Type Inference with Partial Types

Satish Thatte
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor

Abstract

This paper introduces a new form of type expressions which represent partial type information. These expressions are meant to capture the type information statically derivable from heterogeneous objects. The new monotypes form a lattice of subtypes and require type inference based on inclusion constraints. We discuss the existence and form of principle types under this extension and present a semi-decision procedure for the well-typing problem which can be restricted to a form that terminates for most practical programs. The partial type information derivable for heterogeneous entities is not sufficient to guarantee type-correctness for many of their uses. We therefore introduce a notion of statically generated dynamic type checks. Finally, all these elements are pulled together to sketch the derivation of a *static* system for "plausibility checking" which identifies the applications which may require a dynamic check and catches many type errors.

1. Introduction

The use of heterogeneous data structures and functions is common in dynamically typed languages like LISP and contributes significantly to their flexibility. It is very difficult to allow such structures in a static system for the simple reason that there is no way to type, say, a list like *[1,true,['a']]* in such a way that the types of its components can be statically determined later (we use brackets "[]" for both list expressions and types). The standard technique for managing heterogeneity in statically typed languages is the use of variant types, which does not eliminate dynamic checking; it simply shifts the burden of such checks to the programmer as in the following ML declarations:

type intbool == vnum of int | vbool of bool;

val f (vnum n) == n+1 |
    f (vbool b) == if b then 0 else 1;

Pattern-matching in the definition of *f* is used to do something very much like dynamic type-checking. Other functions on *intbool* may expect to see only one pattern, generating a run-time error if the wrong pattern is presented. Of course, this results in programs that are more secure and efficient than dynamically typed ones. However, the burden thus placed on the programmer handicaps statically typed functional languages in attracting users interested in exploratory programming, for whom flexibility is more important than security.

This paper presents a generalization of the Hindley-Milner static type inference system which allows free use of heterogeneous structures while retaining some of the advantages of static typing. We reconsider the well-typing problem for Milner's **Exp** language (Milner, 1978) with additional type expressions representing *partial* type information, which form an inclusion hierarchy similar in some respects to the subtype hierarchies for modeling inheritance. In the first and major part of the paper, we develop the algorithms necessary for type inference with partial types without explicit type declarations. Although fully static typing for a language with heterogeneous structures is clearly impossible, we sketch a system in §6 which uses these algorithms to infer sufficient type information to identify the places where dynamic checks may be required and catches many type errors statically. The system uses the simple observation that when the known (partial) type of the argument in an application is insufficiently defined to guarantee the type-correctness of the application, a dynamic check for the required type may be needed. Static errors arise whenever there is an incompatibility between the known and required types in such situations. The system has an obvious application as a debugging aid and may be useful in compilation.

2. Related Work

Heterogeneity is supported in a somewhat different way by systems with labeled sums (Cardelli, 1984; Wand, 1987). These systems are still far from matching the flexibility of LISP since they cannot support run-time decisions on the choice of components, which, in their setting, would require labels to be first-class values. Type inference for coercions using a notion of inclusion has been discussed by Reynolds (1985). Cardelli (1984) uses subtypes to model inheritance, and Cardelli-Wegner (1985) give an overview of a system combining let-polymorphism with subtypes and existential quantification over types. Both papers use a framework resembling second-order (explicitly) typed λ-calculus. Our system is most closely related to the framework developed by Mitchell (1984) and further by Fuh-Mishra (1987) where the general problem of *implicit* typing in the presence of inclusion relationships among monotypes is considered, and algorithms are developed for the case where such relationships are confined to *atomic* types (e.g., car$\leq$vehicle). We use essentially the same framework as theirs, but instead of predeclared atomic inclusions, we consider general inclusions determined by quantity of structural type information.

3. Partial types

The main difficulty in assigning types to heterogeneous data structures is in describing the component type. One approach is to use manifest sums like *int+bool* which are similar to variant types like *intbool* except for the lack of injection labels. The only advantage of manifest sums over a type expression representing "no information" is that more errors can in principle be statically discovered with them. This possible use is balanced against the fact that they are expensive and often impossible to keep track of, especially in a lazy environment. For example, no finite sum can describe the component type of the list *lst* defined as:

$$lst == cons(1, (map\ (\lambda x.[x])\ lst))$$

We therefore use a single universal sum type Ω, which corresponds to the type-scheme $\exists\alpha.\alpha$ denoting the set of all values not involving *wrong*. Instead of the usual *flat* set of monotypes, we now get a lattice of monotypes generated by Ω as a new basic monotype and the least element in the *approximation* order "$\sqsubseteq$". Any expression that does not go wrong has type Ω, i.e., Ω represents no type information other than the lack of type-error. The approximation order on monotypes (ranged over by μ) is generated by extending the base case $\Omega \sqsubseteq \mu$ monotonically over type constructors, with the exception of the function-space constructor $\rightarrow$ for which it is *anti-monotonic* in the first argument — a natural consequence of the semantic interpretation of types as sets of values.

Since monotypes are no longer disjoint, a value belongs to many monotypes in general. For instance, *(1,[True])* belongs to both $int \times [bool]$ and $\Omega \times [\Omega]$ among others. The former is the best type but it is often necessary to coerce the expression to the latter approximation. Consider *cons(1,[true])*. Given the usual typing $cons{:}\forall\alpha.\alpha \times [\alpha] \rightarrow [\alpha]$, the way to type the application is to coerce *(1,[true])* to $\Omega \times [\Omega]$, and use the generic instance $\Omega \times [\Omega] \rightarrow [\Omega]$ of the type of *cons*, yielding the type $[\Omega]$ for the entire expression. Typically, Ω appears in the type of an expression when the possible values of the corresponding part of the expression have no structure in common.

Conventional coercions occur between types that are structurally distinct (e.g., integer and real) and often involve a change of representation via a coercion function. The relationship between, say, $[int]$ and $[\Omega]$ is rather different. The type $[\Omega]$ is simply less specific than $[int]$ and therefore applies to a larger set of values. The relationship between $[int]$ and $[\Omega]$, thought of as a coercion, would be stated as the *inclusion* $[int] \leq [\Omega]$. It is obviously possible to turn the lattice of monotypes upside down by using an inclusion ordering, with Ω at the top element, which would make it compatible with other type systems based on type inclusions (Cardelli, 1984; Mitchell, 1984). We find it more natural to use the approximation ordering based on "quantity" of type information, and denote the same relationship by $[\Omega] \sqsubseteq [int]$, and call such a pair an approximation rather than a coercion.

As in some other systems with coercions to supertypes (Mitchell, 1984; Fuh-Mishra, 1987), all type information inferable for a given expression cannot in general be represented in the form of a single type expression. In other words, expressions under the regime of partial types don't have principal types in the usual language of type expressions and schemes (Damas-Milner, 1982). For example, we would like *(λx.cons(1,x))[true]* to have the type $[\Omega]$, and *(λx.cons(1,x))[2]* to have the type $[int]$. The type of the λ-abstraction needs to be $[\Omega] \to [\Omega]$ in the first case and $[int] \to [int]$ in the second. The usual way to get both typings would be to either derive them through instantiation or coercion from a principle type. All instances of an expression like $[\alpha] \to [\alpha]$ are clearly not sound for the λ-expression, and coercion does not help either because, due to the anti-monotonicity of $\to$ in its first argument, the join of the two required types is $[\Omega] \to [int]$ which is also unsound for the λ-expression.

The uses of *λx.cons(1,x)* above can be accounted for by instances of the constrained type $\{[\alpha] \to [\alpha] \mid \alpha \sqsubseteq int\}$ which can be thought of as the principle type of the expression. The difference between instantiating α to *int* and using *int* as the upper bound for α is that in the latter case, the type is still capable of instantiations corresponding to all generic instances of the type of *cons* compatible with its present application. Typing rules have to accommodate the need for such constraints by including constraint assumptions for free type variables in addition to the usual typing assumptions for free program variables.

