Type inference with simple subtypes

John C. Mitchell

Department of Computer Science
Stanford University
Stanford, CA 94305

Tuesday, 15 January 1991

Abstract

Subtyping appears in a variety of programming languages, in the form of the "automatic coercion" of integers to reals, Pascal subranges, and subtypes arising from class hierarchies in languages with inheritance. A general framework based on untyped lambda calculus provides a simple semantic model of subtyping and is used to demonstrate that an extension of Curry's type inference rules are semantically complete. An algorithm G for computing the most general typing associated with any given expression, and a restricted, optimized algorithm GA using only atomic subtyping hypotheses are developed. Both algorithms may be extended to insert type conversion functions at compile time or allow polymorphic function declarations as in ML.

1. Introduction

Type inference is a form of type checking. In programming languages where all identifiers are given types as they are introduced, it is often a simple matter to check whether the types of operators, functions and procedures agree with the types of operands and actual parameters. For some programming applications, it is convenient to be able to omit type declarations from programs, leaving the programming language processor (editor, interpreter or compiler) with the task of inferring the missing type information. Automatic type inference may make it easier to write experimental programs quickly or allow a single untyped program to represent many explicitly typed programs. In addition, type inference sometimes provides more useful debugging information. If we expect a function to have one type, and the programming language processor infers another, this may suggest a bug in the function declaration. Type inference seems to have originated with [Curry and Feys 58, Hindley 69] and, independently, [Milner 78]. A general discussion of the use of type inference in programming languages may be found in [Milner 78]; for further information on type inference, see, e.g., [Barendregt, Coppo and Dezani 83, Coppo, Dezani and Venneri, Coppo 83, Kanellakis, Mairson and Mitchell 9+, MacQueen, Plotkin and Sethi 86, Mitchell 90, Wand 87].

Many programming languages use some form of subtyping. The most common uses are in "coercions," as in the automatic conversion of integers to real (floating point) numbers, and in the subclassing mechanisms of object-oriented languages such as C^{++} [Stroustrop 86]. In this paper, which extends the conference abstract [Mitchell 84a], we will investigate the semantics and algorithmic aspects of type inference for pure lambda terms in the presence of various forms of subtyping hypotheses. The main results are a semantic completeness theorem and two type inference algorithms. The semantic study uses a model of subtyping based on set containment: if type σ is a subtype of τ, then we will think of the set of values associated with type σ as a subset of the values associated with type τ. Mathematically, the integers may be regarded as a subset of the reals, and so our model applies. Some "subtyping" relationships, such as simplified versions of Pascal subranges, may also be interpreted in this manner. The basic results in this paper may also have some application to languages with subtyping derived from class hierarchies (cf. [Cardelli 88, Wand 87]). However, we do not consider subtyping based on structural similarities between distinct record types. For discussion of this related topic, see, e.g., [Cardelli 88, Cardelli and Mitchell 89, Jategaonkar and Mitchell 88, Wand 87, Remy 88].

Even without subtyping, type inference allows a single untyped function to have infinitely many types. For example, the body f(x) of the function

$$\text{Apply}(f,x) = f(x)$$

has type t whenever f has functional type $s \longrightarrow t$ and x has type s. (The type operator $\longrightarrow$ means "function space," so $s \longrightarrow t$ is the type of functions with domain s and range t.) Thus Apply has type

$$\text{Apply}: ((s \longrightarrow t) \times s) \longrightarrow t$$

for every pair of types s and t. (The type operator $\times$ means "product space." For example, int $\times$ bool is the type of pairs <a,b> where a is an integer and b is a boolean.) In particular, Apply has every type that is a substitution instance of the above expression, for example

$$\text{Apply: } ((\text{int} \longrightarrow \text{bool}) \times \text{int}) \longrightarrow \text{bool}$$

$$\text{Apply: } ((\text{real} \longrightarrow \text{int}) \times \text{real}) \longrightarrow \text{int}.$$

In typed programming languages similar to Algol or Pascal, we would have to declare a different Apply function for each type of application function that we need. However, since application functions of different types are defined the same way, except for type declarations, this seems an unfortunate restriction; it would be simpler to declare Apply only once. One way for a typed language to allow this flexibility is to use a type inference algorithm. Since an inference algorithm could infer that an untyped Apply function has type $((s \longrightarrow t) \times s) \longrightarrow t$ for every s and t, an inference algorithm could allow calls of the form Apply(g,y) whenever the type of the pair (g,y) is a substitution instance of $(s \longrightarrow t) \times s$. By this means, type inference algorithms may be used to support an implicit form of polymorphism.

ML is a popular and well-known programming language incorporating implicit polymorphism [Gordon, et. al. 79]. In the ML type system, every typable expression M has a *most general typing*, consisting of an association A of types to free variables of M and a type expression σ. The efficiency of the ML type checker seems to be a direct consequence of the fact that every type of an ML expression may be constructed from the most general typing by substitution. In this paper, we will add subtyping to a subset of ML and obtain similar results. For every typable M, there will be a set of subtyping conditions C, an association A of type expressions to variables, and type expression σ such that every legal typing of M can be constructed from C, A and σ. With subtyping, alternative typings are constructed from the principal typing using substitution and a proof system for subtypes.

