
```

translates into:

```
val rec f = fn x => fn y => if true then "hello" else (f 3 y; y)
```

which in the syntax of our mini-language is:

```

rec f => fn x => fn y =>
  case true of true => "hello" | false => (f 3 y; y)

```

Figure 7 shows the type graph created by type inference; here the type of the formal parameter `x` is reconstructed from the type of `f` (rather than the other way round) based on the recursive use of `f`.

Pattern-matching introduces the complications of internally-generated program variables, and needing to record which pattern in a `case` construct contributed to a type variable being instantiated. For example:

```

fun map f [] = []
| map f (x::xs) = (f x)::(map f xs)

```

is translated into the equivalent declaration:

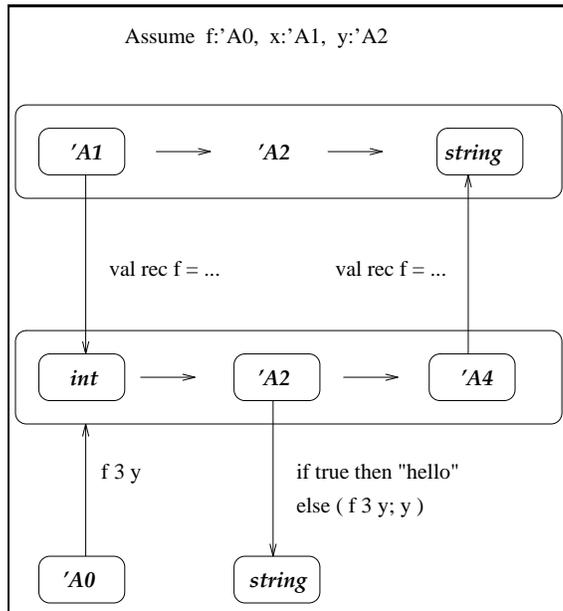


Figure 7: Type of `f` and `x` after: `fun f x y = if true then "hello" else (f 3 y; y)`

```
val rec map = fn f' => fn l' => case (f',l') of
  (f, []) => [] |
  (f,(x::xs)) => (f x)::(map f xs)
```

In this case the type explanation records that the first pattern in the `case` constrains the type of `l'` to be a list type (caused by unifying the type of `(f',l')` with the type of `(f, [])`).

Note that type information may flow either way between the case argument and the pattern:

```
datatype 'a foo = foo of 'a
....
case foo x of foo(u,v) => ...
....
case foo(x,y) of foo u => ...
```

4 Aliasing of Type Variables

Aliasing introduces complications which require the most radical modifications to the traditional type inference algorithm. Consider for example two instantiation edges `('A0,int)` and `('A1,'A2)`. If the normal unification algorithm is called to unify `'A0` with `'A1`, the algorithm first dereferences these to `int` and `'A2`, respectively, and then instantiates `'A2` to `int`. This is shown in Part (a) of the following graphic:



In terms of type explanation this is clearly a mistake. For example, consider an explanation of why program variable `x`, with initial type `'A2`, is instantiated to `int`. This explanation does not even mention the two intervening type variables (which might represent the undetermined types of other program variables) which caused this unification. In our approach we perform the instantiation at the *root* of an aliasing path rather than at the leaf; furthermore we reverse the direction of the edges on the aliasing path which was rooted at that type variable. In the example above, `'A1` is reinstated to `'A0`, and `'A2` is instantiated to `'A1`. This is shown in Part (b) of the graphic. Thus an explanation for why `'A2` is bound to `int` includes an explanation of the instantiation edges (`'A2, 'A1`), (`'A1, 'A0`) and (`'A0, int`).

This approach to instantiating variables has important (and practical) implications for the type-checking algorithm. For one thing, path compression cannot be performed on aliasing paths terminating in unbound type variables; consider the following example:

```
x=y; y=z; z=w; y="ha"
```

Figure 8(a) illustrates the aliasing path created by the first three equality operations. Note that path compression during variable dereferencing is no longer a valid operation. For example, Fig. 8(b) depicts the situation where path compression has redirected `'A0` and `'A1` to `'A3`. Here path compression has also involved concatenating explanations on compressed instantiation edges. `y="ha"` causes `'A3` to be instantiated to `string`. But now the explanation for `z`'s type (rooted at `'A2`) is:

```
z : string
= (z,w) ..... gives z : 'A2 = 'A3
= (y,z) ..... gives y : 'A1 = 'A2
= (y,"ha") ... gives w : 'A3 = string
```

which appears strange (why does `w` appear in the explanation at all?). So the unification algorithm modified for type explanation does not perform path compression. Figure 8(c) depicts what actually happens with our algorithm when `y="ha"` causes the type variables `'A1`, `'A2` and `'A3` to be instantiated to `string`. Every edge on the alias path rooted at `'A1` (the type of `y`) is reversed, and `'A1` is reassigned to point to `string`. Now the explanation for the type of `z` is:

```
z : string
= (y,z) ..... gives z : 'A2 = 'A1
= (y,"ha") ... gives y : 'A1 = string
```

The next example is a slightly more developed example of aliasing:

```
x=y; y=z; y=w; y=u; z="ha"
```

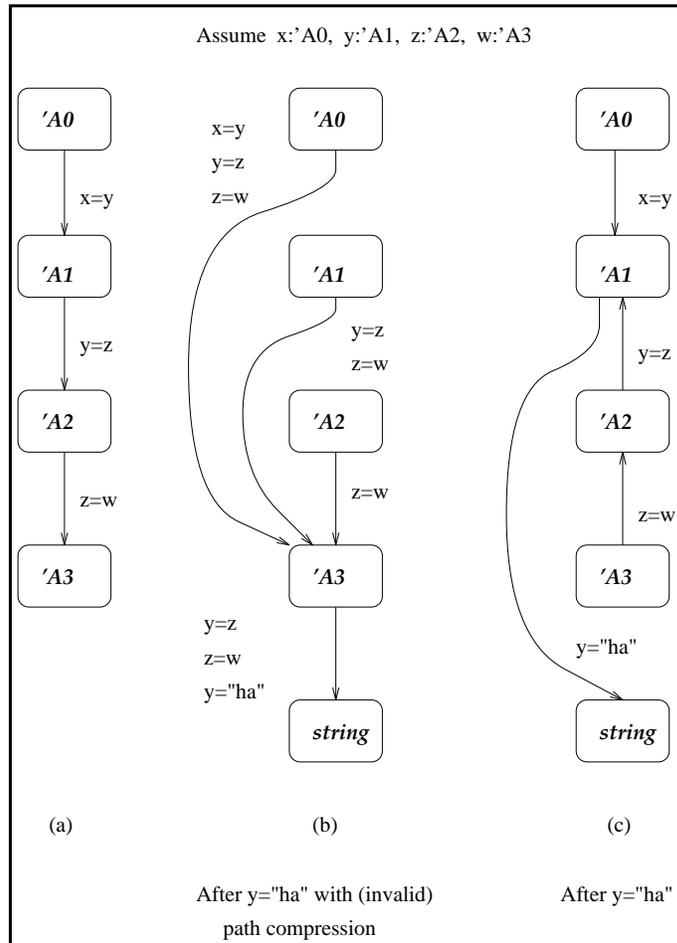


Figure 8: Example of Aliasing: $x=y; y=z; z=w; y="ha"$

Figure 9(a)-(d) depicts the changing type graph as four type variables are aliased to 'A1 (the type of y) and eventually instantiated to `string`.

A final issue to consider is when aliasing is caused by the parallel traversal of two compound type graphs. For example:

`F x; F y; y=z; G u; G v; v=w; if true then F else G`

Figure 10(a) depicts the type graph before the conditional is type checked. In particular the applications of F cause x and y 's type variables to be aliased, while the applications of G cause u and v 's type variables to be aliased. Type checking the conditional now requires the types of F and G to unify, resulting in Fig. 10(b) where the types of x and u are aliased. Recall from Fig. 4 that the explanations for the instantiations of 'A3 and 'A4 were different, reflecting that they were located by the unification algorithm by following different instantiation paths. We can consider aliasing two variables as instantiating them both to a new third variable. Therefore in contrast to Fig. 4, in Fig. 10(b) the aliasing edge between 'A1 and 'A6 contains the location

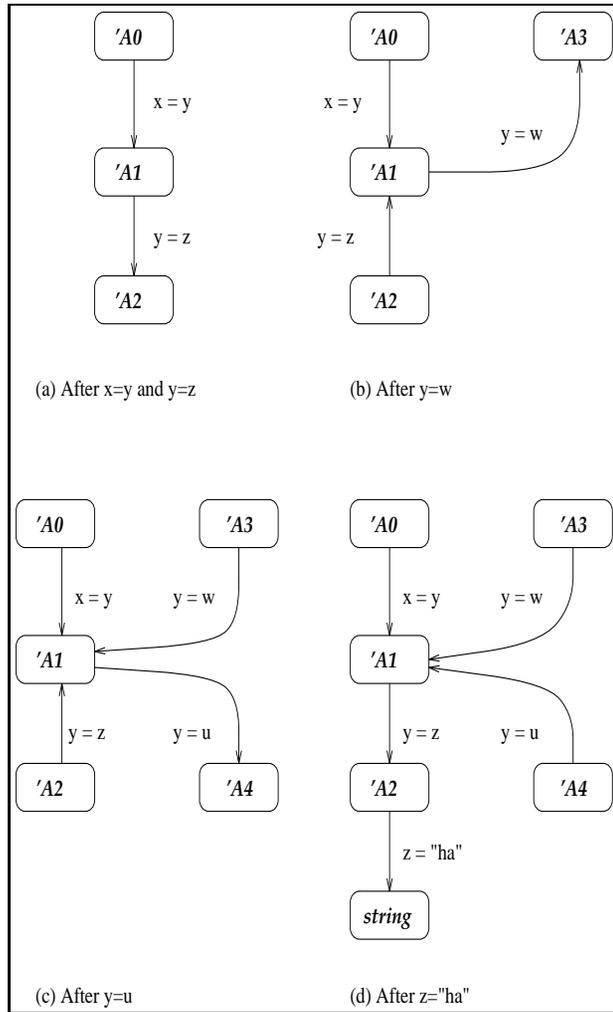


Figure 9: Example of Aliasing:
 $x=y$; $y=z$; $y=w$; $y=u$; $z="ha"$

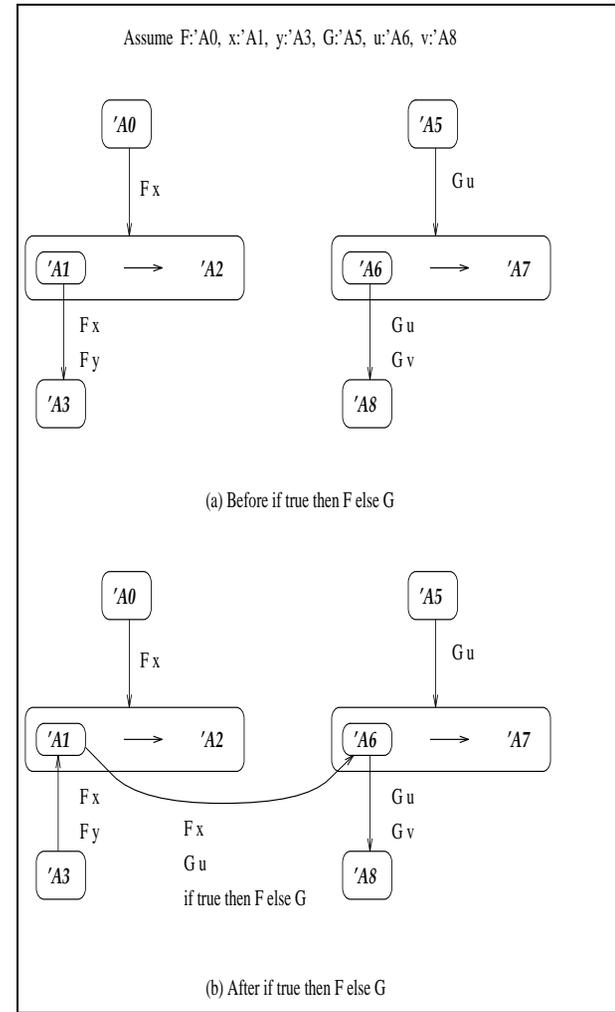


Figure 10: Example of Aliasing:
 Fx ; Fy ; $y=z$; Gu ; Gv ; $v=w$;
 if true then F else G

information for both type variables³.

Appendix B gives the unification algorithm modified to collect information for type explanation. Note that the reversing of aliasing edges is justified by the fact that the location information for both type variables involved is included in the explanation for the edge, and by the fact that the only time the algorithm combines explanations on edges into location information is when those edges lead from type variable nodes to type constructor nodes (the only place in the algorithm where `addHist` is called is in the final case).

This unification algorithm does not tell the whole story, since there is no notion in the algorithm of filtering out duplicate explanations which have been already encountered. As in the explanations given earlier, this filtering process is left to the type explainer. This is a fairly routine exercise of checking for duplicates (using reference equality in ML to check for duplicates) while printing an explanation labelling a path in a type digraph.

Although Fig. 8(b) demonstrates why path compression should not be performed before an aliased variable is bound, this is not an objection once the variable is bound. For example it would be valid in Fig. 8(c) to compress the paths from `'A0`, `'A2` and `'A3` since the edges on these paths will no longer be reversed once `'A1` is instantiated. A suitable place in the algorithm to do this is in the call to `addHist`, which as we have just noted is only called when the algorithm is applied to two types which are, or are aliased to, non-variable types. The full algorithm, presented in the next section, contains this optimization.

5 Polymorphic Aliasing

Although the algorithm as provided is correct, pragmatically we need to consider other details in order to provide a usable output. Consider an example as simple as `x=y`, where `x` and `y` have types `'A0` and `'A1` respectively, and `=` has polymorphic type `'a * 'a -> bool` (ignoring the fact that `'a` should really be an equality type variable). Although in the previous section we would have aliased `'A0` to `'A1`, in reality we would instantiate `'a` with a new non-generic type variable `'A2`, with `'A0` aliased to `'A2` which is then aliased to `'A1`. In this section we consider an approach to providing type explanations which are not unnecessarily cluttered with such intermediate type variables.

We use `'B0`, `'B1`, ... to represent non-generic type variables which are introduced to instantiate a polymorphic type at the use site of a `let`-bound variable. We will refer to these `'B`'s as *let-bound type variables*, while we will refer to the `'A`'s as *λ -bound type variables*. Then we define:

A *polymorphic alias path* is a sequence of labelled directed edges $\tau \xrightarrow{e_0} 'B_1, 'B_i \xrightarrow{e_i} 'B_{i+1}$ for $i = 1, \dots, n-1$, some $n \geq 1$, and $'B_n \xrightarrow{e_n} \tau'$, where each e_i is an explanation, and τ and τ' are any type expressions.

The essential idea is that, whenever we have a polymorphic alias path $\tau \xrightarrow{e_1} 'B_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} 'B_n \xrightarrow{e_{n+1}} \tau'$, we use the explanation e_{n+1} labelling the last edge in this path as the complete explanation for the relationship between τ and τ' . The type explainer prints an explanation for a polymorphic alias path precisely once while traversing that path, using as explanation that labelling the last edge on the path. Essentially a polymorphic alias path is treated as a single aliasing edge.

To see why this is sufficient, consider the following example:

$$'A0 \xleftarrow{e_1} 'B0 \xrightarrow{e_2} 'B1 \xrightarrow{e_3} 'B2 \xrightarrow{e_4} 'A1$$

³We omit the aliasing edge between `'A2` and `'A7` for reasons of space and clarity.

where the type variables 'B0 and 'B1 are being unified. If we choose to direct the aliasing edge from 'B1 to 'B0, then the algorithm updates the explanation on the edge between 'B0 and 'A0:

$$'A0 \xleftarrow{e_2} 'B0 \stackrel{e_2}{=} 'B1 \xrightarrow{e_3} 'B2 \xrightarrow{e_4} 'A1$$

The justification for this is that we are building a polymorphic alias path. Since the path only involves let-bound type variables, this must arise from applying a let-bound program variable to a let-bound program variable. Since applications are type checked “bottom-up,” the addition of an aliasing edge to this path must be labelled with an explanation term which is a superexpression of the explanations labelling the existing edges on the path. Relabelling the edges on the alias path ensures that the explanation labelling the last edge is maximal. Note that for this reasoning to be valid, we must ensure that whenever an uninstantiated let-bound type variable is unified with a type which is not also a let-bound variable, the instantiating edge is directed from the let-bound type variable. In particular we must ensure that there is never an aliasing edge from a λ -bound variable to an unbound let-bound variable.

In addition to updating the edges on the polymorphic alias path rooted at 'B0, it is also necessary to update the edges on the alias path rooted at 'B1, before or during path reversal:

$$'A0 \xleftarrow{e_2} 'B0 \stackrel{e_2}{=} 'B1 \xleftarrow{e_2} 'B2 \xleftarrow{e_2} 'A1$$

To understand why this is necessary, consider if subsequently 'A1 were instantiated, requiring the alias path rooted at 'A1 to be reversed. In the absence of the updating of the explanations on the polymorphic alias path rooted at 'A1, the result would be a polymorphic alias path rooted at 'A0 with concluding edge labelled with explanation e_4 , giving an incomplete explanation for the aliasing of 'A0 and 'A1.

It might be considered sufficient to update the *polymorphic* alias path rooted at 'B1 in the example above, rather than walking over the alias path rooted at 'B1 and updating each polymorphic alias path contained in that alias path. The example in Fig. 11 illustrates why this is insufficient. In Fig. 11(b), the aliasing of 'B0 to 'A2 causes the edge between 'B0 and 'A1 to be reversed; assume for the sake of argument that the explanation on this reversed edge is also updated in the process of instantiation. In Fig. 11(c), the aliasing of 'A3 to 'B0 causes the explanation on the ('B0, 'A2) edge to be updated. Note that although the explanation on the ('A1, 'B0) is still e_3 , the explanation for the polymorphic alias path rooted at 'A1 is taken to be e_4 . Then in Fig. 11(d), the instantiation of 'A0 leaves the wrong explanation on the last edge in the polymorphic alias paths rooted at 'A2 and 'A3 (the ('B0, 'A1) edge should be updated with explanation e_4). So when reversing alias paths, we must be sure to update the explanation on the last edge of any reversed polymorphic alias paths which are subpaths of the alias path. The **ReverseAliasPath** procedure discussed below does this.

Table 1 considers the various possible cases. For example, in Case 2, 'B0 is equated to the λ -bound type variable 'A0 as a result of type-checking the expression e_2 . 'B0 is already aliased to 'B1 as a result of type-checking the expression e_1 . The instantiation edge is directed from 'B0 to 'A0, reversing any aliasing path rooted at 'B0 and updating the explanations on the alias path in the process with e_2 .

As a concrete example, consider:

```

fun curry f x y = f (x,y)
curry : (('a*'b)->'c) -> 'a -> 'b -> 'c
= : 'a * 'a -> bool
curry (op =) x y

```

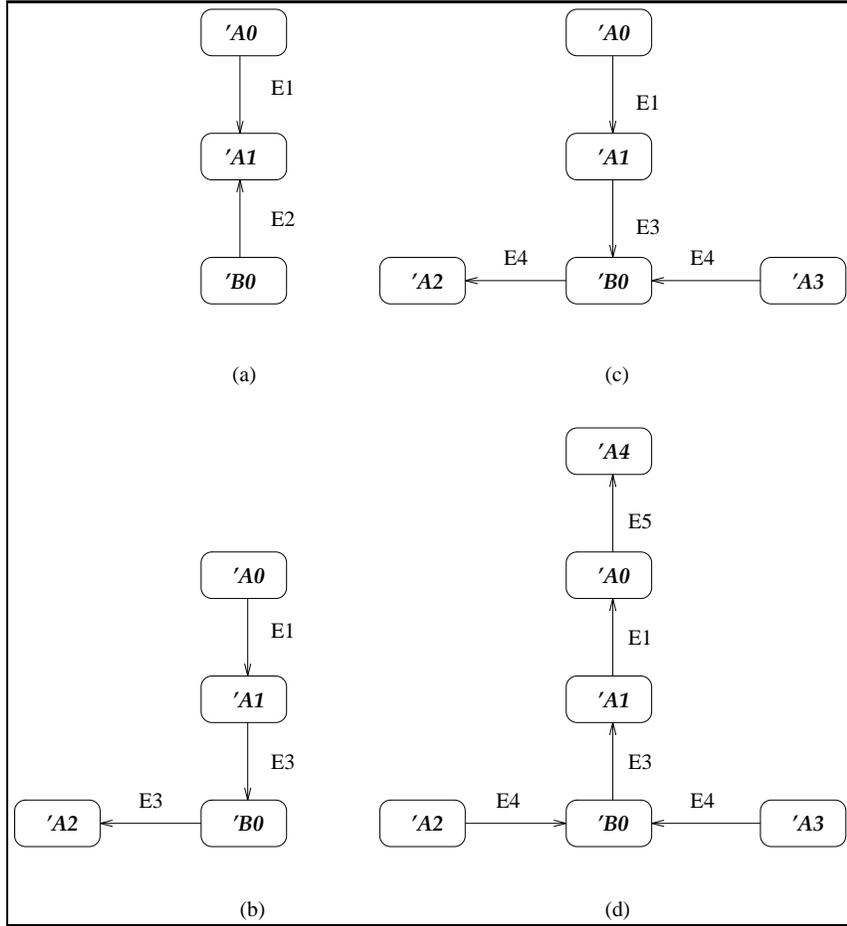


Figure 11: Example of Invalid Polymorphic Alias Path Updating

In this application of `curry`, let `'B0`, `'B1` and `'B2` be the type variables which instantiate `'a`, `'b` and `'c` in the type of `curry`, respectively, and `'B3` be the type variable which instantiates `'a` in the type of `=`. Figure 12(a) shows the type graph after type checking the application `(curry (op =))`, with both `'B0` and `'B1` aliased to `'B3`, and `'B2` instantiated to `bool`.

Figure 12(b) shows the type graph after type checking the application `((curry (op =)) x)`. In this case we follow the rule that whenever aliasing a let-bound type variable and a λ -bound type variable, we direct the aliasing edge from the let-bound variable (`'B0`) to the λ -bound variable (`'A0`). In so doing, we reverse the alias edge between `'B0` and `'B3`, updating the explanations labelling this edge in the process.

Figure 12(c) shows the type graph after type checking the application `((((curry (op =)) x) y)`. Once again, reversing the aliasing edges on the path between `'A0` and `'B1` involves updating the explanations on these edges.

Appendix C gives the complete unification algorithm supporting type explanation. Each step in the unification algorithm involves dereferencing two alias chains (possibly of zero length) rooted at the type vertices `ty1` and `ty2`. In the case where either or both is a variable, an instantiation must be performed. Assume for example that we want to assign `ty1` to `ty2`. We first reverse the alias path rooted at `ty2`, using

Table 1 Cases for when instantiating let-bound type variable.

1. *Let-bound aliased to let-bound, let-bound aliased to let-bound.* Then we have:

$$'B1 \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} 'B2 \xrightarrow{e_3} 'B3 \text{ becomes } 'B1 \xrightarrow{e_2} 'B0 \xrightarrow{e_2} 'B2 \xrightarrow{e_2} 'B3$$

2. *Let-bound aliased to let-bound, λ -bound.* Then we have:

$$'B1 \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} 'A0 \text{ becomes } 'B1 \xrightarrow{e_2} 'B0 \xrightarrow{e_2} 'A0$$

3. *Let-bound aliased to let-bound, constructor type.* Then we have:

$$'B1 \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} \text{tycon} \text{ becomes } 'B1 \xrightarrow{e_2} 'B0 \xrightarrow{e_2} \text{tycon}$$

4. *Let-bound aliased to let-bound, let-bound aliased to λ -bound.* Then we have:

$$'B1 \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} 'B2 \xrightarrow{e_3} 'A0 \text{ becomes } 'B1 \xrightarrow{e_2} 'B0 \xrightarrow{e_2} 'B2 \xrightarrow{e_2} 'A0$$

5. *Let-bound aliased to let-bound, let-bound aliased to constructor type.* Then we have:

$$'B1 \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} 'B2 \xrightarrow{e_3} \text{tycon} \text{ becomes } 'B1 \xrightarrow{e_2} 'B0 \xrightarrow{e_2} 'B2 \xrightarrow{e_2} \text{tycon}$$

6. *Let-bound aliased to λ -bound, λ -bound.*

$$'A0 \xleftarrow{e_1} 'A1 \xleftarrow{e_2} 'B0 \stackrel{e_3}{=} 'A2 \xrightarrow{e_4} 'A3 \text{ becomes} \\ 'A0 \xleftarrow{e_1} 'A1 \xleftarrow{e_3} 'B0 \xleftarrow{e_3} 'A2 \xleftarrow{e_4} 'A3$$

7. *Let-bound aliased to λ -bound, constructor type.*

$$'A0 \xleftarrow{e_1} 'A1 \xleftarrow{e_2} 'B0 \stackrel{e_3}{=} \text{tycon} \text{ becomes } 'A0 \xrightarrow{e_1} 'A1 \xrightarrow{e_3} 'B0 \xrightarrow{e_3} \text{tycon}$$

8. *Let-bound to λ -bound, let-bound to λ -bound.*

$$'A0 \xleftarrow{e_1} 'A1 \xleftarrow{e_2} 'B0 \stackrel{e_3}{=} 'B1 \xrightarrow{e_4} 'A2 \text{ becomes} \\ 'A0 \xrightarrow{e_1} 'A1 \xrightarrow{e_3} 'B0 \xrightarrow{e_3} 'B1 \xrightarrow{e_3} 'A2$$

9. *Let-bound to λ -bound, let-bound to constructor type.*

$$'A0 \xleftarrow{e_1} 'A1 \xleftarrow{e_2} 'B0 \stackrel{e_3}{=} 'B1 \xrightarrow{e_4} \text{tycon} \text{ becomes} \\ 'A0 \xrightarrow{e_1} 'A1 \xrightarrow{e_3} 'B0 \xrightarrow{e_3} 'B1 \xrightarrow{e_3} \text{tycon}$$

10. *Let-bound aliased to constructor type, λ -bound.*

$$\text{tycon} \xleftarrow{e_1} 'B0 \stackrel{e_2}{=} 'A0 \text{ becomes } \text{tycon} \xleftarrow{e_2} 'B0 \xleftarrow{e_2} 'A0$$

ReverseAliasPath. `ty1` may be the root of a polymorphic alias path; we therefore use `UpdatePolyAliasPath` to update the explanations on the edges of any such edge. Finally `ty2` is assigned

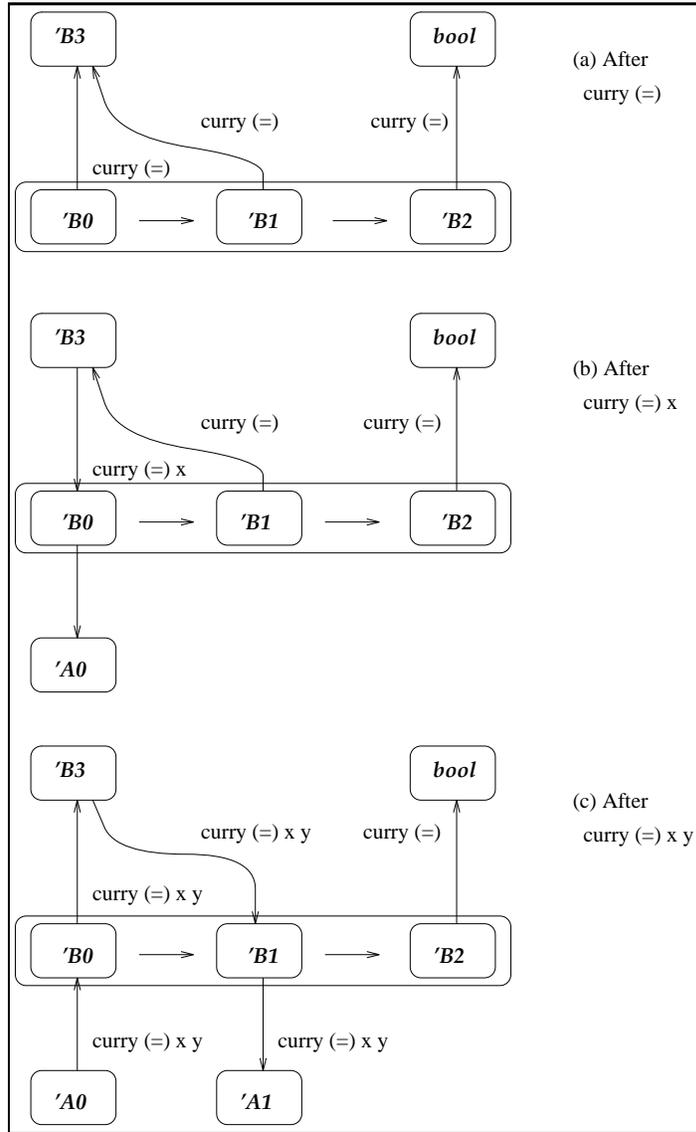


Figure 12: Example of Polymorphic Aliasing: Type of `curry (op =)` after type checking `curry (op =) x y`

a pointer to `ty1`. The following illustrates this example:

$$\begin{array}{ll}
 'A0 \xleftarrow{e_1} 'B0 \xrightarrow{e_2} 'B1 \xrightarrow{e_3} 'B2 \xrightarrow{e_4} 'A1 & \text{ty1 is 'B0, ty2 is 'B1} \\
 'A0 \xleftarrow{e_1} 'B0 \xrightarrow{e_2} 'B1 \xleftarrow{e_4} 'B2 \xleftarrow{e_4} 'A1 & \text{after ReverseAliasPath} \\
 'A0 \xleftarrow{e_2} 'B0 \xrightarrow{e_2} 'B1 \xleftarrow{e_4} 'B2 \xleftarrow{e_4} 'A1 & \text{after UpdatePolyAliasPath} \\
 'A0 \xleftarrow{e_2} 'B0 \xleftarrow{e_2} 'B1 \xleftarrow{e_4} 'B2 \xleftarrow{e_4} 'A1 & \text{after Bind}
 \end{array}$$

When both types `ty1` and `ty2` dereference to type constructor vertices, the `addHist` function traverses

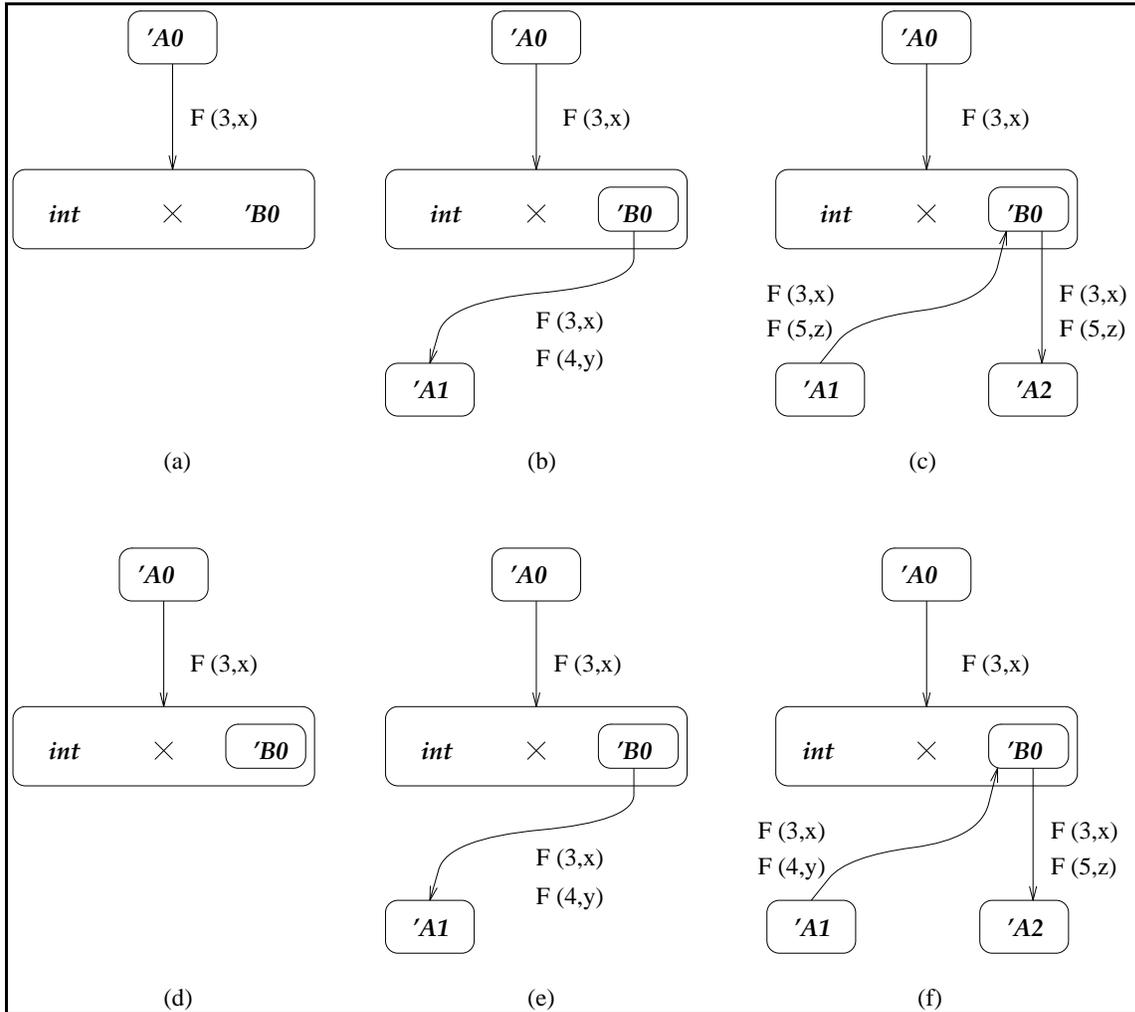


Figure 13: Example of Polymorphic Aliasing with Compound Explanations

the alias paths rooted at $\mathbf{ty1}$ and $\mathbf{ty2}$, in each case gathering the explanations in the path leading to the type constructor vertex. These explanations are added to the end of the list of explanations which has been gathered so far in reaching this vertex in the type tree being traversed by unification. `addHist` is also where the modified unification algorithm performs path compression. As the function walks down the alias path, it resets each type variable node encountered to point to the type constructor vertex at the end. In the process it must also reset the explanations for vertices with the accumulated explanations for the compressed paths.

Our discussion has so far avoided considering the possibility of compound explanations on edges in polymorphic alias paths. There are some non-trivial questions raised by this scenario: for example, when updating compound explanations on polymorphic alias edges, do we update the entire explanation or just the last explanation in that compound explanation? To motivate the answer to this, consider the example in Fig. 13, for the code fragment:

```
F ( 3 , x ) ; F ( 4 , y ) ; F ( 5 , z )
```

assuming the type declarations:

$$F: 'A0 \rightarrow 'A0 \rightarrow 'A0 \rightarrow 'A0, x: 'B0, y: 'A1, z: 'A2$$

Figure 13(a)-(b) demonstrate an invalid explanation graph built as a result of the construction described so far. After the application $F(3, x)$, the type variable $A0$ in F 's type is bound to the product $\text{int} * 'B0$, with a let-bound type variable embedded within it. The application $F(4, y)$ (Fig. 13(b)) aliases the type of x ($'B0$) to that of y ($'A1$). The explanation on this instantiation edge is a compound explanation containing both $F(3, x)$ and $F(4, y)$. Finally the application $F(5, z)$ causes $'B0$ to be aliased to $'A2$, with the previous instantiation edge rooted at $'B0$ reversed. Now since this edge is technically a polymorphic alias path, we must also update the explanation on this reversed edge. Doing this (as in Fig. 13(c)) leaves us in the (obviously wrong!) situation where the explanation for the aliasing of $'A1$ and $'A2$ does not even mention the program variable bound to $'A1$ (y).

The point demonstrated by this example is that, if a let-bound type variable becomes embedded in a type expression to which a λ -bound type variable is instantiated, then our earlier assumption that let-bound type variables are instantiated as a polymorphic application is type-checked “bottom-up” no longer holds. In this example, the let-bound type variable $'B0$ is instantiated twice, each time due to a completely separate application. In fact it is precisely to avoid this scenario that we earlier prohibited the direction of an aliasing edge from a λ -bound type variable to a let-bound type variable. The difficulty here is that we are obtaining a similar effect due to the embedding of a let-bound type variable in a compound type expression to which a λ -bound type variable must be bound. However there is an obvious way to detect this situation when instantiating a let-bound type variable: since that variable is reached by walking over a type graph rooted at a λ -bound type variable, it must be the case that the explanation labelling the instantiation edge is compound. So when we reverse an aliasing edge directed from a let-bound type variable, if that explanation is compound, then we do not perform any updating on the explanation on that edge. Furthermore the type explainer must print such an edge as a single explanation, rather than looking for the last edge in a polymorphic alias path. So we need to redefine what is meant by a polymorphic alias path:

A polymorphic alias path is a sequence of labelled directed edges $\tau \xrightarrow{e_0} 'B_1, 'B_i \xrightarrow{e_i} 'B_{i+1}$ for $i = 1, \dots, n-1$, some $n \geq 1$, and $'B_n \xrightarrow{e_n} \tau'$, where each e_i is an explanation *containing a single program fragment* (i.e. no compound explanations), and τ and τ' are any type expressions.

Fig. 13(c)-(d) demonstrates the correct explanation graph constructed using this reasoning.

We must finally consider the case when a compound type explanation must label an edge directed from a let-bound type variable, due to its being reachable from another let-bound type variable. Fig. 14(a)-(b) presents a scenario which considers the problems, for the code fragment:

$$F(3, x)(4, y)(5, z)$$

with the type declarations:

$$F: 'B0 \rightarrow 'B0 \rightarrow 'B0 \rightarrow 'B0, x: 'B1, y: 'A0, z: 'A1$$

In this case, F is a let-bound program variable and the polymorphic application is being type-checked bottom-up. Type-checking the application $F(3, x)$ instantiates $'B0$ with the product type $\text{int} * 'B1$. Continuing with the rest of the application, type-checking $F(3, x)(4, y)$ aliases $'B1$ to $'A0$. Finally the application $F(3, x)(4, y)(5, z)$ directs an aliasing edge from $'B1$ to $'A1$, reversing the direction of the aliasing

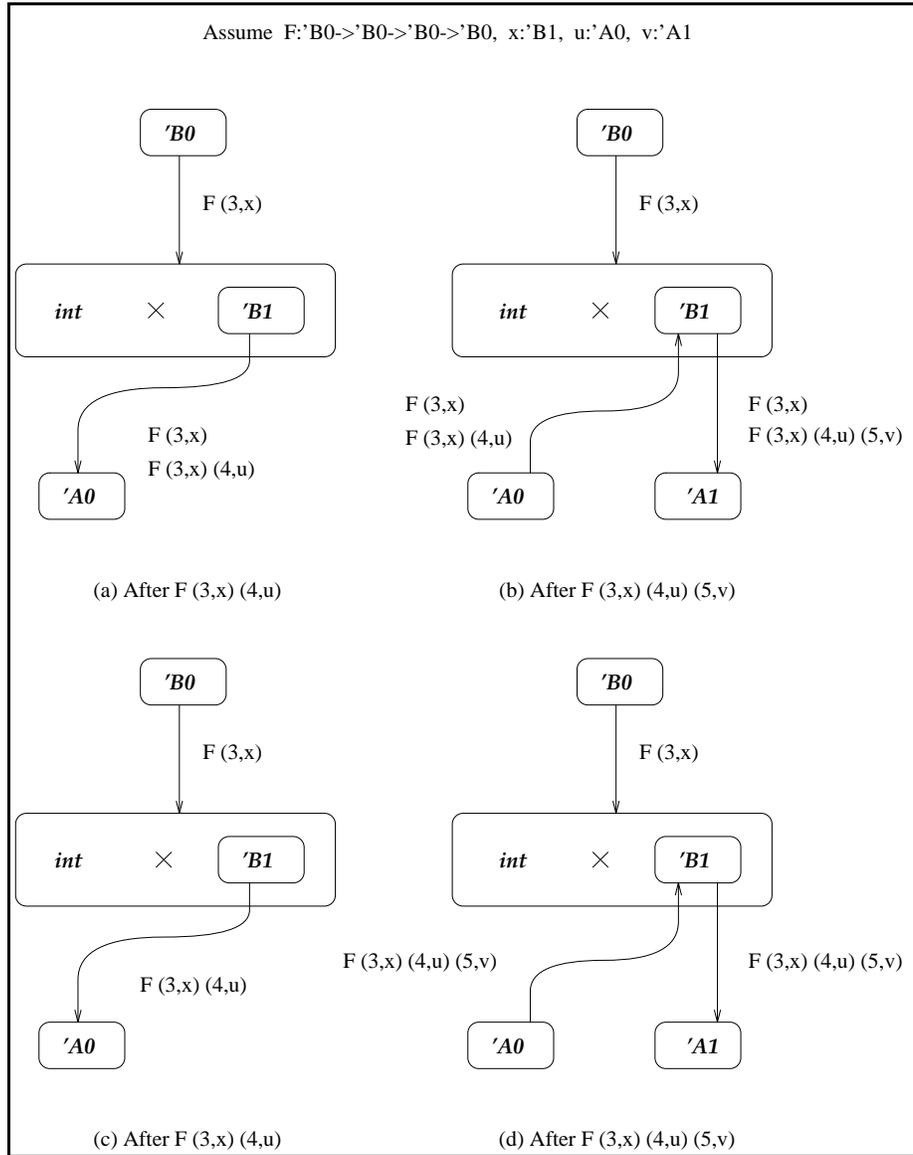


Figure 14: Example of Polymorphic Aliasing with Compound Explanations

edge between $'B1$ and $'A0$. According to the reasoning deduced from the previous example, we should not update the explanation on this edge while performing this reversing.

Although the resulting explanation graph is not technically wrong, it does have the unsatisfactory property that the explanation for the aliasing of $'A0$ and $'A1$ mentions three program fragments, $F(3, x)$, $F(3, x)(4, y)$, and $F(3, x)(4, y)(5, z)$, when only the last of these is really necessary. Preventing this form of redundancy was exactly the motivation for considering polymorphic alias paths in the first place. We refer to this as the problem of *deep polymorphic aliasing*.

There is a simple modification that avoids this problem. We note that the problem arises because a let-

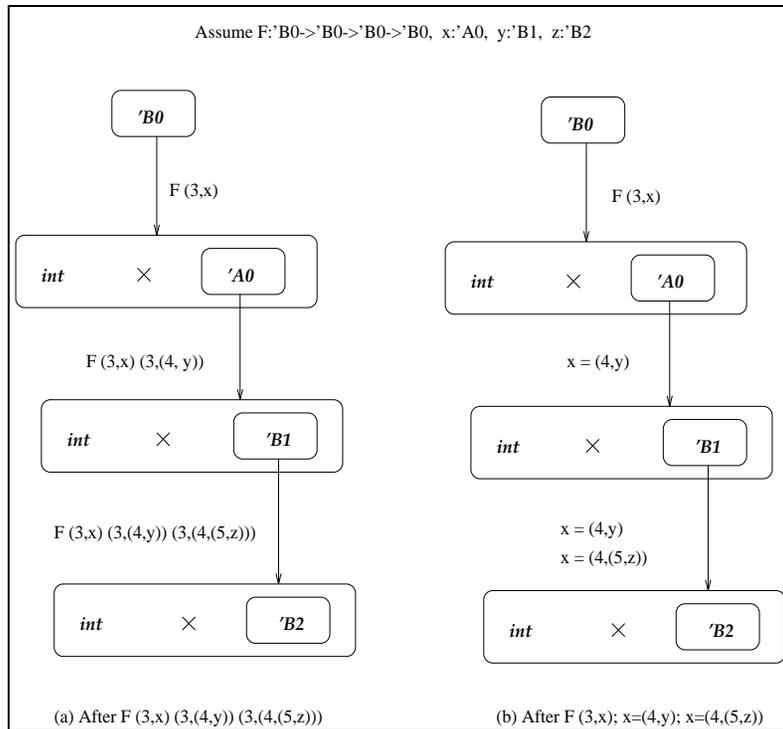


Figure 15: The effect of the entry variable type on edge explanations

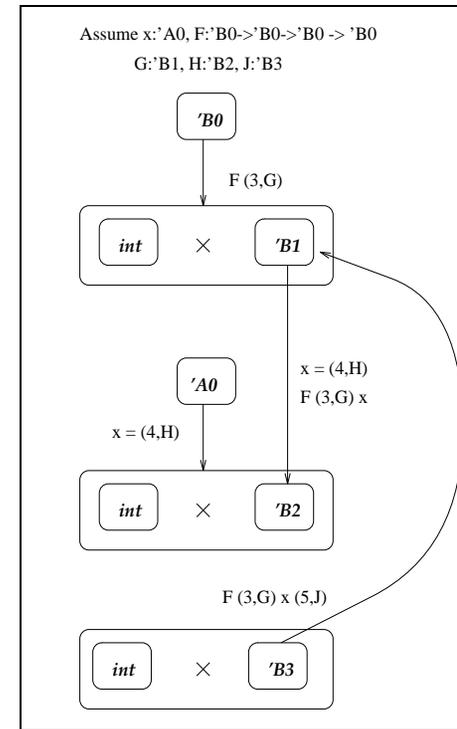


Figure 16: An alias path which is not a polymorphic alias path

bound type variable is instantiated in the process of traversing a type graph rooted at another let-bound type variable. In the example in Fig. 14, 'B1 is embedded in the type graph rooted at 'B0, and is instantiated as a result of unification being called on a type expression containing 'B0. Define a *polymorphic path* to be a sequence $\{\alpha_i \xrightarrow{e_i} \tau(\bar{\tau}^i) \mid i = 1, \dots, n \text{ and } \alpha_{i+1} \in FV(\tau(\bar{\tau}^i))\}$, where $\tau \xrightarrow{e} \tau'$ denotes a polymorphic alias path with common explanation e . Then we follow the rule that, when instantiating a type variable which is reached from a **let**-bound type variable along a polymorphic path, we only record the expression that caused the unification. In particular we omit the history list from the explanation; as a result, reversing a polymorphic alias path rooted at the embedded let-bound type variable ('B1 in the above example) will involve updating the explanation on the reversed alias edge. Figure 14(c)-(d) demonstrates this for the previous example, where the aliasing of 'A0 to 'A1 is explained by the single program fragment $F(3, x)(4, y)(5, z)$.

To understand this rule more fully, consider the following two program fragments:

```
F(3, x)(3, (4, y))(3, (4, (5, z)))
```

and

```
F(3, x); x = (4, y); x = (4, (5, z))
```

under the following type declarations:

```
F: 'B0->'B0->'B0->'B0, x: 'A0, y: 'B1, z: 'B2
```

Both program fragments give rise to the same type graphs. However the explanations aliasing the instantiation edges are very different, reflecting the fact that the entry points into the type graphs are different when type-checking the two program fragments. Fig. 15(a) gives the type graph rooted at 'B0 after type-checking the first of these code fragments. The unification algorithm is called three times, for the three applications $F(3, x)$, $F(3, x)(3, (4, y))$ and $F(3, x)(3, (4, y))(3, (4, (5, z)))$. The entry point in each case into the type graph is 'B0, so in each case only the current program fragment is used to label the instantiation edges which are added. Figure 15(b) gives the type graph rooted at 'B0 after type-checking the second of these code fragments. Although the type graphs are the same in both cases, the explanations on the instantiation edges are very different. In the second case, for the latter two calls to the unification algorithm, the entry point into the type graph is 'A0, so in both cases the instantiation edges which are added are labelled with compound type explanations explaining the path that was followed in the type graph to obtain that instantiation.

As another example of the implications of this approach, consider the code fragment:

```
x = (4, H); F(3, G)x(5, J)
```

under the type declarations:

```
x: 'A0, F: 'B0->'B0->'B0, G: 'B1, H: 'B2, J: 'B3
```

Figure 16 gives the type graph resulting from type-checking this code fragment. In particular the edges 'B3 \rightarrow 'B1 \rightarrow 'B2 do not form a polymorphic alias path, because of the compound explanation on the 'B3 \rightarrow 'B1 edge.

Figure 17 considers two final examples of deep polymorphic aliasing; these examples are useful in understanding the formal justification provided in the next section. Figure 17(a) illustrates that the explanation

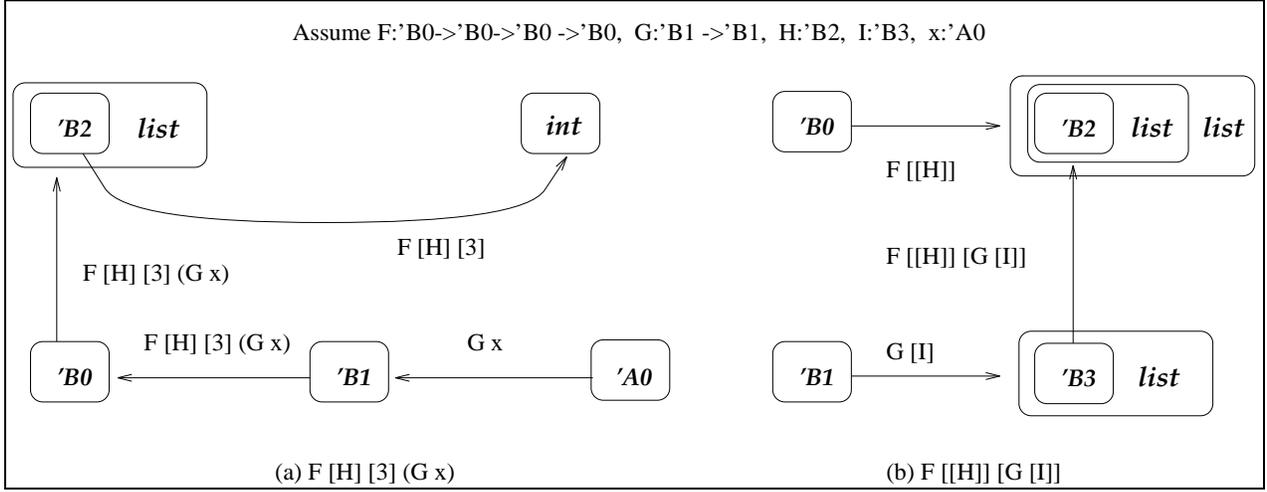


Figure 17: Examples of Polymorphic Aliasing with Compound Explanations

on an embedded polymorphic alias edges is not necessarily a superterm of the explanation on the polymorphic alias edge along which the embedding term was reaching. There may be further updating of the latter edge after the former edge has been inserted with its explanation. Figure 17(b) illustrates the case where an embedded polymorphic alias edge is inserted. In this case the explanation on the embedded edge must be an application which contains the two explanations as rator and rand. Note that further processing may cause the scope of the explanation on the embedded polymorphic alias edge to be expanded. The general case is formalized in the next section.

6 Formalizing Type Explanation

In this section we provide some formal justification for our approach to type explanation. We concentrate on the output of graph unification algorithms, since too much information is lost in the output of tree unification [44]. There is no loss of generality here, since tree unification is impractical for anything but toy problems (as discussed in Sect. 2). The following approach is applicable to the graph unification algorithm provided in App. A, which is the basis for the algorithm developed in this paper, as well as all of the asymptotically superior algorithms discussed in the next section. This includes Martelli and Montanari's almost-linear algorithm [28]. Although their algorithm manipulates multisets of multiequations, the heart of the algorithm is ordering the processing of multiequations in such a way that substitutions are not applied. So their multiequations can be viewed as a representation for graphs in a similar manner to the use of equations in what follows.

Definition 6.1 (Solved Form) *An equation is an unordered pair $\tau = \tau'$. A set of equations S is in solved form if it is of the form $\{\alpha_i = \tau_i \mid i = 1, \dots, n\}$, where:*

1. *The equations of S can be partitioned into subsets S'_1, \dots, S'_k , where each S'_j has at most one equation of the form $\alpha = \tau(\overline{\tau}_m)$, and where partitions are pairwise disjoint: $(\alpha = \tau) \in S, (\alpha = \tau') \in S'$ implies $S = S'$.*

2. There does not exist $S' \subseteq S$ such that $S' = \{\beta_1 = \tau'_1, \dots, \beta_k = \tau'_k\}$, $\beta_{i+1} \in FV(\tau'_i)$ for $i = 1, \dots, k-1$, and $\beta_1 \in FV(\tau'_k)$.

We make use of several metafunctions: $instan(S)$ is defined to be θ , where $(S, \{\}) \xRightarrow{*} (S', \theta)$ under the following transition, which is not applicable to S' :

$$(S \cup \{\alpha = \mathbf{t}(\overline{\tau}_n)\}, \theta) \implies (\{\mathbf{t}(\overline{\tau}_n)/\alpha\}S, \{\mathbf{t}(\overline{\tau}_n)/\alpha\} \circ \theta)$$

Define $alias(S)$ to be θ , where $(S, \{\}) \xRightarrow{*} (S', \theta)$ under the following transition, which is not applicable to S' :

$$(S \cup \{\alpha = \beta\}, \theta) \implies (\{\beta/\alpha\}S, \{\beta/\alpha\} \circ \theta)$$

Definition 6.2 (Type Inference With Explanations) Given A, ξ, e , and τ , with $n = |e|$, then a specification for type inference algorithms with explanations is given by the inference rules in Fig. 18. (Π, TV) is a type derivation for $A; \xi; \{\} \triangleright_1^n e \implies (\tau, \xi)$ if Π is a derivation for this judgement using these inference rules, and TV is a mapping from $\{1, \dots, n\}$ to finite sets of type variables such that $TV(i) \cap TV(j) \neq \{\} \implies i = j$. The explanation trace specification is given by the rules for the judgement form $\xi; \xi_l \triangleright_i ((\tau, \phi, \varepsilon) = (\tau', \phi', \varepsilon')) \implies \xi'_l$. The VT and TV inference rules use the predicate $\mathcal{A}_{\alpha, \phi, \varepsilon}(\xi, i, \varepsilon')$, while the VV rule uses the predicate $\mathcal{A}_{\alpha, \phi, \varepsilon}^{\beta, \phi', \varepsilon'}(\xi, i, \varepsilon'')$, defined in Fig. 19.

The allowability predicates are central to the modifications introduced in Sect. 5 to handle polymorphic aliasing. The i th vertex in a derivation Π is the vertex labelled with judgement $A; \xi; S \triangleright_k^i e \implies (\tau, S')$. Π_j denotes the subderivation of Π with root judgement $A'; \xi; \xi_l \triangleright_i^j e' \implies (\tau', \xi'_l)$ (for some $A', i, e', \tau', \xi_l, \xi'_l$). The **let-bound type variables** in a derivation are those variables introduced by the instantiation of a variable with polymorphic type, or introduced to represent the return type of an application. To formalize this, define:

$$\begin{aligned} head(j) &= i \text{ if the root judgement of } \Pi_j \text{ is } A'; \xi; \xi_l \triangleright_i^j e' \implies (\tau', \xi'_l) \\ spine(i) &= \begin{cases} \{i\} & \text{if } i \text{ corresponds to a VAR rule} \\ \{i\} \cup spine(i_1) & \text{if } i \text{ corresponds to an application } (e_1 e_2), \\ & \text{where } \Pi_{i_1} \text{ is the subderivation for the typing of } e_1 \\ spine(i_1) & \text{if } i \text{ corresponds to an abstraction } \lambda x \cdot e_1 \\ & \text{where } \Pi_{i_1} \text{ is the subderivation for } e_1 \\ spine(i_2) & \text{if } i \text{ corresponds to a let-statement } \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \\ & \text{where } \Pi_{i_2} \text{ is the subderivation for } e_2 \end{cases} \\ appvars(i) &= \bigcup \{TV(j) \mid j \in spine(i)\} \end{aligned}$$

Then a type variable α is **let-bound** if $\alpha \in appvars(i)$ for some i . If $\alpha \in appvars(i)$ and $head(i) = x$, we say that α is introduced for x , and we call this occurrence of x the binding site for α . The following definition, which is used in the definition of allowability, uses this notion of **let-bound** type variables.

Definition 6.3 (Equations With Explanations) An equation set with explanations is a set of triples ξ of the form $\tau \stackrel{\xi}{=} \tau'$, where ε is a set of integers (indices into a derivation tree as made clear below), where the underlying equation set satisfies the previous definition of being in solved form. Define the reduction of an equation set with explanations ξ to a simple equation set S in the obvious way: $\|\{\tau_i \stackrel{\varepsilon_i}{=} \tau'_i\}_i\| = \{\|\tau_i \stackrel{\varepsilon_i}{=} \tau'_i\|\}_i = \{\tau_i =$

VAR	$\frac{A(x) = \forall \bar{\alpha} \cdot \tau \text{ where } TV(i) = \{\bar{\beta}\}}{A; \xi; \xi_l \triangleright_i^1 x \Rightarrow (\{\bar{\beta}/\bar{\alpha}\} \tau, \xi_l)}$
ABS	$\frac{A, x : \alpha; \xi; \xi_l \triangleright_i^j e \Rightarrow (\tau, \xi'_l) \quad TV(j+1) = \{\alpha\}}{A; \xi; \xi_l \triangleright_i^{j+1} \mathbf{fn} \ x \Rightarrow e \Rightarrow (\alpha \rightarrow \tau, \xi'_l)}$
APP	$\frac{A; \xi; \xi_l \triangleright_i^j e_1 \Rightarrow (\tau_1, \xi'_l) \quad A; \xi; \xi'_l \triangleright_{j+1}^k e_2 \Rightarrow (\tau_2, \xi''_l) \quad \ \xi'_l\ \cap \ \xi''_l\ = \{\}}{\xi; \xi''_l \cup \xi'_l \triangleright_{k+1} ((\tau_1, ()), \{\}) = (\tau_2 \rightarrow \alpha, ()), \{\}) \Rightarrow \xi'''_l \text{ where } TV(k+1) = \{\alpha\}}{A; \xi; \xi_l \triangleright_i^{k+1} e_1 e_2 \Rightarrow (\alpha, \xi''_l \cup \xi'''_l)}$
LET	$\frac{A; \xi; \xi_l \triangleright_{j+1}^k e_1 \Rightarrow (\tau_1, \xi'_l) \quad \theta = \mathit{instan}(\ \xi'_l\) \text{ and } \theta' = \mathit{alias}(\ \xi'_l\) \quad \{\bar{\alpha}\} = FV(\theta'(\theta(\tau_1))) - FV(\theta'(\theta(A))) \text{ and } \sigma = \forall \bar{\alpha} \cdot (\theta' \uparrow \{\bar{\beta} \in FV(\theta(\tau_1)) \mid \theta'(\bar{\beta}) \in \{\bar{\alpha}\}\})(\tau_1)}{A, x : \sigma; \xi; \xi'_l \triangleright_i^j e_2 \Rightarrow (\tau_2, \xi''_l)}{A; \xi; \xi_l \triangleright_i^{k+1} (\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2) \Rightarrow (\tau_2, \xi''_l)}$
ID	$\frac{}{\xi; \xi_l \triangleright_i ((\tau, \varphi, \varepsilon) = (\tau', \varphi', \varepsilon')) \Rightarrow \{\}}$
DEC	$\frac{\xi; \xi_l \triangleright_i ((\tau_j, \varphi, \varepsilon) = (\tau'_j, \varphi', \varepsilon')) \Rightarrow \xi_l^j \text{ for } j = 1, \dots, n \quad \ \xi_l^i\ \cap \ \xi_l^j\ \neq \{\} \Rightarrow i = j}{\xi; \xi_l \triangleright_j ((\mathbf{t}(\tau_1, \dots, \tau_n), \varphi, \varepsilon) = (\mathbf{t}(\tau'_1, \dots, \tau'_n), \varphi', \varepsilon')) \Rightarrow \xi_l^1 \cup \dots \cup \xi_l^n}$
UNF	$\frac{(\alpha_h \stackrel{\varepsilon_h}{=} \alpha_{h+1}) \in \xi \text{ for } h = 1, \dots, n-1 \quad (\alpha_n \stackrel{\varepsilon_n}{=} \mathbf{t}(\bar{\tau})) \in \xi}{\xi; \xi_l \triangleright_i ((\mathbf{t}(\bar{\tau}), (\bar{\beta}_m, \bar{\alpha}_n), \varepsilon_1 \cup \dots \cup \varepsilon_n \cup \varepsilon) = (\tau', \varphi', \varepsilon')) \Rightarrow \xi'_l}{\xi; \xi_l \triangleright_i ((\alpha, (\bar{\beta}_m), \varepsilon) = (\tau', \varphi', \varepsilon')) \Rightarrow \xi'_l}$
VV	$\frac{\mathcal{A}_{\alpha, \varphi, \varepsilon}^{\beta, \varphi', \varepsilon'}(\xi, i, \varepsilon'')}{\xi; \xi_l \triangleright_i ((\alpha, \varphi, \varepsilon) = (\beta, \varphi', \varepsilon')) \Rightarrow \{\alpha \stackrel{\varepsilon''}{=} \beta\}}$
VT	$\frac{\mathcal{A}_{\alpha, \varphi, \varepsilon}(\xi, i, \varepsilon'') \quad (\alpha = \mathbf{t}(\bar{\tau}_n)) \notin \ \xi'_l\ }{\xi; \xi_l \triangleright_i ((\mathbf{t}(\bar{\tau}_n), (\bar{\beta}_m, \alpha), \varepsilon'') = (\mathbf{t}(\bar{\tau}_n), \varphi', \varepsilon')) \Rightarrow \xi'_l}{\xi; \xi_l \triangleright_i ((\alpha, (\bar{\beta}_m), \varepsilon) = (\mathbf{t}(\bar{\tau}_n), \varphi', \varepsilon')) \Rightarrow \xi'_l \cup \{\alpha \stackrel{\varepsilon''}{=} \mathbf{t}(\bar{\tau}_n)\}}$

Figure 18: Type Inference With Type Explanation

$\tau'_i\}_i$. A polymorphic alias path is a sequence of triples $\alpha_i \stackrel{\{k\}}{=} \alpha_{i+1}$ for $i = 0, \dots, n$, where $\alpha_1, \dots, \alpha_n$ are **let-bound**. A polymorphic path is a sequence of equalities of the form:

$$\alpha_j \stackrel{k_j}{=} \alpha_{j+1} \text{ for } j = 1, \dots, m_i - 1, \quad \alpha_{m_i}^i \stackrel{k_i}{=} \mathbf{t}(\bar{\tau}^i) \text{ for } i = 1, \dots, n, \\ \alpha_1^{i+1} \in FV(\mathbf{t}(\bar{\tau}^i)) \text{ for } i = 1, \dots, n-1$$

where α_1^1 and each α_j^i is **let-bound**, $2 \leq j \leq m_i, i \leq n$.

The definition of allowability contains some non-local conditions. The inductive nature of this inference system may be made explicit by associating, with each subderivation Π_i , the set of indices for the roots of the derivations in which Π_i must occur as a subderivation. These index sets then become part of a side-condition

$$\begin{array}{l}
\mathcal{A}_{\alpha, \varphi, \varepsilon}(\xi, i, \varepsilon') \iff \left\{ \begin{array}{l} (\varepsilon' = \varepsilon \cup \{i\}) \text{ or } (\varepsilon' = \{k\}) \text{ and} \\ \Pi_i \text{ is a subderivation of } \Pi_k \text{ and } (\mathcal{AT}_{\alpha, \varepsilon}(\xi, k) \text{ or } \mathcal{AE}_{\alpha, \varphi}(\xi, k)) \end{array} \right. \\
\mathcal{AT}_{\alpha, \varepsilon}(\xi, k) \iff \left\{ \begin{array}{l} \varepsilon = \{ \} \text{ and } \alpha \text{ is reachable along a polymorphic alias path from} \\ \text{some } \beta \in \text{appvars}(k) \end{array} \right. \\
\mathcal{AT}_{\alpha, \varepsilon}^{\alpha', \varepsilon'}(\xi, k) \iff \left\{ \begin{array}{l} \varepsilon = \varepsilon' = \{ \}, \text{ and } \alpha \text{ or } \alpha' \text{ is reachable along a polymorphic alias} \\ \text{path from some } \beta \in \text{appvars}(k) \end{array} \right. \\
\mathcal{AE}_{\alpha, \varphi}(\xi, k) \iff \left\{ \begin{array}{l} \alpha \text{ is reachable along a polymorphic path given by the variables in} \\ \text{the sequence } \varphi, \text{ where the first variable in this path is some } \beta \in \\ \text{appvars}(k) \end{array} \right. \\
\mathcal{A}_{\alpha, \varphi, \varepsilon}^{\alpha', \varphi', \varepsilon'}(\xi, i, \varepsilon'') \iff \left\{ \begin{array}{l} (\varepsilon'' = \varepsilon \cup \varepsilon' \cup \{i\}) \text{ or } (\Pi_i \text{ is a subderivation of } \Pi_k \text{ and} \\ (\varepsilon'' = \varepsilon \cup \{k\} \text{ and } \mathcal{AE}_{\alpha', \varphi'}(\xi, k)) \text{ or} \\ (\varepsilon'' = \varepsilon' \cup \{k\} \text{ and } \mathcal{AE}_{\alpha, \varphi}(\xi, k)) \text{ or} \\ (\varepsilon'' = \{k\} \text{ and } \mathcal{AE}_{\alpha, \varphi}(\xi, k) \text{ and } \mathcal{AE}_{\alpha', \varphi'}(\xi, k)) \text{ or} \\ (\varepsilon'' = \{k\} \text{ and } \mathcal{AT}_{\alpha, \varepsilon}^{\alpha', \varepsilon'}(\xi, k)) \end{array} \right.
\end{array}$$

Figure 19: Definition of Allowability Predicates

on the inductive definition of derivations. We ignore the tiresome aspects of formulating this explicitly inductive definition of derivations further. We only note that it is exactly this non-local nature of allowability which requires explanations to be updated at certain points during type inference.

Theorem 3 *Given $A, e, n = |e|$. Then $\vdash A; \xi; \{ \} \triangleright_1^n e \implies (\tau, \xi)$, for some ξ and τ , if and only if $\mathcal{W}_A(e) = \langle \tau, \theta \rangle$, where $\theta = \text{alias}(\|\xi\|) \circ \text{instan}(\|\xi\|)$.*

The above specification of type explanations is independent of the algorithm used to compute the equation set in solved form. The restriction on the equation set with explanations ξ formalizes the notion of a type explanation as an accumulation of the explanations on two paths, from the root of the trees corresponding to the type graphs being unified, to a possible instantiation point for one of the variables instantiated during that call to unification. The definition ensures that, for every type variable instantiated during unification, the explanation associated with that instantiation is consistent with the explanation at exactly one of these points. This generalization of the notion of type explanation is applicable for example to the algorithms of Paterson-Wegman [39] and Martelli-Montanari [28] discussed in the next section, which improve the asymptotic performance of unification by a suitable reordering of the equations as they are being solved.

The correspondence between this declarative specification and the algorithm provided in Sect. 3 and Sect. 4 is fairly immediate, if we restrict the definition of allowability to: $\mathcal{A}_{\alpha, \varphi, \varepsilon}(\xi, i, \varepsilon') \iff \varepsilon' = \varepsilon \cup \{i\}$. Variables are implemented as references (heap cells), and equalities $\alpha = \tau$ are implemented by making the heap cell for α point to τ . The definition of solved form for equation sets essentially restricts the undirected graphs represented by these equations to trees. These trees are represented in the algorithm of App. C by

parent pointers (essentially UNION-FIND trees) associated with each vertex. Reversing of alias paths is necessary because the equality $\alpha = \beta$ can be represented either as a pointer from α to β , or as a pointer from β to α . On the other hand, the algorithm brings out several practical aspects of type explanation which are completely elided by the abstract specification, because the latter does not consider the implementation of equation sets. These issues arise in the practical algorithm provided in App. C because the representation of variable aliasing is at the heart of all implementations of unification algorithms. These issues are discussed at some length in Sect. 4. Without the modifications described in that section, the explanations constructed by a graph unification algorithm (as described in the next section) will be incorrect according to this specification.

The following lemma is at the heart of our treatment of polymorphic aliasing:

Lemma 6.1 *Given (Π, TV) a derivation for $\vdash A; \xi; \{\} \triangleright_1^{|e|} e \implies (\tau, \xi)$. Assume the constraint $(\alpha \stackrel{\{k\}}{=} \tau) \in \xi$ is introduced in type-checking the application $(e_1 e_2)$, where the explanation is permitted by an \mathcal{AT} allowability predicate. Let i be the index of the corresponding type judgement in Π for this application, with subderivations Π_{i_1} and Π_{i_2} for type judgements for e_1 and e_2 . Then $\{\alpha\} \cup \{\beta \in FV(\tau) \mid \beta \text{ is } \mathbf{let}\text{-bound}\} \subseteq \text{appvars}(i_1) \cup \text{appvars}(i_2) \cup TV(i)$.*

Lemma 6.2 *Suppose $\mathcal{A}_{\alpha, \varphi, \varepsilon}(\xi, i, \{k\})$ because $\mathcal{AT}_{\alpha, \varepsilon}(\xi, k)$, or $\mathcal{A}_{\alpha, \varphi, \varepsilon}^{\alpha', \varphi', \varepsilon'}(\xi, i, \{k\})$ because $\mathcal{AT}_{\alpha, \varepsilon}^{\alpha', \varepsilon'}(\xi, k)$, so α or α' is reachable via a polymorphic alias path from some $\beta \in \text{appvars}(k)$. Then the expression at k contains the binding site of every \mathbf{let} -bound type variable in this polymorphic alias path. Furthermore the expression at type derivation vertex i is a superterm of the expression at j , for every $j \leq i$ such that type-checking the expression at j caused one of the equations in the polymorphic alias path to be added.*

PROOF: By induction on the length of the polymorphic path, using Lemma 6.1. The last part of the lemma follows from the fact that type inference is bottom-up. This means that in type-checking the application $(e_1 e_2)$, the set of \mathbf{let} -bound type variables bound in e_1 (and therefore reachable via polymorphic alias paths or polymorphic paths from variables in the type of e_1) is disjoint from that for e_2 . Any extension of a polymorphic alias path must result from following polymorphic alias paths or polymorphic paths from \mathbf{let} -bound variables in the types of e_1 and e_2 , therefore $(e_1 e_2)$ is a superterm of e_1 and e_2 , which are superterms of any terms which gave rise to the edges in these paths during the type-checking of e_1 and e_2 . \square

Lemma 6.3

1. *Suppose $\mathcal{A}_{\alpha, \varphi, \varepsilon}(\xi, i, \{k\})$ because $\mathcal{AE}_{\alpha, \varphi}(\xi, k)$. Then the expression at k contains the binding site of every \mathbf{let} -bound type variable in the polymorphic paths traced by φ to α .*
2. *Suppose $\mathcal{A}_{\alpha, \varphi, \varepsilon}^{\alpha', \varphi', \varepsilon'}(\xi, i, \varepsilon'')$ because $\mathcal{AE}_{\alpha, \varphi}(\xi, k)$ and/or $\mathcal{AE}_{\alpha', \varphi'}(\xi, k)$ (so $\varepsilon'' \subseteq \varepsilon \cup \varepsilon' \cup \{k\}$). Then the expression at k contains the binding site of every \mathbf{let} -bound type variable in any polymorphic paths traced by φ and φ' to α and α' , respectively.*
3. *The expression at type derivation vertex i is a superterm of the expression at j , for every $j \leq i$ such that type-checking the expression at j caused one of the equations in any of the polymorphic paths to be added.*

PROOF: Part (3) is verified in a similar way to the last part of Lemma 6.2, while Part (2) is similar to Part (1). We concentrate therefore on verifying Part (1). Suppose the polymorphic path is simply $\alpha \stackrel{\{k\}}{=} \mathbf{t}(\overline{\tau}_m)$, where

$\beta \in FV(\tau_i)$ for some i and the equation $\beta = \tau$ is added. Suppose the former equation arose from type-checking the application $(e_1 e_2)$, so $\alpha, \beta \in \text{appvars}(e_1) \cup \text{appvars}(e_2) \cup \{j\}$ (where j in the derivation tree index for $(e_1 e_2)$). Since type inference is a bottom-up process and the new equation is added as a result of unifying a pair of types, one of which contains α , the equation $\beta = \tau$ must arise from type-checking some superterm of $(e_1 e_2)$. So the binding sites for the **let**-bound variables in $\alpha \stackrel{\{k\}}{=} \tau(\overline{\tau_m})$ must be included in the explanation for the addition of $\beta = \tau$. The general result follows by induction on the length of the polymorphic path. \square

These lemmata are the justification for our approach to explanations on polymorphic alias paths. The definition of allowability permits the explanations associated with the aliasing equations $\alpha = \beta_1, \beta_1 = \beta_2, \dots, \beta_n = \tau$, where each equation arises from a polymorphic application and each β_i is **let**-bound, to be expanded to the greatest enclosing polymorphic application, as justified by these lemmata. This in turn allows a polymorphic alias path $\alpha \stackrel{\{i\}}{=} \beta_1, \beta_1 \stackrel{\{i\}}{=} \beta_2, \dots, \beta_{n-1} \stackrel{\{i\}}{=} \beta_n, \beta_n \stackrel{\{i\}}{=} \tau$ to be replaced in the output of the type explainer with the single explanation $\alpha \stackrel{\{i\}}{=} \tau$, which is a consequence of transitivity. We regard a polymorphic alias path $\alpha \xrightarrow{e_0} \beta_1 \xrightarrow{e_1} \dots \beta_i \xrightarrow{e_i} \beta_{i+1} \dots \beta_n \xrightarrow{e_n} \tau$ constructed by the algorithm in App. C as a representation for the polymorphic alias path $\alpha \stackrel{\{k\}}{=} \beta_1 \stackrel{\{k\}}{=} \dots \beta_i \stackrel{\{k\}}{=} \beta_{i+1} \dots \beta_n \stackrel{\{k\}}{=} \tau$, where the type derivation vertex labelled with k corresponds to the expression e_n . As verified by Lemma. 6.4, the algorithm inflates the scope of the explanation on the equations in a polymorphic alias path as required during bottom-up type-checking, in order to extend the polymorphic alias path.

Lemma 6.4 *Suppose the algorithm in App. C adds an instantiation edge with explanation e , or updates the explanation on an edge with explanation e . Then this explanation is permitted by allowability.*

PROOF: We verify that the explanation satisfies the subderivation restriction. If the variable being instantiated is not reached via a polymorphic path, then Lemma 6.2 is used for the verification, by induction on the length of the polymorphic alias path. The case where the variable being instantiated is reached via a polymorphic path is verified using Lemma 6.3, by induction on the length of the polymorphic path. The remaining part of the verification is based on Lemma 6.1. \square

Theorem 4 *The unification algorithm in App. C is correct in the sense that it implicitly constructs a derivation in the derivation system presented in Fig. 18. In the case where an instantiation of a variable arises after tracing one or two polymorphic paths from the original types being unified, the algorithm simplifies the explanation by using the \mathcal{AE} allowability predicate to omit that part of the explanation corresponding to a polymorphic path (which is sufficient, by Lemma 6.3). Furthermore in the case where allowability is used to expand the scope of an explanation, the algorithm chooses the minimal such explanation while maximizing the length of polymorphic alias paths (which is sufficient, by Lemma 6.2).*

7 Type Explanation With Other Unification Algorithms

We have consciously chosen to use a graph-based implementation of Robinson’s original unification algorithm, as given in App. A. We refer to this algorithm as *naive graph unification*. We have chosen to use this algorithm because it is simple, so that the extensions for type explanation can be clearly described and motivated, and because it is by far the most widely used implementation of unification [33]. For example, in looking at several widely-used implementations of ML and Haskell (ANU ML, Standard ML of New Jersey,

Edinburgh ML, Yale Haskell, Glasgow Haskell, Gofer and CAML Light), all use naive graph unification. We were not able to find a single implementation of type inference (in a production environment, such as one of the above) which used one of the algorithms discussed in this section⁴. This reflects several years of experience that the uses of unification in type inference are normally small and frequent, so that the overhead of most of these algorithms swamps any performance gain from their asymptotically superior performance. This is also true of the logic programming community. Prolog compiler technology is essentially based on compiling the naive graph unification algorithm given in App. A [2].

Naive graph unification is based on unifying type graphs. In theory its worst-case complexity is exponential, due to the occurs check and the retraversal of shared subgraphs. However the kinds of examples which exhibit this exponential complexity do not arise in practice [33]. Incontrovertible evidence for this is offered by the spectacular success of type inference in ML and Haskell. Experience with Haskell compilers suggests that the real issue in implementing type inference is the implementation of substitution [45]. Naive graph unification only merges two vertices where one of these is a variable vertex. The graph unification algorithm in App. D avoids the theoretical exponential blow-up in naive graph unification by merging any pair of vertices which are unified, and the children of those vertices. This algorithm was proposed independently by Baxter [5, 6] and by Huet [22]. Huet also considered the extension of this algorithm to unifying infinite terms. Although neither algorithm has ever been published, several very similar algorithms have been published in tutorials and surveys [33, 1, 26]. The algorithm presented in App. D is based on the description given by Knight [26].

The algorithm is based on a well-known method for checking the equivalence of finite-state automata, using the UNION-FIND algorithm to merge equivalent states [49]. In this case states correspond to vertices in the type graph, while state transitions correspond to descendant edges [13] between parent and child vertices. Recursion is triggered in the case where two type constructor vertices are merged. In this case n subtrees are removed by merging, and n subtrees are traversed recursively, giving an overall linear complexity for graph traversal. The remaining source of complexity is in the implementation of the UNION and FIND operations for merging equivalent states. UNION can be implemented as a constant time operation which sets the parent pointer of one vertex to point to the other vertex. Load balancing and path compression can then be used to achieve a $O(n \cdot G(n))$ amortized time complexity for the FIND operations, where $G(n)$ is the inverse of Ackerman's function. Part of the reason for the non-exponential behaviour of the algorithm is that the occurs check is postponed until the end of the algorithm; the occurs check is then the standard linear-time depth-first-search-based acyclicity check for directed graphs [48]. The Baxter/Huet algorithm thus is $O(n \cdot G(n))$, which means that for all practical purposes it is linear. Furthermore, as can be seen by comparing App. A and App. D, it is similar enough in some of the details to the familiar naive graph unification algorithm to be fairly accessible to implementors. Finally the Baxter/Huet algorithm also has the useful property that it is applicable to the unification of infinite terms [22].

Asymptotically linear algorithms have also been proposed in the literature. Actually two different algorithms were developed independently and contemporaneously: Martelli and Montanari's linear-time algorithm [27] and Paterson and Wegman's algorithm [39, 12]. The first of these was never published. Since the latter is the only algorithm whose description is available in the published literature [39, 12], we concentrate on Paterson and Wegman's algorithm. This algorithm can be viewed as an extension of the Baxter/Huet graph unification algorithm, where the next pair of vertices to be merged is chosen such that those vertices

⁴The only exception to this are current work, by L. Birkendal and independently by the first author [13], to add the Baxter/Huet algorithm to the Standard ML Kit compiler.

are not accessible from the remaining vertices in the graph. This leads to a considerable complication in the algorithm, including parent pointers on vertices, unified vertices needing to be deleted, and two traversals required of the graphs to be unified, *before unification*, to unset various markers [18].

The conventional wisdom among implementors of ML and Haskell type checkers is that the Paterson-Wegman algorithm is completely impractical for type inference [D. MacQueen, G. Morrisett, K. Hammond, personal communications]. Any asymptotically optimal performance from the algorithm is swamped by the overhead of initializing and manipulating very complicated data structures, for unification problems which are invariably small and frequent. The only place where we have found an application of this algorithm in type-checking is in the signature-matching code for the Standard ML of New Jersey compiler [3]. This algorithm is only called once in the elaboration of an entire structure, and was intended to deal with very large functor applications. Even the implementors of this signature-matching code consider the use of Paterson-Wegman in type inference a dubious proposition [D. MacQueen, G. Morrisett, personal communications]. As already noted, SML/NJ uses the simple and familiar naive graph unification algorithm for type inference. Some indication of the difference in the scale of complexity between this algorithm and Paterson-Wegman may be gauged from the fact that, in the SML/NJ v1.03f compiler, the former consists of 206 lines of code, while the latter consists of 1330 lines of code. With our type explainer in SML/NJ, unification consists of 296 lines of code, plus about 250 lines of code for support routines. Extending Paterson-Wegman with our approach to type explanation would require an at least comparable increase in the size of the implementation, as discussed below.

Martelli and Montanari have proposed another almost-linear-time unification algorithm [28] (Unfortunately, since this algorithm was published, it is frequently confused with their earlier algorithm and incorrectly referred to as “linear”). Both the Baxter/Huet algorithm and the Paterson-Wegman algorithm merge vertices using the UNION-FIND algorithm, using parent pointers to point to representative vertices for equivalence classes (the latter avoids the need to chase down long alias paths by a suitably ordering of the vertices which are merged). We refer to these, and to naive graph unification, as *graph algorithms*. The Martelli-Montanari algorithm avoids the use of pointers by using multisets of variables. Whereas the other algorithms manipulate graphs, this algorithm manipulates multisets of *multi-equations*, $S = M$, where S is a multiset of variables and M a multiset of type constructor vertices. Essentially, in a multiequation of the form $\{\alpha_1, \dots, \alpha_n\} = \{\tau\}$, the multiset of variables $\{\alpha_1, \dots, \alpha_n\}$ represents an equivalence class of merged vertices in the graph being unified. The algorithm merges such multiequations where the left hand sides share variables, and reduces the result to a multiset of multiequations, all of this form. The linear behaviour of the algorithm comes from “selecting at each step a multiequation in such a way that no substitution ever has to be applied” [28, Page 271]. The non-linear behaviour comes from the merging of variable lists when merging multiequations; since a variable contains a pointer to its multiequation, this requires a resetting of this pointer for each variable in a multiequation which is deleted.

Martelli and Montanari cite a study by Trum and Winterstein (unfortunately, again, unpublished and not available in the literature [50]) which apparently demonstrates that overall their algorithm fares better than either Baxter/Huet or Paterson-Wegman. Despite this, their algorithm has never been implemented in a production-quality ML or Haskell compiler. Martelli and Montanari note [28, Page 280] that the Baxter/Huet algorithm performs the best in the situation where there is a high probability of unification failing due to a clash of type constructors. Our experience with ML type inference, in the context of using ML both for education and for research, suggests that this is by far the most likely scenario when using ML type inference⁵.

⁵The author, and others, have observed the phenomenon wherein ML programmers treat the type-checker as a demanding but

We conclude by mentioning the algorithm of Corbin and Bidoit [11, 33]. Although Corbin and Bidoit’s algorithm is offered as a “rehabilitation” of Robinson’s original algorithm, the data structures for their RG2 algorithm are actually somewhat similar to those for Baxter/Huet. In particular arbitrary vertices are merged during unification, not just variable vertices and other vertices. In contrast to Baxter/Huet, Corbin and Bidoit perform the occurs check during unification, using a stamping scheme for vertices to avoid the (theoretical) exponential blow-up associated with the occurs check in the naive unification algorithm. The resulting algorithm has quadratic complexity. Equivalence classes of vertices are implemented using the UNION-FIND algorithm. The FIND operation is simplified by not incorporating load balancing or path compression. Corbin and Bidoit also compare the actual measured performance of Martelli-Montanari’s almost-linear algorithm [28] (MM) with the applicative unification algorithm given in Sect. 2 (RT), the naive graph unification algorithm given in App. A⁶ (RG1), and their algorithm as described above (RG2). They report that:

1. Not surprisingly, given Sansom’s results from profiling GHC [45], RT performs significantly slower than all of the other algorithms.
2. RG2 is actually faster than MM on all “real” problems for which they gathered performance data.
3. RG1 (which in theory has exponential complexity) in practice is competitive with MM. This observation can be attributed to the fact that problems that cause exponential complexity do not arise in practice, and that RG1 has a considerably simpler implementation.

We now consider how our approach to type explanation can be adapted to these algorithms. We first of all clarify why it is useful to add type explanation to any of these algorithms. Herbrand [19] provided the original and most general characterization of unification, in terms of solving equations over a free term algebra. All of the algorithms discussed in this section essentially compute such a solution, in terms of a set of equations of the form $\alpha = \tau$. The graph algorithms implement variables as pointers (heap cells), as discussed in Sect. 2. The Martelli-Montanari algorithm [28] groups such equations together into multiequations and represents these multiequations explicitly. However the solution computed by the algorithm is still understood as a solution to a system of equations [28, Page 264]. Therefore the formalization of type explanation provided in the previous section is applicable to all of these algorithms. We now consider the practical implications of adapting our type explanation algorithm, based on naive graph unification, to one of these algorithms.

We refer to the algorithms of Baxter, Huet, Paterson-Wegman, and Corbin-Bidoit as the *vertex-merging algorithms*. For the case of the vertex-merging algorithms, the adaptation for type explanation is fairly straightforward. The only aspect of these algorithms which introduces fresh complications is the merging of type constructor vertices. Figure 20(a) considers the type graph constructed as a result of type-checking the code fragment:

```
F x; G y; if true then F else G; F 3
```

under the type declarations:

reliable “program verifier.” Rather than understand all of the types for their program, they exhibit a preference for using the type-checker to automate this process.

⁶Actually RG1 is naive graph unification with Corbin and Bidoit’s stamping scheme for avoiding repeated occurs checks. RG1 is still theoretically exponential, due to repeated traversal of shared terms. The algorithm for type explanation given in App. C is completely independent of the implementation of the occurs check, and so can also be considered as a redesign of RG1 for type explanation.

F: 'A0, x: 'A1, G: 'A3, y: 'A4

The corresponding type explanation for the type of F is⁷:

```

F : 'A1 -> 'B0
F x ..... gives F : 'A0 = 'A1 -> 'B0

F : 'A4 -> 'B0
G y ..... gives G : 'A2 = 'A3 -> 'B1
if true ..... gives x : 'A1 = 'A3

F : int -> 'B0
F 3 ..... gives y : 'A3 = int

```

This explanation is consistent with the formal definition of type explanations given in the previous section, since the algorithms of Baxter/Huet, Paterson-Wegman and Corbin-Bidoit all only operate on “one-level terms” [28] with equations of the form $\alpha = \tau(\beta_1, \dots, \beta_n)$. Therefore the above explanation is consistent with the following equation set:

$$'A0 = \alpha, \alpha = 'A1 \rightarrow 'B0, 'A2 = \beta, \beta = 'A3 \rightarrow 'B1, \alpha = \beta, 'A3 = \gamma, \gamma = \text{int}$$

We see that, rather than making type explanations any easier, these algorithms actually complicate the picture even further. This is entirely to be expected. To be usable, the output of type explanation must be reasonably intuitive and easy to understand, even for novice programmers. This is possible with naive graph unification precisely because the algorithm is so simple. With these asymptotically superior graph unification algorithms, the more complicated processing of the algorithm is reflected in the more convoluted and unintuitive type explanations.

The following approach, which is a generalization of the approach in Sect. 4, fixes the implementation of the Baxter/Huet and Corbin-Bidoit algorithms to provide a more intuitive explanation in this case. We call a sequence of instantiation edges $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$, where each τ_i is a type constructor vertex, a *merge path*. A *variable alias path* is a sequence of edges $\alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_m$. An *alias path* is a sequence of merge paths and variable alias paths (p_1, \dots, p_k) , where the last vertex in p_i is connected by an instantiation edge to the first vertex in p_{i+1} , and where it is not the case that p_i and p_{i+1} are both variable alias paths or are both merge paths, for any $i < k$. The definitions of the vertex merging algorithms ensure that, if p_k is a variable alias path, then $k = 1$. Let $p_i, 1 \leq i \leq 2$, be the first merge path in such an alias path (assuming there is one). The *Deref* operation (which implements FIND in UNION-FIND) now performs the following:

1. Find the first and last vertices in p_i , call these v and v' respectively.
2. Reverse the edges on the merge path rooted at v .
3. Set v 's instantiation edge to point to the previous sink of the destination edge for v' .

⁷The return types of function applications are initially `let`-bound variables. We have avoided stating this minor point, and only mention it here to justify the fact that type explanation does not bother unfolding `'B0` any further. In reality we expect this variable to be instantiated to something else, even if this is only a generic type variable after generalization.

Note that whereas the reversal of edges in a variable alias path is done during UNION, the reversal of edges in a merge path must be done during FIND. Figure 20(b) demonstrates the result of this on the example given earlier. We now have the following explanation for the type of **F**:

```

F : 'A1 -> 'B0
F x ..... gives F : 'A0 = 'A1 -> 'B0

F : int -> 'B0
F 3 ..... gives x : 'A1 = int

```

The Baxter/Huet algorithm relies crucially on path compression for its asymptotic performance. Our algorithm demonstrates that, although path compression does not have to be omitted altogether, it must be used carefully in order to preserve the correctness of the explanations. In particular, it can *only* be used for the case of traversing a variable alias path whose last vertex points to a merge path (since edge reversal is done by FIND for merge paths). Omitting path compression entirely from the Baxter/Huet algorithm leaves it with a $O(n \cdot \log n)$ complexity.

The issues and algorithm adaptations provided in Sect. 3 and Sect. 5 remain equally relevant to the Paterson-Wegman algorithm. This algorithm appears to avoid the issues discussed in Sect. 4, by only considering equations between type graph vertices which are no longer reachable. However it is not clear to the author how it could be modified, analogous to the discussion above, to provide more intuitive explanations with merging of type constructor vertices. On the other hand, as we have noted, the use of Paterson-Wegman in type inference is not a serious proposition, because of its very high overhead.

Sect. 3 and Sect. 5 are equally relevant to Martelli and Montanari's almost-linear-time algorithm. The issue of variable aliasing in Sect. 4 does not appear immediately relevant, since equivalence classes of variables are represented by multiequations of the form $\{\alpha_1, \dots, \alpha_n\} = \{\tau\}$ and $\{\alpha_1, \dots, \alpha_n\} = \{\}$ (bound and unbound, respectively). However, we still need to keep track of the reasons for the aliasing of two variables. Martelli and Montanari represent the left-hand sides of multiequations as lists of variables, so it appears plausible that we store this part of the type explanation with the links in these lists. This is exactly how aliasing information is stored in our type explanation algorithm. Martelli and Montanari also suggest [28, Page 276] that the $O(m \log m)$ complexity of merging variable lists (and resetting multiequation pointers) could be reduced to $O(m \cdot G(m))$ by using UNION-FIND trees rather than lists to represent the left-hand sides of multiequations. The results of Sect. 4 demonstrate that this optimization is not available in an implementation of Martelli-Montanari which supports type explanation.

8 Conclusion

We have presented an algorithm for explaining type reconstruction in statically typed languages such as ML and Haskell, where type information is implicit in the program. We expect implicit typing and type reconstruction to play an important role in the design and implementation of scripting languages (such as TCL and Python), which are increasingly popular but are so far untyped. Our modified type inference algorithm can play a useful role in programming environments for such languages.

It is straightforward to extend the algorithm to a type system with circular types [1, 24]. In this case the occurs check is omitted from the unification algorithm, so that type graphs are no longer necessarily acyclic;

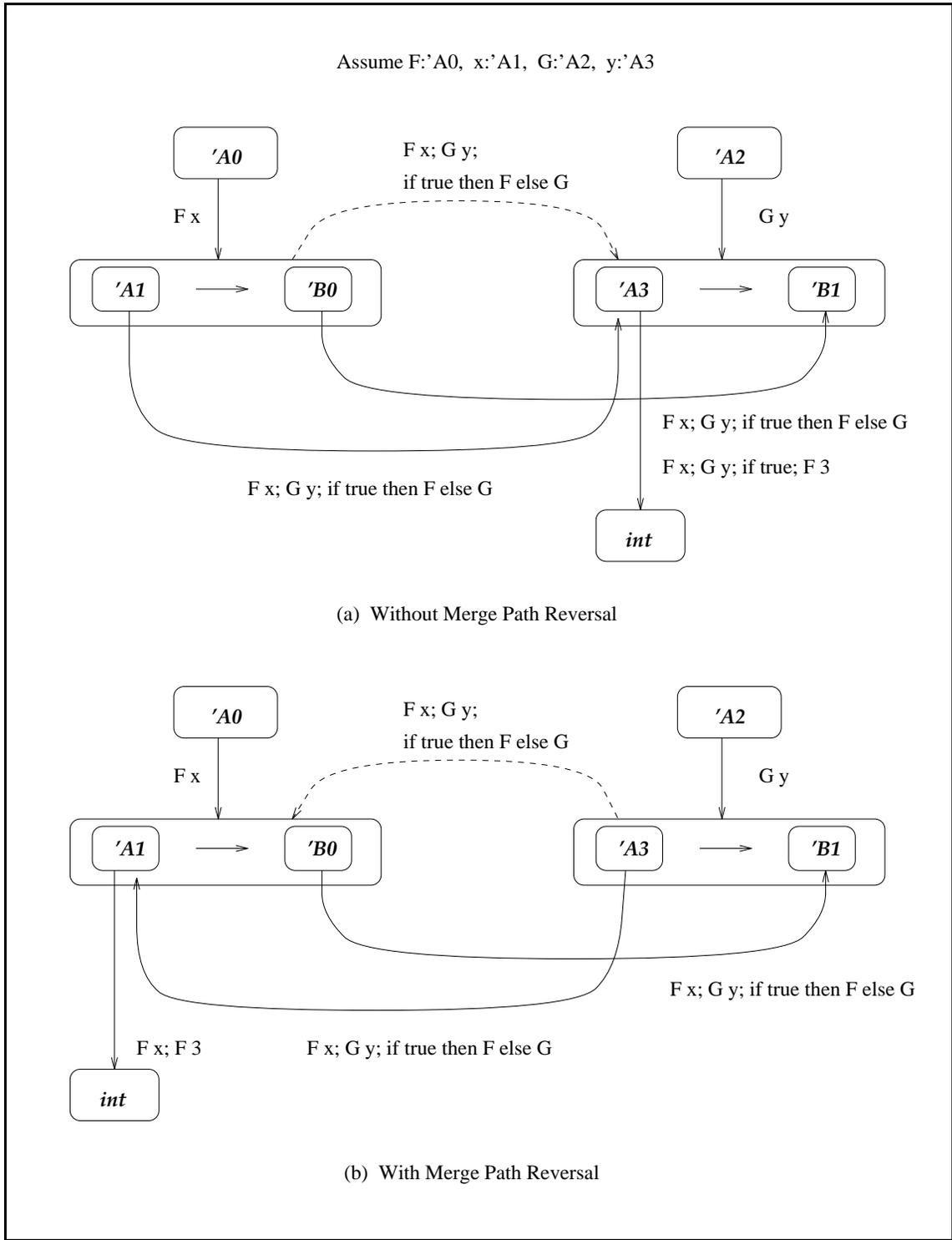


Figure 20: Merge Path Reversal With Baxter/Huet Unification

the unification algorithm must now perform loop-checking while traversing a type graph. As originally observed by Huet [22], the almost-linear algorithm in App. D is easily adaptable to circular types, by omitting the separate pass of the algorithm which performs the cycle check. The previous section demonstrated how this algorithm could be extended to deal with type explanation. On the other hand, circular types have been criticized for masking static typing errors which would be detected by the occurs check in unification [23]. It is not yet generally agreed whether they are worth the extra complication.

It may perhaps be more interesting to try extending this algorithm to Haskell type classes, where the unification algorithm is augmented with a constraint solver for resolving the use of overloaded symbols [15, 35]. As discussed below, based on our experience with implementing the type explanation algorithm in the Standard ML of New Jersey compiler [3, 14], we do not believe that Haskell classes introduce any material complications into our approach to type explanation. In this case overload resolution is an operation which succeeds type inference, ensuring that the type set membership constraints accumulated during type inference are satisfiable [15, 35]. On the other hand, in [37] it is shown that with a sufficiently rich combination of overloading and parametric polymorphism, it might actually be useful to have some type variables instantiated as a result of overload resolution. Extending our algorithm to this situation would lead to an interesting combination of type explanations, where instantiation edges for some type variables would be labelled by explanations for the overload resolutions which lead to those instantiations. The type inference algorithm described by Ophel et al. [37] is implemented in a different compiler (ANU ML extended with parametric overloading) from that used to implement the type explainer (SML/NJ), so a direct extension of our implementation is not possible. This remains a topic for future work.

Another direction to consider for this algorithm is in debugging environments for logic programming languages such as Prolog and CLP(R), where unification drives the computation. In fact the Warren Abstract Machine for efficient Prolog implementations is essentially a compilation of the naive graph unification algorithm of App. A into abstract machine code [2]. It might also be useful to consider extending this approach to explaining unification to higher-order unification algorithms. A practical example would be extending it to β_0 -unification [29]. Qian [43] has described a linear-time algorithm for β_0 -unification, based on the unpublished linear-time first-order unification algorithm due to Martelli and Montanari [27]. Although an efficient and practical implementation of β_0 -unification is still an open research problem [16, 34], one possible avenue to pursue is an adaptation of the more practical almost-linear-time algorithm provided by Martelli and Montanari [28]

Finally, unification is also used in several other tools besides type checkers. The Centaur programming environment [9] compiles specifications of language semantics in natural semantics style [25] into Prolog programs. Since, as noted, Prolog compilation is based on compiling naive graph unification into abstract machine code, our approach is immediately applicable to providing debugging support in such a programming environment. The PSG system [4] is based on checking the static semantics of a program by generating a “context relation⁸” [46] in a bottom-up fashion from an abstract syntax tree. Order-sorted unification is used in a crucial way in this construction. Order-sorted unification is similar in spirit to (although more complicated than) the constraint-solving unification algorithm used in Haskell type inference [36, 35, 15]. So our unification algorithm for type explanation provides a possible extension to the PSG system with debugging support.

A prototype of this algorithm has been implemented in the type checker for the Standard ML of New Jer-

⁸A context relation is a mapping from each vertex of an abstract syntax tree to the set of attributes of that vertex, represented by a (possibly non-ground) term in an order-sorted free algebra.

sey compiler, v0.93 [3, 7]. The design of the algorithm benefited critically from practical experience with this implementation. Based on experience with this prototype, the type explainer has recently been completely reimplemented in the SML/NJ compiler, v1.03f. Modulo some obvious differences of scale, the unification algorithm used in the SML/NJ compiler is exactly the same as the algorithm described in App. A, while our reimplementations of this unification algorithm for type explanation is exactly the same as the algorithm described in App. C. Language features such as weak type variables and equality types do not materially affect the algorithm, and (as noted above) we believe based on this experience that the same is also true for Haskell type classes. The only major complication introduced in unification is the treatment of user-defined type variables, while the major complication with the type inference algorithm is in the generalization operation for polymorphic types. Both of these complications arise because of the need to maintain certain aliasing paths uncompressed during type inference (as explained in some detail in Sect. 4). We provide more details on this and other aspects of the practical implementation and use of this algorithm in a sequel paper [14] This sequel also provides more details of the type explainer, and its continuation-based interface, which we have incorporated into our implementation of type explanation in SML/NJ.

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] Hassan Ait-Kaci. *The Warren Abstract Machine*. The MIT Press, 1991.
- [3] Andrew Appel and David MacQueen. Standard ML of new jersey. In *Proceedings of the Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Springer-Verlag Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1991.
- [4] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive program environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.
- [5] Lewis D. Baxter. An efficient unification algorithm. Technical Report CS-73-23, University of Waterloo, Ontario, Canada, 1973.
- [6] Lewis D. Baxter. *The complexity of unification*. PhD thesis, University of Waterloo, Ontario, Canada, 1976.
- [7] Frederick Bent. An ML type error explanation system. Master’s thesis, University of Waterloo, Waterloo, Ontario, 1994. In preparation.
- [8] H.-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–206. ACM Press, 1989.
- [9] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 14–24, 1988.
- [10] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

- [11] Jacques Corbin and Michel Bidoit. A rehabilitation of Robinson’s unification algorithm. In R. E. A. Mason, editor, *Information Processing: Proceedings of the IFIP World Congress*, pages 909–914, Paris, France, September 1983. IFIP, North-Holland.
- [12] Dennis de Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and Systems Sciences*, 32(32):79–90, 1985.
- [13] Dominic Duggan. Practical subtype inference. In preparation. Earlier version submitted to *ACM Conference on Functional Programming and Computer Architecture*, 1995.
- [14] Dominic Duggan. A type inference explanation facility for ML. In preparation, 1995.
- [15] Dominic Duggan and John Ophel. Kinded parametric overloading. Technical Report CS-94-35, University of Waterloo, Department of Computer Science, September 1994. Supersedes CS-94-15 and CS-94-16, March 1994, and CS-93-32, August 1993.
- [16] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 12, pages 289–325. The MIT Press, 1991.
- [17] Paola Giannini. Type checking and type deduction in polymorphic programming languages. Technical report, Carnegie-Mellon University, 1985.
- [18] Kevin Hammond. Efficient type inference using monads. In *Functional Programming, Workshops in Computing*, pages 146–157, Glasgow, Scotland, August 1991. Springer-Verlag.
- [19] Jacques Herbrand. *Recherches sur la Theorie de la Demonstration (these)*. PhD thesis, Université de Paris VII, 1930.
- [20] Pat M. Hill and John W. Lloyd. The Gödel programming language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, October 1992.
- [21] Paul Hudak, Simon Peyton-Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict purely functional language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.
- [22] Gerard Huet. *Resolutions d’equations dans les langages d’ordre 1,2, . . . , ω* . PhD thesis, Université de Paris VII, Paris, 1976.
- [23] John Hughes. Message on `haskell` electronic mailing list. Available by anonymous ftp from `nebula.systemz.cs.yale.edu`, in directory `/pub/haskell/list-archive`, October 1993.
- [24] Greg F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 44–57. ACM Press, 1986.

- [25] Gilles Kahn. Natural semantics. In F. Brandenburg, editor, *Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Springer-Verlag Lecture Notes in Computer Science*. Springer-Verlag, 1987.
- [26] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.
- [27] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Technical Report B76-16, Ist. di Elaborazione delle Informazioni, Consiglio Nazionale delle Ricerche, Pisa, Italy, 1976.
- [28] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.
- [29] Dale Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
- [30] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [31] Gopalan Nadathur and Frank Pfenning. Types in higher order logic programming. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, 1992.
- [32] Rishiyur S. Nikhil. Practical polymorphism. In *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*, pages 319–333, 1985.
- [33] Tobias Nipkow. Equational reasoning. Unpublished course notes, 1992.
- [34] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of IEEE Symposium on Logic in Computer Science*. IEEE, 1993.
- [35] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 409–418. ACM Press, 1993.
- [36] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*, pages 1–14. Springer-Verlag, 1991. Lecture Notes in Computer Science 523.
- [37] John Ophel, Dominic Duggan, and Gordon Cormack. Parametric overloading and liberal resolution. Technical Report CS-94-06, University of Waterloo, 1994. Submitted for publication.
- [38] John Ousterhout. TCL: An embeddable command language. In *Proceedings of the USENIX Conference*, pages 133–146, Winter 1990.
- [39] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and Systems Sciences*, 16:158–167, 1978.
- [40] Simon Peyton-Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: A technical overview. In *Joint Framework for Information Technology Technical Conference*, Keele, March 1993.

- [41] Simon Peyton-Jones and Philip Wadler. Imperative functional programming. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993. ACM Press.
- [42] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of ACM Symposium on Lisp and Functional Programming*. ACM Press, 1988.
- [43] Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, 1993.
- [44] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–29, 1965.
- [45] Patrick M. Sansom. Time profiling a lazy functional compiler. In *Functional Programming, Workshops in Computing*, Glasgow, Scotland, July 1993. Springer-Verlag.
- [46] Gregor Snelting. The calculus of context relations. *Acta Informatica*, 28:411–445, 1991.
- [47] Helen Soosaipillai. An explanation-based polymorphic type checker for Standard ML. Master’s thesis, Heriot-Watt University, Edinburgh, Scotland, 1990.
- [48] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [49] Robert Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.
- [50] P. Trum and G. Winterstein. Description, implementation and practical comparison of unification algorithms. Technical Report Internal Rep. No. 6/78, Fachbereich Informatik, Universität Kaiserslautern, 1978.
- [51] David A. Turner. Miranda: A nonstrict functional language with polymorphic types. In *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*. Springer-Verlag Lecture Notes in Computer Science, 1985.
- [52] Guido van Rossum and Jelke de Boer. Interactively testing remote servers using the Python programming language. Technical report, CST Department, CWI, August 1993.
- [53] Janet A. Walz. *Extending Attribute Grammar and Type Inference Algorithms*. PhD thesis, Cornell University, Ithaca, New York, January 1989.
- [54] Mitchell Wand. Finding the source of type errors. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 38–43. ACM Press, 1986.

A DAG Unification in ML

This appendix provides a realistic implementation of the unification algorithm as used in most ML and Haskell type-checkers. In this implementation variables are represented as references and substitution is implemented by assignment, as described in Sect. 2. This gives rise to a directed acyclic graph (DAG) representation for types.