4. Typing Rules

In the following, the notation, and the syntax and semantics of type and value expressions are similar to those of Damas-Milner (1982). As usual, μ will range over monotypes, τ over types, σ over type-schemes (i.e., possibly quantified types), and $\sigma \geq \tau$ means τ is a generic instance of σ. The notation $FV(\sigma)$ (or $FV(\tau)$) is used to denote the set of free variables in a type expression σ (or τ). We begin with typing rules for the plain (monomorphic) λ-calculus and then consider the extension for let-bound polymorphic identifiers. The only new element in the typing rules for the plain λ-calculus is the introduction of constraints on free type variables, with typing rules of the form $C, A \vdash e : \tau$ where C is a *set* of constraints of the form $\tau_1 \sqsubseteq \tau_2$. This form of typing rules is not new (see, e.g., (Mitchell, 1984)). We write $C \Vdash C'$ to mean that the set C *entails* the set C' in the sense that any model (solution) for C is also a model for C'. The symbol $\Vdash$ is used instead of $\vdash$ to emphasize that the relation is not expected to be decidable in general. If $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$ then we write $C_1 \equiv C_2$.

A *model* or *solution* for a set C is a substitution S mapping its free variables in such a way that SC contains only valid assertions. This relationship is denoted by $S \models C$. An assertion is *valid* if all its ground instances hold for the lattice of monotypes. A set C is *consistent* if it has a solution. Unlike sets of first-order equality constraints which arise in the Hindley-Milner system, consistent sets of approximation constraints for partial types often do not have *most general* solutions. If they did, principle partial types would be expressible without constraints.

The rules for deriving partial types for λ-expressions are straightforward extensions of the standard rules (we assume the existence of constants for elimination of tupling):

TAUT: $C, A \vdash x : Ax$

ABS: $$\frac{C, A[x{:}\tau'] \vdash e : \tau}{C, A \vdash \lambda x.e : \tau' \to \tau}$$

TUP: $$\frac{C, A \vdash e : \tau \qquad C, A \vdash e' : \tau'}{C, A \vdash e, e' : \tau \times \tau'}$$

COMP: $$\frac{C, A \vdash e : \tau' \qquad C, A \vdash e' : \tau'' \qquad C \Vdash \tau'' \to \tau \sqsubseteq \tau'}{C, A \vdash e\, e' : \tau}$$

In order to avoid the possibility of multiple typing-derivations for the same expression, we permit coercions implicitly in the COMP rule via the condition $C \Vdash \tau'' \rightarrow \tau \sqsubseteq \tau'$, instead of providing a separate rule for coercion. The presence of this condition makes these rules non-effective. However, in practice one is interested in starting with an environment A and an expression e and infering both a type τ and a set of constraints C under which the typing relationship $C, A \vdash e : \tau$ holds. In the next section, we give a simple algorithm for this purpose which is *complete* for these rules.

Soundness of the typing rules amounts to the implication $C, A \vdash e : \tau \Rightarrow C, A \models e : \tau$, where the semantic assertion $C, A \models e : \tau$ is equivalent to

$$(\forall S \models C)(\forall \eta \in \mathbb{T}(SA))(\mathbb{E}[\![e]\!]\eta \in \mathbb{T}(S\tau))$$

$\mathbb{T}$ denotes the semantic function mapping monotypes and closed type schemes to sets of values — extended pointwise for sets of type assumptions, and $\mathbb{E}$ maps expressions to their values given an environment η for values of free identifiers. Of these, $\mathbb{E}$ is standard (see, e.g., Milner, 1978), and $\mathbb{T}$ is defined below after the introduction of quantified types. In the definition for "$\models$", SA and $S\tau$ are assumed to be closed; otherwise, the corresponding assertion must be true for all their closed instances.

These rules allow us to derive partial types for many expressions which are not well-typed under the usual rules. Aside from heterogeneous expressions like *cons(1,[true])*, these include expressions such as $\lambda x.xx$ and *(λf.λa,b.(f a),(f b))(λx.x)(1,true)*. The latter is of interest because it was cited by Milner (1978) as an example of the limitations of his system. The (most general) typing statements derivable for these two expressions are:

Example 1

$\{\alpha_1 \sqsubseteq \alpha_1 \rightarrow \alpha_2, \beta \sqsubseteq \alpha_2\}, \emptyset$
$\quad \vdash \lambda x.xx : (\alpha_1 \rightarrow \alpha_2) \rightarrow \beta$

$\emptyset$, {*1:int, true:bool*}
$\quad \vdash$ *(λf.λa,b.f a, f b)(λx.x)(1,true)*: (Ω, Ω)

Some of the details for these derivations are given in later examples. A typing statement of this form can be derived for *any* λ-expression. This may create the misleading impression that all expressions are well-typed in our sense. We restrict the notion of well-typing to those expressions for which a typing statement with a *consistent* set of constraints can be derived. The restriction can be motivated semantically by observing that well-typing normally implies the absence of type errors, which can be ensured (assuming the typing rules are sound) only if C is consistent, since the assertion $C, A \models e : \tau$ is vacuous for an inconsistent C. Note that the constraint sets in both cases in Example 1 are consistent ($\alpha_1 \sqsubseteq \alpha_1 \rightarrow \alpha_2$ is valid with, e.g., $\alpha_1 = \Omega$), so these expressions are well-typed. In contrast, although typing statements can be derived for the classic unsolvable term *(λx.xx)(λx.xx)* and the Y combinator *λf.(λx.fxx)(λx.fxx)*, the constraint sets involved do not have finite solutions. Expressions involving such complex self-applications are among those that create the most serious difficulties for our system, leading to the possibility of nontermination in the constraint resolution procedure. Although more λ-expressions (like $\lambda x.xx$) are well-typed in our system than in that of Hindley-Milner, we believe that the strong normalization property holds for well-typed expressions in our case as well, though we have no proof yet.

In order to accommodate polymorphic identifiers, we will need quantified types as usual, with two differences. We will need to *simultaneously* quantify sets of variables, and we will need to *constrain* the quantified variables. Let the *constrained scheme* $\sigma = \forall \{\alpha_1, \ldots, \alpha_k\}^C.\tau$ denote the quantification of the set of variables $\{\alpha_1, \ldots, \alpha_k\}$ constrained by C over the (unquantified) type τ. Whenever the set of constraints in a scheme is empty, we omit it from the notation. When a substitution is applied to such a σ, it instantiates the unquantified variables in C as well as τ. The need for constrained quantification is clear, given that type variables are usually constrained. The need for simultaneous quantification arises from the fact that consistent circular constraints are possible as in the $\lambda x.xx$ example above. Since quantification applies to constraints, one would want to topologically sort the variables for sequential quantification which is not possible if the circularities are indirect (mutual).

A *generic instance* of σ is a pair (C',τ'), where $C'=SC$ and $\tau'=S\tau$ for some substitution S mapping variables $\alpha_1,\ldots,\alpha_k$. Following Damas-Milner (1982) we shall denote this relationship by $(C',\tau')\leq\sigma$. The *most general* generic instance of a scheme is obtained by using the INST operation, which is defined as:

$$\begin{aligned}\mathrm{INST}(\forall\{\alpha_1,\ldots,\alpha_k\}^C.\tau) &= (\emptyset,\tau),\text{ if } k=0\\ &= (SC,S\tau),\text{ otherwise}\\ &\quad\text{where } S\alpha_i=\beta_i,\text{ and all }\beta_i\text{ are new}\end{aligned}$$

The inverse operation of *generification* is denoted by GEN; $\mathrm{GEN}(C,A,\tau) = \forall\{\alpha_1,\ldots,\alpha_k\}^C.\tau$ where $\{\alpha_1,\ldots,\alpha_k\} = FV(\tau)-FV(A)$ as usual. In formulating typing rules for polymorphic identifiers, we follow the variant of the system of Damas-Milner (1982) given by Clement, *et al* (1986), which produces normalized typing derivations. Along with a new LET rule, we need to replace the TAUT rule for identifiers with the SPEC rule, as follows:

$$\text{SPEC: }\frac{(C',\tau)\leq Ax \qquad C\Vdash C'}{C, A\vdash x:\tau}$$

$$\text{LET: }\frac{\begin{array}{l}C,\ A\vdash e:\tau'\\ C,\ A[x{:}\mathrm{GEN}(C,A,\tau')]\vdash e':\tau\end{array}}{C, A\vdash \text{let } x=e \text{ in } e':\tau}$$

Example 2.

$\{\beta\sqsubseteq int\}, A\vdash \lambda x.cons(1,x):[\beta]\to[\beta]$
  with $A\supseteq\{1{:}int,\ cons{:}\forall\alpha.\alpha\times[\alpha]\to[\alpha]\}$

$\emptyset$, A
  $\vdash$ *let* $\lambda x.cons(1,x)$ *in* $f\,[1], f\,[true]:([int],[\Omega])$
$\emptyset, A\vdash(\lambda x.cons(x,x))[1]{:}[\Omega]$
 follow if in addition $A\supseteq\{[1]{:}[int], [true]{:}[bool]\}$
  treating *[1]* and *[true]* as identifiers for brevity.

We now outline the semantics of type expressions. The domain *V* of semantic values must satisfy the equation (given that integers and booleans are the basic values; other basic domains can be easily added)

$$V\cong I+B+V\times V+V\to V+W$$

where *I* and *B* are the flat domains of integers and booleans, respectively, "$\cong$" represents domain isomorphism, "+" is disjoint (injected) union, "$\to$" is function space, and *W* is a one point domain containing the special value *wrong* which represents type error. Scott (1982) has shown that such equations have least solutions with nice properties, and one can give a semantics for types in which types denote ideal subsets of sets like Scott's solution for *V*. This is also the idea in the equations for $\mathbb{T}$ (which maps closed type expressions and schemes to their meanings) below:

$$\begin{aligned}&\mathbb{T}(int)=in_I(I)\\&\mathbb{T}(bool)=in_B(B)\\&\mathbb{T}(\mu\times\mu')=in_\times(\mathbb{T}(\mu)\times\mathbb{T}(\mu'))\\&\mathbb{T}(\mu\to\mu')=in_\to\{f\in V\to V\mid \mathbb{T}(\mu')\supseteq f(\mathbb{T}(\mu))\}\\&\mathbb{T}(\forall\{\alpha_1,\ldots,\alpha_k\}^C.\tau)\\&\quad=\bigcap\{\mathbb{T}(\mu)\mid C',\mu\leq\forall\{\alpha_1,\ldots,\alpha_k\}^C.\tau\\&\qquad\qquad\text{and } C' \text{ is valid}\}\end{aligned}$$

where in_I, etc., denote the injections needed to "lift" values from the argument domains to the domain *V*. It is convenient to extend $\mathbb{T}$ pointwise to sets of type assumptions A whenever Ax is *closed* for each identifier x in the domain of A. In this case, $\mathbb{T}(A)$ is a set of *environments* (functions from identifiers to values) with the same domain as A. Specifically,

$$\mathbb{T}(A)=\{\eta\mid\forall x.\eta(x)\in\mathbb{T}(Ax)\}$$

Given these definitions, it is fairly routine to prove that the typing rules SPEC, ABS, TUP, COMP, and LET form a sound system. We omit the proof for lack of space.

Combining polymorphic identifiers with inclusion constraints is often considered problematic (Mitchell, 1984; Reynolds, 1985). The problem arises only if quantification over types must be unconstrained. The difficulty is in finding most general solutions for constraint sets. With *constrained* quantification, it is easy to introduce polymorphic identifiers since constraints can be copied along with the type expression during generic instantiation as we do above. This is only one step short of actually copying the expression named in a let binding in place of the name in the relevant scope. In other words, if the requirements of neatness and efficiency are separated from the question about the possibility of typing, the difficulty is seen to be in the former. The sparer the form in which the constraints are maintained, the more efficiency one attains. The height of austerity is the complete elimination of constraints as they arise, as in the incremental Hindley-Milner algorithm. As we shall see, although approximation constraints on partial types cannot be eliminated, they can be maintained in a relatively spare "normal form" in which all constraints are in the form of upper bounds on variables.

5. Type Inference

The problem of type inference is to find the principle type of an expression e under a set of typing rules given a set A of typing assumptions about free identifiers. In our case this means finding a type expression and a set of (consistent) constraints on free type variables which together constitute the principle type in a sense to be made precise below. Generalizing a point made by Wand (1987), it is possible to think of this problem as a sequence of three distinct phases:

1. Find a most general typing τ for e along with a set C of *verification conditions* which must hold for the typing to be correct.
2. Check if C is a consistent (i.e., solvable) set.
3. Find a most general solution for C in the form of a substitution S, and apply S to A and τ.

Following Milner (1978), it is customary to present the inference algorithm as an "Algorithm W" which incrementally solves all three subproblems using unification. This is no doubt the most efficient approach since it avoids any accumulation of constraints and is therefore able to make do with unconstrained quantification. Since it is impossible to eliminate all approximation constraints anyway, we find it more convenient to separate the phases in the initial development of our algorithm.

5.1 Algorithm Type

Phase 1, embodied in Algorithm **Type** below, is easy since there is only one skeleton of a typing derivation for any expression. **Type** resembles Wand's (1987) "action rules" except that the verification conditions are approximation rather than equality constraints.

```
Type(A,e) = Case e of
  x: INST(Ax)
  $\lambda x.e_1$:
     $C, \beta \to \tau$
     where $C, \tau$ = Type(A[x:$\beta$],$e_1$)
  $e_1, e_2$:
     $C_1 \cup C_2, \tau_1 \times \tau_2$
     where $C_1, \tau_1$ = Type(A,$e_1$)
           $C_2, \tau_2$ = Type(A,$e_2$)
  $e_1 e_2$:
     $C_1 \cup C_2 \cup \{\tau_1 \sqsupseteq \tau_2 \to \beta\}, \beta$ — new $\beta$
     where $C_1, \tau_1$ = Type(A,$e_1$)
           $C_2, \tau_2$ = Type(A,$e_2$)
  let x=$e_1$ in $e_2$:
     $C_1 \cup C_2, \tau_2$
     where $C_1, \tau_1$ = Type(A,$e_1$)
       $C_2, \tau_2$ = Type(A[x:GEN($C_1$,A,$\tau_1$)], $e_2$)
```

The types and constraints infered by **Type**, as in Example 3, are nowhere near as neat as those in Examples 1 and 2.

Example 3.

Type($\emptyset$, $\lambda x.xx$) = $\{\alpha \sqsupseteq \alpha \to \beta\}, \alpha \to \beta$

Type($\emptyset$,$(\lambda x.xx)(\lambda x.xx)$) = C, γ
where C = $\{\alpha_1 \sqsupseteq \alpha_1 \to \beta_1, \alpha_2 \sqsupseteq \alpha_2 \to \beta_2,$
$\alpha_1 \to \beta_1 \sqsupseteq (\alpha_2 \to \beta_2) \to \gamma\}$

Type($\emptyset$, $\lambda f.\lambda a,b f\, a, f\, b$) =
C, $\alpha_1 \to \alpha_2 \times \alpha_3 \to \gamma_1 \times \gamma_2$
where C = $\{\alpha_1 \sqsupseteq \alpha_2 \to \gamma_1, \alpha_1 \sqsupseteq \alpha_3 \to \gamma_2\}$

Type(A,$\lambda x.cons(1,x)$) = C, $\beta \to \gamma$
where C = $\{\delta \times [\delta] \to [\delta] \sqsupseteq int \times \beta \to \gamma\}$
and A is the same as in Example 2.

The sets of constraints in Example 3 are not obviously consistent (in the case of the second expression they are inconsistent), nor are they simplified to the extent possible. As we remarked after Example 1 above, a typing statement can be derived in our system for any expression whatsoever, and therefore **Type** always succeeds whether or not the expression given contains type errors. It does have the property of being *correct* and *complete* with respect to the typing rules, as made precise in the following theorems. In both the following proofs, essential use is made of the *normalization* property of the typing rules, which derives from the fact that there is exactly one typing rule for each syntactic construct. The antecedents of any typing statement in a typing proof are therefore unique, at least in form.

Theorem 1 (Correctness of Type).
If **Type**(A,e) = C, τ, then C, A $\vdash$ e : τ.
Proof. Easy by induction on the structure of e. ∎

Another way of understanding the completeness theorem below is that **Type** computes a principle type for the given expression. Any other type derivable for the expression is a more constrained instance of the principle type. This is the definition of principle type in our context. Unlike the Hindley-Milner system, our principle types are clearly not unique up to renaming due to the variety of ways the constraint component may be presented, which may "hide" more or less of the solution. It is possible to define principle types in such a way that they have the uniqueness property, but they are not computable in that form.