```
datatype 'a option = NONE | SOME of 'a
```

```
fun app f [] = ()  
| app f (x::xs) = ( f x; app f xs )
```

```
fun zip [] [] = []  
| zip (x::xs) (y::ys) = (x,y)::(zip xs ys)
```

```
datatype ty = tycon of { tycon:string, tyargs:ty list } | tyvar of (ty option) ref
```

10

```
fun Deref (tyvar (tyv as ref (SOME (ty as tyvar tyv')))) = ( tyv := !tyv'; Deref ty )  
| Deref (tyvar (tyv as ref (SOME ty))) = ty  
| Deref ty = ty
```

```
exception Abort
```

```
fun OccursCheck tyv (tycon { tycon=_, tyargs=types, ... }) = app (fn t => OccursCheck tyv (Deref t)) types  
| OccursCheck tyv (tyvar (tyv' as ref NONE)) = if tyv = tyv' then raise Abort else ()
```

20

```
fun Unify ty1 ty2 = Unify' (Deref ty1) (Deref ty2)
```

```
and Unify' (ty1 as tyvar (tyvar1 as ref NONE)) (ty2 as tyvar (tyvar2 as ref NONE)) =  
  if tyvar1 = tyvar2 then () else tyvar1 := SOME ty2  
| Unify' (ty1 as tyvar (tyvar1 as ref NONE)) (ty2 as tycon _) =  
  ( OccursCheck tyvar1 ty2; tyvar1 := SOME ty2 )  
| Unify' (ty1 as tycon _) (ty2 as tyvar (tyvar2 as ref NONE)) =  
  ( OccursCheck tyvar2 ty1; tyvar2 := SOME ty1 )  
| Unify' (tycon {tycon=tyc1,tyargs=types1,...}) (tycon {tycon=tyc2,tyargs=types2,...}) =  
  if tyc1 = tyc2 then app (fn (t1,t2) => Unify t1 t2) (zip types1 types2) else raise Abort
```

30

B Unification Algorithm Supporting Type Explanation

In this appendix we provide the unification algorithm modified to support type explanation. This algorithm does not properly support polymorphic aliasing (as discussed in Sect. 5). The full algorithm, including support for polymorphic aliasing and with path compression, is given in App. C.

```
datatype Tyvar = Tyvar of { tyexpl:(ty * Tyexplain) option, tyvname:string, progvar:string }
```

```
and Tyexplain = ExplainAlias of ExplainPath * ExplainPath * Absyn  
| ExplainInstan of ExplainPath * Absyn
```

```
and ty = tycon of { tycon:string, tyargs:ty list } | tyvar of Tyvar ref
```

```
withtype ExplainPath = (Tyvar ref) list
```

10

```
fun Deref (tyvar (tyv as ref (Tyvar {tyexpl=Some (ty' ,-),...}))) = Deref ty'
| Deref ty = ty
```

```
fun OccursCheck tyv (tycon {tyargs=types,...}) = app (fn t => OccursCheck tyv (Deref t)) types
| OccursCheck tyv (tyvar (tyv' as ref (Tyvar {tyexpl=None,...}))) = if tyv = tyv' then raise Abort else ()
```

```
(* Build a list of type vars in alias path from root, with prefix history *)
(* addHist : (history:ExplainPath) -> (root:ty) -> ExplainPath *)
```

20

```
(* Bind type var “tyv” to “ty”, with explanation “explain” *)
fun Bind (tyvar (tyv as ref (Tyvar {tyexpl=_,tyvname=A,progvar=x,...}))) ty explain =
  ( tyv := Tyvar {tyexpl=Some(ty,explain),tyvname=A,progvar=x} )
```

```
(* Reverse the direction of edges in alias path rooted at “root” *)
(* ReverseAliasPath : (root:ty) -> unit *)
```

```
fun Unify' ty1 hist1 ty2 hist2 tm = Unify' ty1 (Deref ty1) hist1 ty2 (Deref ty2) hist2 tm
```

```
and Unify' ty1 (tyvar tyvar1) hist1 ty2 (tyvar tyvar2) hist2 tm =
  if tyvar1 = tyvar2 then () else ( ReverseAliasPath ty1; Bind ty1 ty2 (ExplainAlias (hist1,hist2,tm)) )
| Unify' ty1 (tyvar tyvar1) hist1 ty2 (ty2' as tycon _) _ tm =
  ( OccursCheck tyvar1 ty2'; ReverseAliasPath ty1; Bind ty1 ty2 (ExplainInstan (hist1,tm)) )
| Unify' ty1 (ty1' as tycon _) _ ty2 (tyvar tyvar2) hist2 tm =
  ( OccursCheck tyvar2 ty1'; ReverseAliasPath ty2; Bind ty2 ty1 (ExplainInstan (hist2,tm)) )
| Unify' ty1 (tycon {tycon=tyc1,tyargs=types1,...}) hist1 ty2 (tycon {tycon=tyc2,tyargs=types2,...}) hist2 tm =
  if tyc1 = tyc2
  then app (fn (t1,t2) => Unify' t1 (addHist hist1 ty1) t2 (addHist hist2 ty2) tm) (zip types1 types2)
  else raise Abort
```

30

40

```
fun Unify ty1 ty2 tm = Unify' ty1 [] ty2 [] tm
```

C The Full Unification Algorithm With Type Explanation

In this appendix we provide the complete code for the unification algorithm modified for explaining type inference.

```
datatype 'a option = NONE | SOME of 'a
```

```
datatype Tyvar = Tyvar of { tyexpl : ( ty * Tyexplain ) option,
                          tyvname : string,
                          progvar : string,
                          poly     : bool
                        }
```

```

and Tyexplain = ExplainAlias of ExplainPath * ExplainPath * Absyn                                10
|
|           ExplainInstan of ExplainPath * Absyn
|           ExplainPath   of ExplainPath * Tyexplain (* for compressed paths *)

and ty = tycon of { tycon : string, tyargs : ty list } | tyvar of Tyvar ref

withtype ExplainPath = ( Tyvar ref ) list

exception Abort

fun app f [] = () | app f (x::xs) = ( f x; app f xs )                                           20

fun zip [] [] = [] | zip (x::xs) (y::ys) = (x,y)::(zip xs ys)

fun all p l = List.revfold (fn (x,b) => b andalso p x) l true

fun isPolyEdge (tyvar tv) = isPolyEdge' tv
| isPolyEdge _ = false

and isPolyEdge' (ref (Tyvar {tyexpl=SOME (_,expl),poly=true,...})) = isPolyExpl expl
| isPolyEdge' _ = false                                                                    30

and isPolyExpl expl = case expl of
| ExplainAlias ([],[,-] => true
| ExplainInstan ([,-] => true
| ExplainPath (path,expl) => (all isPolyEdge' path) andalso (isPolyExpl expl)
| _ => false

fun Deref' (tyvar (tyv as ref (Tyvar{tyexpl=SOME (ty as tyvar (ref (Tyvar{tyexpl=SOME _,...})), _,...})), poly) =
| Deref' (ty, poly andalso isPolyEdge' tyv)
| Deref' (tyvar (ref (Tyvar{tyexpl=SOME (ty, _),...})), poly) = (ty, poly)
| Deref' (ty, poly) = (ty, poly)                                                            40

fun Deref (tyvar (tyv as ref (Tyvar{tyexpl=SOME (ty', _),...})), poly) = Deref' (ty', poly)
| Deref (ty, poly) = (ty, poly)

fun OccursCheck tyv (tycon{tyargs=types,...}) = app (fn t => OccursCheck tyv (Deref t)) types
| OccursCheck tyv (tyvar (tyv' as ref (Tyvar{tyexpl=NONE,...}))) = if tyv = tyv' then raise Abort else ()

fun Bind (tyvar(tyv as ref(Tyvar{tyexpl=_tyvname=A,progvar=x,poly=C}))) ty explain =
| ( tyv := Tyvar{tyexpl=SOME(ty,explain),tyvname=A,progvar=x,poly=C} )                          50

fun mkAliasExplain (hist1 as (ref(Tyvar{poly=true,...})):_) hist2 tm = mkAliasExplain [] hist2 tm
| mkAliasExplain hist1 (hist2 as (ref(Tyvar{poly=true,...})):_) tm = mkAliasExplain hist1 [] tm
| mkAliasExplain hist1 hist2 tm = ExplainAlias(hist1,hist2,tm)

fun BindAlias hist1 ty1 (ref(Tyvar{poly=poly1,...})) hist2 ty2 tyvar2 tm =
| let val explain = mkAliasExplain hist1 hist2 tm in
|   if poly1 then Bind ty1 ty2 explain else Bind ty2 ty1 explain
| end                                                                                                                                    60

fun mkInstanExplain (hist as (ref(Tyvar{poly=true,...})):_) tm = ExplainInstan([], tm)
| mkInstanExplain hist tm = ExplainInstan (hist, tm)

```

```

fun BindInstan hist1 ty1 ty2 tm = Bind ty1 ty2 (mkInstanExplain hist1 tm)

fun BindTyvar tyv ty explain = Bind (tyvar tyv) ty explain

(* ReverseAliasPath : (root:ty) -> unit *)
fun ReverseAliasPath root =
let
  (* Insert vertices in alias path into list in reverse order *)
  fun path2list list (tyv as tyvar (ref(Tyvar{tyexpl=SOME (nextty,-),..})) = path2list (tyv :: list) nextty
  | path2list list tyv = tyv :: list

  (* Walk along list of vertices, reversing pointers and updating poly alias edges *)
  fun update' tyv _ [] = () (* tyv (was root, now is leaf) will be reset by Bind *)
  | update' tyv NONE ((ty as tyvar(tyv' as ref(Tyvar{tyexpl=SOME(-expl),..})))::tyvs) =
    (* Optional explanation argument is not there *)
    if isPolyEdge ty then
      (* At end of poly alias path whose explanations have to be updated (since the edges on this path are *)
      (* being reversed). The optional explanation argument is initialized to the explanation on this, *)
      (* the first edge of reversed poly alias path. *)
      ( BindTyvar tyv ty expl; update' tyv' (SOME expl) tyvs )
    else
      ( BindTyvar tyv ty expl; update' tyv' NONE tyvs )
  | update' tyv (SOME expl) ((ty as (tyvar tyv'))::tyvs) = (* Optional explanation argument is there *)
    if isPolyEdge ty then
      (* Continue updating explanations on poly alias path *)
      ( BindTyvar tyv ty expl; update' tyv' (SOME expl) tyvs )
    else
      (* We must have just reached the end of the poly alias path *)
      ( BindTyvar tyv ty expl; update' tyv' NONE tyvs )

  fun update ((tyvar(tyv as ref(Tyvar{tyexpl=NONE,..}))) :: tyvs) = update' tyv NONE tyvs
  | update [] = ()
in
  (* Build a list of tyvar vertices on path rooted at "root", in reverse order. Then walk through this *)
  (* list reversing pointers and updating explanations on any poly alias paths which arise. *)
  update (path2list [] root)
end

(* UpdatePolyAliasPath (root:ty) -> (explain:Tyexplain) -> unit *)
fun UpdatePolyAliasPath root explain =
let
  fun update (ty as tyvar (ref (Tyvar{tyexpl=SOME(nextty,-),..})) =
    if isPolyEdge ty then ( Bind ty nextty explain; update nextty ) else ()
  | update _ = ( (* done *) )
in update root end

(* addHist : (history:ExplainPath)->(endOfPath:ty)->(root:ty)->ExplainPath *)
fun addHist history endOfPath root =
let fun addHist' (root as (tyvar (tyv as ref (Tyvar {tyexpl=SOME(ty,explain),..}))) =
  let val explPath = addHist' ty in
    Bind root endOfPath (ExplainPath (explPath, explain));
    tyv :: explPath

```

```

    end
|   addHist' - = []
in
    history @ (addHist' root)
end

fun Unify' (ty1,hist1,poly1) (ty2,hist2,poly2) tm = Unify' (ty1, Deref(ty1,poly1), hist1) (ty2, Deref(ty2,poly2), hist2) tm

and Unify' (ty1, (tyvar tyvar1, poly1), hist1) (ty2, (tyvar tyvar2, poly2), hist2) tm =
  if tyvar1 = tyvar2 then () else
  (
    ReverseAliasPath ty1;
    UpdatePolyAliasPath ty2 (mkAliasExplain hist1 hist2 tm);
    BindAlias hist1 ty1 tyvar1 hist2 ty2 tyvar2 tm
  )
|   Unify' (ty1, (tyvar tyvar1, poly1), hist1) (ty2, ((ty2' as tycon -), poly2), -) tm =
  (
    OccursCheck tyvar1 ty2;
    ReverseAliasPath ty1;
    UpdatePolyAliasPath ty2 (mkInstanExplain hist1 tm);
    BindInstan hist1 ty1 ty2 tm
  )
|   Unify' (ty1, ((ty1' as tycon -), poly1), -) (ty2, (tyvar tyvar2, poly2), hist2) tm =
  (
    OccursCheck tyvar2 ty1;
    ReverseAliasPath ty2;
    UpdatePolyAliasPath ty1 (mkInstanExplain hist2 tm);
    BindInstan hist2 ty2 ty1 tm
  )
|   Unify' (ty1, ((ty1' as tycon{tycon=tyc1,tyargs=types1,..}), poly1), hist1)
      (ty2, ((ty2' as tycon{tycon=tyc2,tyargs=types2,..}), poly2), hist2) tm =
  if tyc1 = tyc2
  then app (fn (t1,t2) => Unify' (t1, addHist hist1 ty1' ty1, poly1) (t2, addHist hist2 ty2' ty2, poly2) tm)
           (zip types1 types2)
  else raise Abort

```

(* Unify : (ty1:ty) -> (ty2:ty) -> (tm:Absyn) -> unit *)

```

fun Unify ty1 ty2 tm = Unify' (ty1, [], true) (ty2, [], true) tm

```

D Baxter/Huet Unification in ML

This appendix provides an implementation of graph unification in ML which is asymptotically superior to the naive graph unification algorithm provided in App. A. In particular, although the latter algorithm has a worst-case exponential time complexity, the algorithm provided here has almost-linear-time worst-case complexity. Duggan [13] considers an extension of this algorithm to provide polynomial-time subtype constraint solving in ML-like languages extended with subtyping. This algorithm was independently developed by Baxter [5, 6] and Huet [22]. The description here is based on that provided by Knight [26].

```

datatype tyvertex = TYCON of { tycon:string, tyargs:ty list } | TYVAR
and ty = TY of { class:ty option ref, ty:tyvertex }

```

```

fun mkvar () = TY { class=ref NONE, ty=TYVAR }

fun isvar (TY { ty=TYVAR,...}) = true (* Assume we are at representative vertex *)
| isvar _ = false

fun Deref (TY { class=tyv as ref (SOME (ty as TY { class=tyv'=ref (SOME _), ... })), ... }) =
  ( tyv := !tyv'; Deref ty )
| Deref (TY { class=ref (SOME ty), ...}) = ty
| Deref ty = ty

fun Unify ty1 ty2 =

  let val stack = ref ([(ty1,ty2)])

    fun CycleCheck ... = (* Standard DFS to check for cycles *)

    fun Unify' () = if null (!stack) then CycleCheck ty1 else
      let val (t1,t2) = hd (!stack)
        val _ = (stack := tl (!stack))
        in Unify' (Deref t1) (Deref t2) (* FIND *)
      end

    and Unify'' (ty1' as TY {class=t1,ty=ty1}) (ty2' as TY {class=t2,ty=ty2}) =
      if t1 = t2 then () else (
        if isvar ty1' then t1 := SOME ty2' else t2 := SOME ty1'; (* UNION *)
        Unify'' ty1 ty2
      )

    and Unify''' (TYCON {tycon=tyc1, tyargs=tys1}) (TYCON {tycon=tyc2, tyargs=tys2}) =
      if tyc1 <> tyc2 then raise Abort else (
        stack := (zip tys1 tys2) @ (!stack);
        Unify' ()
      )
      | Unify''' _ _ = Unify' ()

  in Unify' ()

end

```