Theorem 2 (Completeness of Type). If A' is a substitution instance of A, C', A' $\vdash$ e : τ', and **Type**(A,e) = C, τ, then there is a substitution S such that A' = SA, C' $\Vdash$ SC and τ'=Sτ.
Proof. By induction on the structure of e. The cases where e = x, e = e_1, e_2 and e = λx.e_1 are easy given proof normalization. The e = let x=e_1 in e_2 case is non-trivial but not hard and we leave it to the reader for lack of space. Let e = e_1 e_2, C = $C_1 \cup C_2 \cup \{\tau_1 \sqsupseteq \tau_2 \rightarrow \beta\}$, and $\tau = \beta$. The required antecedents are (i) C', A' $\vdash$ e_1: τ_1', (ii) C', A' $\vdash$ e_2: τ_2', and (iii) C' $\Vdash$ $\tau_2' \rightarrow \tau' \sqsubseteq \tau_1'$. Applying the inductive assumption to (i) and (ii) we know that there are S_1 and S_2 such that S_1A = A' = S_2A, C'$\Vdash$ S_1C_1, C'$\Vdash$ S_2C_2, and $S_1C_1 \Vdash S_1\tau_1 \sqsupseteq \tau_1'$, $S_2C_2 \Vdash S_2\tau_2 \sqsupseteq \tau_2'$. Clearly, S_1 and S_2 agree on variables in FV(A). By inspection of the body of **Type**, it is obvious that the only variables common between FV(C_1)$\cup$ FV(τ_1) on the one hand and FV(C_2)$\cup$FV(τ_2) on the other, are in FV(A). Moreover, the variables each S_i (i=1,2) *needs* to map are in FV(C_i)$\cup$FV(τ_i)$\cup$FV(A). The two substitutions therefore agree on the variables common to their domains, and S = $S_1 \cup S_2 \cup [\tau'/\beta]$ is well-defined. We therefore have SA=A', and Sτ = Sβ = τ'. It remains to show that C' $\Vdash$ SC = $SC_1 \cup SC_2 \cup S\{\tau_1 \sqsupseteq \tau_2 \rightarrow \beta\}$. We already know that C'$\Vdash$ SC_1, C'$\Vdash$ SC_2, by the inductive assumptions and the fact that $SC_1 = S_1C_1$ and SC_2=S_2C_2. We still need to show that C' $\Vdash$ $S\{\tau_1 \sqsupseteq \tau_2 \rightarrow \beta\}$ = $S\tau_1 \sqsupseteq S\tau_2 \rightarrow \tau'$. Since $S_1C_1 \Vdash S_1\tau_1 \sqsupseteq \tau_1'$ and $S_2C_2 \Vdash S_2\tau_2 \sqsupseteq \tau_2'$, we have (1) C'$\Vdash$ $S\tau_1 \sqsupseteq \tau_1'$ and (2) C'$\Vdash$ $S\tau_2 \sqsupseteq \tau_2'$ by the transitivity of $\Vdash$, and the fact that $S\tau_1$=$S_1\tau_1$ and $S\tau_2$=$S_2\tau_2$. The rest follows from facts (1) and (2), antecedent (iii) (C' $\Vdash$ $\tau_2' \rightarrow \tau' \sqsubseteq \tau_1'$), together with the transitivity of $\sqsupseteq$. ∎

5.2. Solving Constraint Sets

Unlike Phase 1 of the type inference process, Phase 2 (consistency of constraint sets) is harder for approximation than for equality constraints. Recall that our goal is to derive algorithms for checking consistency and for simplification rather than elimination of constraint sets. We have not been able to discover a *complete* decision procedure for consistency, and suspect that it does not exist. We do have a *semi*-decision procedure that is guaranteed to terminate with success for consistent sets, and will either halt with failure or loop for inconsistent ones. We offer some suggestions for improving the termination behavior in §5.3.

The procedure developed below checks for consistency by attempting to simplify the given constraint set to a "normal form". A set C of approximation constraints is said to be in *normal form* if each constraint in C is of the form $\alpha \sqsubseteq \tau$. The τ in this case is said to be an upper bound on α and there are in general several such upper bounds on each variable. It is easy to see that *any* constraint set in normal form is consistent since a substitution mapping all variables to Ω will make such a set valid. Constraints specifying lower bounds do not appear in constraint sets in normal form, although they obviously arise in practice. The term normal form suggests that such constraints can be eliminated without loss of information. This process, analogous to instantiation of variables in unification, is embodied in the procedure **Eliminate** given below. One of the difficulties in defining **Eliminate** is the fact that a constraint $\alpha \sqsupseteq \tau$ may be solvable even if α occurs in τ. This is a consequence of the antimonotonicity of the $\rightarrow$ operator in its first argument. For example, $\alpha \sqsupseteq \alpha \rightarrow int$ has $[\Omega \rightarrow int/\alpha]$ and $\alpha \sqsupseteq (\alpha \rightarrow int) \times bool$ has $[(\Omega \rightarrow int) \times bool/\alpha]$ as the simplest solution. It is therefore necessary to precisely define the kind of circular occurrence which destroys solvability. This is the idea captured in the definition of *monotonic* occurrence below.

In order to define monotonic occurrences, we need a precise notion of *occurrences* of subexpressions in type expressions. An occurrence is a string of integers specifying a path to the subexpression concerned. If p is an occurrence within τ then τ/p will denote the subexpression occurring at p. The empty string Λ will reach the expression itself, i.e., $\tau/\Lambda=\tau$. If $\tau = \tau_1 \times \tau_2$ or $\tau_1 \rightarrow \tau_2$, then $\tau/$ '1' $= \tau_1$ and $\tau/$'2' $= \tau_2$, and $\tau/$'21' $= \tau_2/$'1', etc. The idea is the same as in term rewriting, with type operators and constants playing the role of function symbols. The concatenation of occurrences p and q is denoted by p·q.

Definition (monotonic occurrences). A monotonic occurrence p in τ is one which satisfies one of the following conditions:

(i) $p = \Lambda$

(ii) $\tau = \tau_1 \times \tau_2$, q is a monotonic occurrence in τ_1 or τ_2, and p = '1'·q or '2'·q, resp.

(iii) $\tau = \tau_1 \rightarrow \tau_2$, q is a monotonic occurrence in τ_2, and p = '2'·q

In such a case, we shall also say that p is a monotonic occurrence of τ/p in τ, and for short, p is monotonic in τ.

Proposition 3. If $\mu \sqsupseteq \mu'$ and p is monotonic in μ' then p is also monotonic in μ and $\mu/p \sqsupseteq \mu'/p$.

Proof. Easy by induction on the structure of μ'. ∎

Lemma 4. The constraint $\alpha \sqsupseteq \tau$ is not solvable if there is a monotonic occurrence of α in τ and $\alpha \neq \tau$.

Proof. Suppose there is a monotonic occurrence p of α in τ and $\alpha \neq \tau$ (i.e., $p \neq \Lambda$). Suppose for contradiction that there is a substitution S such that $S\alpha = \mu$, $S\tau = \mu'$ and $\mu \sqsupseteq \mu'$. Since instantiation preserves monotonic occurrences, p is monotonic in μ'. By Proposition 3, $\mu/p \sqsupseteq \mu'/p$, and p is monotonic in μ. However, since $\tau/p=\alpha$, we have $\mu'/p=\mu$. We therefore have $\mu/p \sqsupseteq \mu$, and p is monotonic in μ. Applying Proposition 3 again, $(\mu/p)/p \sqsupseteq \mu/p$ and p is monotonic in μ/p. This construction can obviously be carried on ad infinitum which is impossible with a finite μ. ∎

In the following, $\mathbb{1}$ denotes the identity substitution.

```
Eliminate($\alpha \sqsupseteq \tau$)
 if there is a monotonic occurrence of $\alpha$ in $\tau$
   and $\alpha \neq \tau$ then fail
 else Case $\tau$ of
   $\Omega$:  return $(\mathbb{1},\emptyset)$
   atomic:  return $([\tau/\alpha],\emptyset)$
   variable:  return $(\mathbb{1},\{\tau \sqsubseteq \alpha\})$
   $\tau_1 \times \tau_2$:  let $S_0 = [\beta_1 \times \beta_2/\alpha]$,  new $\beta_1$ and $\beta_2$
            let $(S_1,C_1) =$ Eliminate($\beta_1 \sqsupseteq S_0\tau_1$)
            let $(S_2,C_2) =$ Eliminate($\beta_2 \sqsupseteq S_1S_0\tau_2$)
            return $(S_2S_1S_0,\ S_2C_1 \cup C_2)$
   $\tau_1 \rightarrow \tau_2$:  let $S_0 = [\beta_1 \rightarrow \beta_2/\alpha]$,  new $\beta_1$ and $\beta_2$
             let $(S_2,C_2) =$ Eliminate($\beta_2 \sqsupseteq S_0\tau_2$)
             return $(S_2S_0,\ \{\beta_1 \sqsubseteq S_2S_0\tau_1\} \cup C_2)$
```

Example 4.

Eliminate($\alpha \sqsupseteq \alpha \times (\beta \rightarrow \text{int}) \rightarrow (\beta \rightarrow \text{bool})$) = S, C

where $S = [\beta_1 \rightarrow (\beta_{21} \rightarrow \text{bool}) / \alpha]$

$C = \{\beta_1 \sqsubseteq (\beta_1 \rightarrow (\beta_{21} \rightarrow \text{bool})) \times (\beta \rightarrow \text{int}),$
$\beta_{21} \sqsubseteq \beta\}$

It is not quite obvious that **Eliminate** always terminates, since the size of arguments does not necessarily decrease in the recursive calls of **Eliminate**. The halting measure is the *weight* of τ, which is the total number of monotonic occurrences (of *all* subexpressions) in τ.

Lemma 5. Eliminate($\alpha \sqsupseteq \tau$) terminates.

Proof. All we need to show is that the weight of the second argument decreases in all recursive calls within the call Eliminate($\alpha \sqsupseteq \tau$). The conclusion follows easily in all cases from the fact that there is no monotonic occurrence of α in τ. ∎

Proposition 6. Eliminate($\alpha \sqsupseteq \tau$) succeeds iff either there is no monotonic occurrence of α in τ or $\alpha = \tau$.

Proof. Easy with the observation that whenever the condition in the if part holds for the original call, it holds for all recursive calls as well. ∎

The observation in the proof above can be used to improve the efficiency of Eliminate to being linear in proportion to the *weight* of its second argument.

Proposition 7. The substitution returned by a call Eliminate($\alpha \sqsupseteq \tau$) does not map any variables except α and new variables generated during the call.

Proof. Obvious by inspection of Eliminate. ∎

Lemma 8. If Eliminate($\alpha \sqsupseteq \tau$) succeeds and returns (S,C) then C is in normal form.

Proof. The only non-trivial situation is the set S_2C_1 returned as part of the answer in the $\tau_1 \times \tau_2$ case. The proof rests on Proposition 7 together with the claim that C_1 does not contain constraints of the form $\beta_2 \sqsubseteq \tau'$. The only way the claim might be false is if a call of the form Eliminate($\beta' \sqsupseteq \beta_2$) occurs in the course of the call Eliminate($\beta_1 \sqsupseteq S_0\tau_1$). This possibility is precluded because there are no monotonic occurrences of α in τ. ∎

Lemma 9. If Eliminate($\alpha \sqsupseteq \tau$) succeeds and returns (S,C) then C $\Vdash$ $S\alpha \sqsupseteq S\tau$.

Proof. Proceed by induction on the weight of τ, which decreases in recursive calls as argued in the proof of termination of Eliminate. The basis cases where τ is Ω, atomic or a variable are trivial.

Case 1: $\tau = \tau_1 \to \tau_2$. By the inductive assumption $C_2 \Vdash S_2\beta_2 \sqsupseteq S_2S_0\tau_2$. Now the (S,C) returned in this case is $(S_2S_0, \{\beta_1 \sqsubseteq S_2S_0\tau_1\} \cup C_2)$, and $S_2S_0\alpha = S_2\beta_1 \to S_2\beta_2$, $S_2S_0\tau = S_2\tau_1 \to S_2\tau_2$. We must show that $\{\beta_1 \sqsubseteq S_2S_0\tau_1\} \cup C_2 \Vdash S_2\beta_1 \sqsubseteq S_2S_0\tau_1$ and $\{\beta_1 \sqsubseteq S_2S_0\tau_1\} \cup C_2 \Vdash S_2\beta_2 \sqsupseteq S_2S_0\tau_2$. The latter follows from the inductive assumption noted above since a larger constraint set simply reduces the number of available substitutions. The former is obvious with the observation that $S_2\beta_1 = \beta_1$ by Proposition 7.

Case 2: $\tau = \tau_1 \times \tau_2$. By the inductive assumptions, $C_2 \Vdash S_2\beta_2 \sqsupseteq S_2S_1S_0\tau_2$ and $C_1 \Vdash S_1\beta_1 \sqsupseteq S_1S_0\tau_1$. We need to show that $S_2C_1 \cup C_2 \Vdash S_2S_1S_0\alpha \sqsupseteq S_2S_1S_0(\tau_1 \times \tau_2)$. It is enough to show

(i) $S_2C_1 \Vdash S_2S_1\beta_1 \sqsupseteq S_2S_1S_0\tau_1$

(ii) $C_2 \Vdash S_2S_1\beta_2 \sqsupseteq S_2S_1S_0\tau_2$

Assertion (i) is obvious from the inductive assumptions. To see (ii) note that by Proposition 7, $S_1S_2 = S_2S_1$. Since the substitution S_1 is already incorporated in C_2, $S_1C_2 = C_2$. The rest follows by the inductive assumptions. ∎

Lemma 10. Eliminate($\alpha \sqsupseteq \tau$) succeeds iff $\alpha \sqsupseteq \tau$ is solvable.

Proof. By Propositions 6, we need only show that $\alpha \sqsupseteq \tau$ is solvable iff there is no monotonic occurrence of α in τ or $\alpha = \tau$. Lemma 3 is the only if part (in contrapositive form). For the if part, suppose there is no monotonic occurrence of α in τ or $\alpha = \tau$. By Proposition 6, Eliminate($\alpha \sqsupseteq \tau$) succeeds. By Lemmas 8 and 9, $\alpha \sqsupseteq \tau$ is solvable. ∎

Lemma 11. If Eliminate($\alpha \sqsupseteq \tau$) succeeds and returns (S,C) and for some S', $S'\alpha \sqsupseteq S'\tau$ is valid, then there is an R such that S'=RS and RC is valid.

Proof. By induction on the weight of τ. The three basis cases are trivial.

Case 1: $\tau = \tau_1 \to \tau_2$. By the definition of validity, $S'\alpha = \tau'_1 \to \tau'_2$, and $\tau'_2 \sqsupseteq S'\tau_2$ and $\tau'_1 \sqsubseteq S'\tau_1$ are valid. Therefore, given that $S_0 = [\beta_1 \to \beta_2/\alpha]$, there is an R' such that $S' = R'S_0$, $\tau'_1 = R'\beta_1$ and $\tau'_2 = R'\beta_2$. We thus have $\tau'_2 = R'\beta_2 \sqsupseteq R'S_0\tau_2 = S'\tau_2$. Since Eliminate($\beta_2 \sqsupseteq S_0\tau_2$) succeeds and returns (S_2, C_2), by the inductive assumption there is an R such that $R' = RS_2$, and RC_2 is valid. Clearly, $S' = RS_2S_0$ as required. Since S_2 does not map β_1 by Proposition 7, $R\beta_1 = RS_2\beta_1 = R'\beta_1 = \tau'_1$. Since $\tau'_1 \sqsubseteq S'\tau_1$ is valid, so is $R\{\beta_1 \sqsubseteq S_2S_0\tau_1\}$.

Case 2: $\tau = \tau_1 \times \tau_2$. By an argument similar to *Case 1* we can show that $\tau'_1 = R'\beta_1 \sqsupseteq R'S_0\tau_1 = S'\tau_1$ and $\tau'_2 = R'\beta_2 \sqsupseteq R'S_0\tau_2 = S'\tau_2$. Since Eliminate($\beta_1 \sqsupseteq S_0\tau_1$) succeeds and returns (S_1, C_1), by the inductive assumption there is an R_1 such that $R' = R_1S_1$, and R_1C_1 is valid. The valid constraint $R'\beta_2 \sqsupseteq R'S_0\tau_2$ can now be rewritten as $R_1S_1\beta_2 \sqsupseteq R_1S_1S_0\tau_2$. By Proposition 7, $S_1\beta_2 = \beta_2$, therefore the constraint can be further rewritten as $R_1\beta_2 \sqsupseteq R_1S_1S_0\tau_2$. Since Eliminate($\beta_2 \sqsupseteq S_1S_0\tau_2$) succeeds and returns (S_2,C_2), by the inductive assumption there is an R such that $R_1 = RS_2$, and RC_2 is valid. Clearly, $S' = RS_2S_1S_0$ as required. The validity of $R(S_2C_1 \cup C_2)$ follows from the validity of $RS_2C_1=R_1C_1$ and RC_2 which were shown to hold due to the inductive assumptions. ∎

The property proved in Lemma 11 is reminiscent of one of the properties of most general unifiers (Wand, 1987). To make this precise, define a substitution S to be a *needed approximator* for a constraint set C, written $C \gg S$, iff whenever $S' \models C$, there is a substitution R such that $S'=RS$. A substitution S is said to be a *complete approximator* for a constraint set C, iff $C \gg S$ and whenever $C \gg S'$, there is a substitution R such that $S = RS'$. Both needed and complete approximators specify necessary *but not sufficient* instantiations for the given constraint set. A needed approximator is a factor in any substitution which validates a set of constraints. A complete approximator is as specific as possible without ceasing to be needed.

Lemma 12. A complete approximator S exists for every constraint set C.

Proof. If C is inconsistent, the statement is vacuous. Otherwise, suppose $M \models C$. Define a sequence $S_0, S_1, S_2, \ldots$ of needed approximators, and a corresponding sequence $C_0, C_1, C_2, \ldots$ of constraint sets. Let $C_0=C$. If there is an $S_i \neq \mathbb{1}$ mapping variables in C_i such that $C_i \gg S_i$, then $C_{i+1}=S_iC_i$, otherwise the sequence terminates. Clearly, for each i, $C \gg S_iS_{i-1}\cdots S_0$. Since M is finite, and the number of symbols in M must be greater than that in any needed approximator, the sequence must terminate at some i=n. If n=0 then $S=\mathbb{1}$, otherwise $S = S_nS_{n-1}\cdots S_0$. ∎

This Lemma can be used to define a specialization of the notion of principle type (as a pair C,τ) where the complete approximator for C is required to be $\mathbb{1}$. This would yield a unique principle type but one we don't know how to derive!

The properties of **Eliminate** are summarized in:

Theorem 13. There is an algorithm **Eliminate** such that **Eliminate**$(\alpha \sqsupseteq \tau)$ succeeds iff $\alpha \sqsupseteq \tau$ is solvable, and upon success, returns a substitution S and a set C such that
(i) C is in normal form
(ii) $C \equiv \{S\alpha \sqsupseteq S\tau\}$
(iii) $\{\alpha \sqsupseteq \tau\} \gg S$

Proof. Straightforward consequence of Lemmas 5, 8, 9, 10 and 11. ∎

It is easy to generalize **Eliminate** to a procedure **Resolve** which *partially* solves a set C of arbitrary constraints, by determining the consistency of C, and converting it to normal form, producing a needed approximator as the partial solution. Unfortunately, the unrestricted generalization produces only a semi-decision procedure for consistency due to the possibility of indirect unsolvable circularities. In formulating **Resolve**, it is convenient to define a simple auxiliary procedure **Simplify**. Suppose we refer to any constraint $\tau_1 \sqsubseteq \tau_2$ as a *bound* whenever either τ_1 or τ_2 is a variable. A bound will be called *nontrivial* when $\tau_1 \neq \tau_2$. **Simplify** will reduce any C to an equivalent set consisting only of nontrivial bounds.

```
Simplify(C)
Partition C into $C^1$ and $C^2$, where $C^1$ contains all
  the nontrivial bounds in C.
if $C^2=\emptyset$ then return C else pick any $\tau_1 \sqsubseteq \tau_2 \in C^2$
  and let $C' = C - \{\tau_1 \sqsubseteq \tau_2\}$
if $\tau_1=\Omega$ or $\tau_1=\tau_2$ then return Simplify(C')
else if $\tau_1$ is atomic then
  if $\tau_1 = \tau_2$ then return Simplify(C') else fail
else if $\tau_1 = \tau_{11} \rightarrow \tau_{12}$ then
  if $\tau_2 \neq \tau_{21} \rightarrow \tau_{22}$ then fail
  else return Simplify($C' \cup \{\tau_{21} \sqsubseteq \tau_{11}, \tau_{12} \sqsubseteq \tau_{22}\}$)
else $\tau_1 = \tau_{11} \times \tau_{12}$ (only possibility left)
  if $\tau_2 \neq \tau_{21} \times \tau_{22}$ then fail
  else return Simplify($C' \cup \{\tau_{11} \sqsubseteq \tau_{21}, \tau_{12} \sqsubseteq \tau_{22}\}$)
```

It is easy to prove that **Simplify** terminates since the number of symbols decreases in each tail-recursive call. It is equally obvious by the definition of validity that **Simplify**(C) fails only if C is inconsistent, and if it succeeds then $C \equiv$ **Simplify**(C). Procedure **Resolve** is presented in iterative rather than recursive form to help separate its partial correctness properties from its termination properties. The former can be stated as loop invariants, and termination for consistent sets follows from those invariants. See (Wand, 1987) for a similar iterative formulation for solving equality constraints and (Fuh-Mishra, 1987) for an iterative consistency algorithm for atomic coercions. Recall that $\mathbb{1}$ represents the identity substitution.

Resolve

```
Argument: A constraint set $C_0$
Initialization: Set $C = \text{Simplify}(C_0)$, $S = \mathbb{1}$
Loop
   Partition C into $C^1$ and $C^2$, where $C^1$ is the
         largest subset of C in normal form.
   if $C^2=\emptyset$ then halt else pick any $\tau \sqsubseteq \alpha \in C^2$
         and let $C' = C - \{\tau \sqsubseteq \alpha\}$
   Set $C = \textbf{Simplify}(C_1 \cup S_1C')$ and $S = S_1S$
         where $(S_1,C_1) = \textbf{Eliminate}(\alpha \sqsupseteq \tau)$
End of Loop
```

Example 5.

Type(A,*(λf.λa,b.(f a), (f b))(λx.x)(1,true)*)
$= C_0,\gamma$

$A = \{1{:}int,\ true{:}bool\}$

$C_0 = \{\alpha_1 \to \alpha_2 \times \alpha_3 \to \gamma_1 \times \gamma_2 \sqsupseteq (\beta \to \beta) \to \gamma_3,$
$\gamma_3 \sqsupseteq \text{int} \times \text{bool} \to \gamma,\ \alpha_1 \sqsupseteq \alpha_2 \to \gamma_1,\ \alpha_1 \sqsupseteq \alpha_3 \to \gamma_2\}$

Resolve(C_0) terminates with

$C = \{\beta \sqsubseteq \alpha_{11},\ \alpha_{11} \sqsubseteq \alpha_2,\ \alpha_{11} \sqsubseteq \alpha_2,\ \alpha_2 \sqsubseteq \gamma_{311},$
$\alpha_3 \sqsubseteq \gamma_{312},\ \gamma_{311} \sqsubseteq int,\ \gamma_{312} \sqsubseteq bool,\ \alpha_{12} \sqsubseteq \beta,$
$\gamma_1 \sqsubseteq \alpha_{12},\ \gamma_2 \sqsubseteq \alpha_{12},\ \gamma_{32} \sqsubseteq \gamma_1 \times \gamma_2,\ \gamma \sqsubseteq \gamma_{32}\}$

$S = [\alpha_{11} \to \alpha_{12} / \alpha_1,\ \gamma_{311} \times \gamma_{312} \to \gamma_{32} / \gamma_3]$

It is implicitly assumed that if the call to either **Eliminate** or **Simplify** fails then the **Resolve** procedure halts with failure. If the halt instruction in the loop is encountered, it halts with success and returns the current values of C and S as the result. The following invariants hold for the loop:

Invariants:

1. $(\forall M)\ (M \models C_0 \Rightarrow (\exists M')\ (M{=}M'S \text{ and } M' \models C))$
2. $C \Vdash SC_0$

These invariants are exactly analogous to those given by Wand (1987) for his most general unifier algorithm. Both trivially hold after initialization. It remains to show that they are preserved by each iteration, if the iteration completes successfully.

Lemma 14. The loop in Resolve preserves:

$(\forall M)\ (M \models C_0 \Rightarrow (\exists M')\ (M{=}M'S \text{ and } M' \models C))$

Proof. Consider an arbitrary M such that $M \models C_0$. Since the property holds at the beginning of the iteration, there is an M' such that $M' \models C$ and $M = M'S$ for the starting values of C and S. Therefore, since $M'\alpha \sqsupseteq M'\tau$ is valid as part of M'C, by Theorem 13, there is an R' such that $M'{=}R'S_1$ and $R'C_1$ is valid. Putting this together with the fact that $M'C'{=}\ R'S_1C'$ is also valid as part of M'C, we know that $R'(C_1 \cup S_1C')$ is valid. Since $M = M'S = R'S_1S$, and $\text{Simplify}(C_1 \cup S_1C') \equiv (C_1 \cup S_1C')$, $R' \models C$ and $M = R'S$ for the values of C and S at the end of the loop. ∎

Lemma 15. The loop in Resolve preserves the property: $C \Vdash SC_0$.

Proof. If C is inconsistent, the property is vacuous, therefore assume that C is consistent. By Theorem 13, Eliminate($\alpha \sqsupseteq \tau$) succeeds and gives (C_1,S_1) such that $C_1 \equiv S_1\alpha \sqsupseteq S_1\tau$. Clearly therefore $(C_1 \cup S_1C') \equiv S_1C$. Since $S_1C \Vdash S_1SC_0$ by the inductive assumption, the rest follows given that $\text{Simplify}(C_1 \cup S_1C') \equiv (C_1 \cup S_1C')$. ∎

Lemma 16. Resolve(C_0) terminates if C_0 is consistent.

Proof (sketch). The proof is based on Lemma 14 which asserts that the substitution S is a factor in any model of C_0. Suppose V is the set of variables in C_0. Suppose Z_n is the total number of symbols in the expressions to which variables in V are mappped by S after n iterations of the Resolve loop. Since these expressions are a factor in any model, the number Z_n cannot go on increasing if there is a finite model for C_0. We argue that Z_n does increase without limit in case of nontermination. It is easy to see by inspection of Eliminate and Resolve that each variable in C either belongs to V or is a part of $S\alpha$ for some $\alpha \in V$. Since S_1 cannot indefinitely map variables to atomic types without exhausting the supply of variables, the only way nontermination can happen is if S_1 maps the variable being eliminated to a functional or product type infinitely often, which implies that Z_n increases without limit. ∎

Lemma 17. Resolve(C_0) terminates with failure only if C_0 is inconsistent.

Proof. Easy induction on the number of iterations given the invariants for the loop and the properties of Eliminate and Simplify. ∎

Except for termination, the properties of **Resolve** are exactly analogous to those of **Eliminate**.

Theorem 18. There is a procedure Resolve such that Resolve(C_0) terminates with success iff C_0 is consistent, and upon success, returns a substitution S and a constraint set C such that

(i) C is in normal form.

(ii) $C \equiv SC_0$

(iii) $C_0 \gg S$

Proof. This theorem is exactly analogous to Theorem 13, and largely derives from it. The proof follows from Lemmas 14, 15, 16 and 17. ∎

To summarize, **Type** and **Resolve** can be combined to produce a well-typing procedure that is a generalization of the Hindley-Milner algorithm for the **Exp** language. All our rules are generalizations of a version of the Damas-Milner (1982) rules given by Clement, *et al* (1986). Whenever $A \vdash e{:}\tau$ in the latter system, we have $\emptyset, A \vdash e : \tau$. By the completeness of **Type, Type**(A,e) returns (C,τ) such that $\emptyset \Vdash SC$ for some S. Therefore, $S \models C$, i.e., C is consistent, and **Resolve** halts with success. Our procedure also terminates successfully for heterogeneous expressions which are considered ill-typed under Hindley-Milner merely due to heterogeneity. Of the rest which are ill-typed in both systems, some cause nontermination in the **Resolve** phase of the procedure.

5.3 Termination and other improvements

It seems unlikely that **Resolve** can be restricted to a terminating version without losing useful generality. There *are* some heuristics that greatly reduce the chances of nontermination without significant loss of generality in practice.

One example for nontermination is $\{\alpha \sqsupseteq \beta \times int, \alpha \sqsubseteq \beta\}$ where β indirectly has a lower bound with a monotonic circular occurrence. Such sets cause nontermination because the circularity never becomes manifest. These cases can be detected by standard algorithms for detecting cycles in digraphs. Fortunately, a set that is cycle free stays that way, so the check needs to be performed only once. Nonterminating constraint sets such as those for *(λx.xx)(λx.xx)* contain circularities that are more subtle. One way to avoid these is to disallow (solvable) circular lower bounds such as $\alpha \sqsupseteq \alpha \to \beta$ which seem to be involved in these cases in an essential way. This reduces generality but seems to affect only expressions with complex self-application which are of little practical interest.

The efficiency of the whole process can be improved by changing one of the typing rules and making the inference procedure incremental. The COMP rule given in §4, though convenient for a neat formulation of **Type**, needlessly introduces extra variables. The equivalent rule

$$\text{COMP: } \frac{C, A \vdash e: \tau' \to \tau \qquad C, A \vdash e': \tau'' \qquad C \Vdash \tau'' \sqsupseteq \tau'}{C, A \vdash e\,e': \tau}$$

is better. It is equally easy to reformulate the combination of **Type** and **Resolve** as an incremental procedure. If a set of constraints currently in force is carried as an argument and result, **Resolve** can be applied to it every time a new constraint is added.

Even when **Resolve** terminates successfully, it leaves the constraints in a less than fully reduced form. A simple way to further reduce constraints is to observe that a variable with multiple (direct and indirect) upper bounds is effectively bounded above by the *meet* of all such bounds. Consider for instance **Diagram 1** below which shows the relationships between variables in the result of **Resolve** in **Example 5**. The variable α_{11} is bounded above by both *int* and *bool*, whose meet is Ω. The only possible instantiation for α_{11} (and β, α_{12}, γ_1 and γ_2) within these constraints is therefore Ω. The type of the overall expression, γ, is bounded above only by $\Omega \times \Omega$, which is therefore the best type for theexpression. This idea can be formalized by defining a pair of algorithms **Meet** and **Join**. We leave the details to the reader.

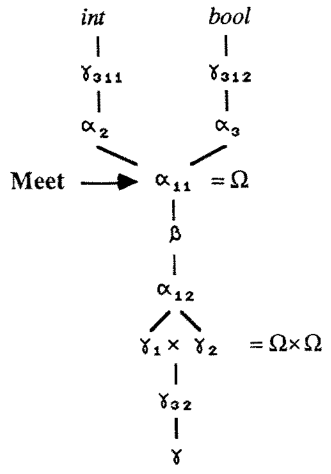

Diagram 1

6. Dynamic checks and error detection

We started by motivating the system described above as an attempt to generalize the Hindley-Milner type inference system to permit heterogeneous structures. The generalization offered by the system presented so far is rather one-sided. Heterogeneous expressions like *[1,true,['c']]* and *(λf.λa,b.(f a), (f b))(λx.x)(1,true)* are well-typed, but their components cannot be used without type-error, because it is not possible in general to statically derive sufficient type information to validate non-trivial uses of heterogeneous structures. It would not be very useful to allow *[1, true]* but disallow *succ (hd [1, true])*. Our goal must therefore be to check if it is *plausible* that such expressions will evaluate without type-errors at run time. To accomplish this, we need explicit (system-generated) dynamic checks and a way to verify that each such check is plausible, i.e., has a chance to succeed at run-time. The former requires a new syntactic form $e\uparrow\tau$. An expression in this form has independent meaning only if the type τ is closed, i.e., a monotype. In that case we define

$$\mathbb{E}[\![e\uparrow\mu]\!]\eta = \text{if } v\in\mathbb{T}(\mu) \text{ then } v \text{ else } \textit{runerror} \quad \text{where } v = \mathbb{E}[\![e]\!]\eta$$

The new semantic value *runerror* represents a type error *detected* at run-time, in contrast to *wrong*, which represents the result of *unchecked* nonsensical computation. We assume that *runerror* belongs to *every* monotype. This makes it easy to give a typing rule for the new form since $e\uparrow\mu$ possesses the type μ whether or not e does. The antecedent in the rule below ensures that e does not contain type errors (if C is consistent).

$$\text{CHECK: } \frac{C, A \vdash e: \tau'}{C, A \vdash e\uparrow\tau : \tau}$$

With such a liberal rule, *any* type can be infered for an error-free expression with appropriate dynamic checks. It is not clear if there is an optimal strategy which will maximize static error detection. A reasonable strategy is to introduce a minimum of dynamic checks needed to make the expression well-typed. The process of identifying needed dynamic checks and detecting errors resembles code optimization in that the aim in both processes is to produce the maximum results attainable at reasonable expense rather than to achieve every gain imaginable. The strategy presented here is almost certainly less than optimum, but it serves to illustrate the possibilities, and also some of the difficulties. Before describing the details, we first discuss and illustrate some of the ideas involved.

It is clear that dynamic checks need to be introduced in those applications where a type error is detected, e.g., as in *succ (hd [1,true])↑int*. In general, the check consists of the argument type demanded by the function part. In a side-effect oriented implementation, this process involves undoing bindings generated by the failed call on **Resolve**, i.e., backtracking Prolog-style. Of course, one must also ensure that the check is *plausible*. If it is not then we have found a *real* error that must be reported to the user.

A difficulty in this process is the essentially monomorphic nature of λ-bound identifiers, which tends to cause a premature commitment of the types of such identifiers. Consider for example *λx.(succ (car x), not (cadr x))* where *car* and *cadr* are as in LISP. The constraints generated by *succ (car x)* cause the type of x to be instantiated to $[int]$. Introducing the check *(cadr x)↑bool* later does not help because the check is then not plausible. We want to permit this expression because it simply requires x to be a mixed list. Since we do not want to ask the user to declare the types of λ-bound identifiers, we need an extra pass to derive the necessary information.

The plausibility of the check $e\uparrow\tau$ in the CHECK rule is the assurance that the check is a "reverse coercion", i.e., that $C \Vdash \tau' \sqsubseteq \tau$ for every type τ' known to be possessed by e subject to the constraints in C. The obvious way to enforce this constraint would be to include it as a condition in CHECK. This turns out to be unsatisfactory because the addition of this constraint may itself force a reduction in the "known" type information about e. For example, *not (hd [1])* causes a type error and is transformed to *not (hd [1])↑bool*. The type of *(hd [1])* without a check is $\{\alpha \mid \alpha \sqsubseteq int\}$; adding the plausibility constraint would make it $\{\alpha \mid \alpha \sqsubseteq int \;\&\; \alpha \sqsubseteq bool\}$ which would simply cause α to be instantiated to Ω, making the check plausible. While this would be sound, it would defeat the purpose of plausibility checking. It is better to derive all type information for each checked expression and then use the best (smallest) known type in the plausibility check. The best type is obtained by replacing any variable which has a *unique* upper bound by that bound in all *non-functional* types. This requires a separate accumulation of plausibility checks to be applied after type inference is finished.

A more serious difficulty arises in the interaction between λ-bound identifiers and conditional expressions. For instance, what should be the type of *(λx. if x=1 then succ x else x)*? In a LISP-like language, this function would accept any input without error. However, the uses of x do not *force* the conclusion that the function is meant to accept heterogeneous arguments. The solution of this problem appears to require special treatment of conditional expressions. In essence, the constraints generated by alternatives in a conditional expression need to be viewed *disjunctively*, whereas those generated by the components of tuples, lists, etc., are construed *conjunctively*. In the absence of such a scheme, we have to be either optimistic or pessimistic about homogeneity for such expressions in deriving the expected types of λ-bound identifiers in *Pass 1* below. We have chosen to be optimistic, which means that we do not always recognize potential heterogeneity in functions. In particular, the type for the expression above comes out as $int \rightarrow int$.

The two passes of the well-typing and error detection algorithm for expressions *without* let-bindings can now be informally described:

Pass 1:

1. Transform the given expression e by inserting a check consisting of a new variable for each (non-binding) occurrence of a λ-bound identifier.
2. Process the expression using the incremental version of **Type+Resolve**, inserting additional checks on discovery of type errors as described above. In addition to the constraints generated by COMP, add to C a new constraint $\alpha \sqsubseteq \beta$ for each non-binding occurrence of a λ-bound identifier, where α is the type for the binding occurrence and β is the check for the non-binding one. Reduce the remaining constraint set C by the methods of §5.3.

Pass 1 always finishes successfully since all errors are masked by checks and plausibility of checks is not tested.

Pass 2:

1. Forget the transformation and typing information associated in *Pass 1* with e *except the form* of the typing for binding occurrences of λ-bound identifiers. This is essentially the "best types" for these identifiers with all variables replaced with distinct new variables.
2. Restart the **Type+Resolve** process with e enhanced with type assumptions for λ-bindings. Each time a check has to be introduced due to a type error, add the pair consisting of the type of the checked subexpression and the check variable to a global "plausibility set".
3. At the end of **Type+Resolve** (with §5.3 enhancements), replace the first component in each pair in the plausibility set by the corresponding best type. The pairs in the transformed plausibility set are then added to the

remaining constraints one at a time and checked for consistency using **Resolve**. If consistency is maintained throughout, the expression is accepted otherwise a type-error is declared for the subexpression whose plausibility pair caused inconsistency.

If we start *Pass 1* with the expression $\lambda x.(succ\ (car\ x \uparrow \alpha),\ not\ (cadr\ x \uparrow \beta))$ where α and β are fresh variables, we have the instantiations $\alpha=[int]$ and $\beta=[bool]$, which result in the typing $x:[\Omega]$. *Pass 2* therefore starts with $\lambda x:[\Omega]:(succ\ (car\ x),\ not\ (cadr\ x))$ and ends with the expression $\lambda x:[\Omega]:(succ\ (car\ x) \uparrow int,\ not\ (cadr\ x) \uparrow bool)$ with dynamic checks at the right spots. The plausibility set $\{\Omega \sqsubseteq int,\ \Omega \sqsubseteq bool\}$ is trivial, so the expression does not contain a static error.

The system accepts some obviously erroneous expressions. For instance, $\lambda x.\ succ\ x,\ not\ x$ produces $\lambda x:\Omega.\ succ\ x,\ not\ x$ after *Pass 1* and $\lambda x:\Omega.\ succ\ x \uparrow int,\ not\ x \uparrow bool$ after *Pass 2*. It is tempting to suggest that a typing of Ω for a λ-binding is an error, which would indeed allow us to catch many more real errors. Unfortunately, it would also disallow heterogeneity produced by conditional expressions. The special treatment of conditional expressions suggested above would permit a separation of these cases.

If the expression concerned contains let-bindings, this affects the generation of plausibility checks. The plausibility tests in the body of a let binding are not automatically added to the global set. As with constraints, they are quantified if they involve quantified variables, and copies with appropriate generic instantiation of variables are added each time a let-bound identifier is used. Pairs that don't involve quantified variables are added to the global set.

7. Conclusions

We have shown that the Hindley-Milner system of implicit typing for the **Exp** dialect of the λ-calculus can be extended using partial types to allow the typing of many heterogeneous expressions. The extension was used to show the feasibility of extending type inference to languages with heterogeneous data structures with the ability to detect the locations and content of required dynamic checks, and also to catch many errors statically. The methods sketched in §6 are preliminary and need both theoretical and practical refinement. The single most important theoretical refinement needed is a better treatment of conditional expressions.

I would like to thank Prateek Mishra for helping spot a significant error in an earlier version.

References

Cardelli, L. (1984), A Semantics of Multiple Inheritance, *Semantics of Data Types, International Symposium*, LNCS **173**.

Cardelli, L. and Wegner, P. (1985), On Understanding Types, Data Abstraction and Polymorphism, *Computing Surveys* **17** (4).

Clement, *et al* (1986), A Simple Applicative Language: Mini-ML, *Proceedings of 1986 ACM Conference on LISP and FP*, Cambridge, Mass.

Damas, L. and Milner, R. (1982), Principle Type-schemes for Functional Programs, *Proceedings of the 9th POPL*, Albuquerque, NM.

Milner, R. (1978), A Theory of Type Polymorphism in Programming, *JCSS* **17**.

Fuh, Y.-C. and Mishra, P. (1987), Type Inference with Subtypes, *manuscript*.

Mitchell, J. (1984), Lambda Calculus Models of Typed Programming Languages, PhD. Thesis, M.I.T.

Reynolds, J. (1985), Three Approaches to Type Structure, *TAPSOFT 1985*, LNCS **186**.

Scott, D. (1982), Domains for Denotational Semantics, *Proceedings of ICALP'82*.

Wand, M. (1987), Complete Type Inference for Simple Objects, *Proceedings of 1987 IEEE Symposium on Logic in Computer Science*.