The typing algorithms developed in this paper provides a means for combining implicit ML-style polymorphism with user-specified subtyping or coercions. There are a number of programming language design issues surrounding user-defined coercions which will not be addressed in any detail. As far as type inference goes, it may be possible to allow arbitrary coercions to be declared at any point in a program. However, if user-supplied conversion functions are used, then the meaning of a program may not be uniquely determined (see [Reynolds 80]). For this reason, most practical programming languages are likely to allow only a restricted form of user-defined subtypes, such as those associated with class and subclass declarations. One reasonable and algorithmically interesting kind of subtyping is coercion between atomic types. Since the most reasonable place to declare new subtyping relationships seems to be when new types are declared, and type declarations generally introduce new atomic type names, this may be the most useful case in practice. In addition, atomic subtyping restricts the type inference problem in a way that allows optimization of the typing algorithm. Finally, if added to ML, atomic subtyping would preserve the character of the language in that precisely the same set of pure expressions would be typable. One implication is that programmers would receive approximately the same kind of error messages in ML with atomic subtyping as in ML without. Since the ML type checker has proven useful for detecting errors over many years, the restrictions imposed by atomic coercions might be useful in practice.

Some basic facts about lambda calculus will be reviewed in Section 2, followed by a discussion of type expressions, coercions sets and typing statements in Section 3. The typing rules and soundness and completeness theorems are appear in Section 4. A general typing algorithm is presented in Section 5, and a restricted, optimized version developed in Section 6. Some extensions and variations of the algorithms are discussed in Section 7, with Section 8 concluding. Unification and some variants, which are used in the typing algorithms, are discussed in the appendix.

Preliminary versions of some of the results presented here were summarized in [Mitchell 84a]. Later studies (based on [Mitchell 84a]) include [Jategaonkar and Mitchell 88, Fuh and Mishra 88, Wand and O'Keefe 89].

2. Lambda Calculus and its Semantics

Lambda calculus will be used to demonstrate type inference with subtyping. The terms of untyped lambda calculus are defined by the grammar

$$M ::= x \mid MN \mid \lambda x.M,$$

where x may be any variable. Intuitively, MN is the application of function M to argument N, and λx.M is the function we obtain by treating M as a function of x. Although programs also contain constants like 0, 1, 2, + and "**if**...**then**...**else**...", it will be notationally simpler to only consider pure lambda terms without constants. The main results of this paper may be extended to expressions with constants without difficulty.

A *lambda model* <D, $\cdot$, ε> is a set D together with binary operation $\cdot$, "choice element" $\varepsilon \in$ D, and elements K, S $\in$ D satisfying certain algebraic conditions. This is the combinatory model definition of [Meyer 82]; see also [Barendregt 84]. We will not be concerned with the specific properties of K and S, but it is worth pointing out that $(\varepsilon \cdot d)\cdot e = d\cdot e$ for all d, e$\in$D, and if $d_1 \cdot e = d_2 \cdot e$ for all e$\in$D, then $\varepsilon\cdot d_1 = \varepsilon \cdot d_2$. Intuitively, these conditions mean that $\varepsilon\cdot d$ represents the same function (on D) as d, and that if d_1 and d_2 represent the same function, then $\varepsilon\cdot d_1 = \varepsilon\cdot d_2$. This means that for every function f which is represented by some d$\in$D, we can use ε to choose "canonical" element $\varepsilon\cdot d$ representing f. The fact that d and $\varepsilon\cdot d$ represent the same function will be important for understanding some properties of functional types.

Given a lambda model <D, $\cdot$, ε> and environment η mapping variables to elements of D, the meaning of a lambda term M is defined inductively by

$$[\![x]\!]\eta = \eta(x)$$

$$[\![MN]\!]\eta = [\![M]\!]\eta \cdot [\![N]\!]\eta$$

$$[\![\lambda x.M]\!]\eta = \varepsilon\cdot d, \text{ where } d\cdot e = [\![M]\!]\eta[e/x]$$

The existence of K and S ensure that there always exists a d as required in the definition of $[\![\lambda x.M]\!]$. The element ε makes the meaning of $\lambda x.M$ independent of the specific choice of d. Again, the reader is referred to [Barendregt 84, Meyer 82] for more information. One specific fact relevant to functional types is that if x is not free in M, then $[\![\lambda x.Mx]\!]\eta = \varepsilon\cdot[\![M]\!]\eta$

A few facts about the reduction rules (operational semantics) of lambda calculus will be used. The reader is referred to [Barendregt 84] for a comprehensive presentation. We consider lambda terms modulo α-conversion

$$(\alpha) \quad \lambda x.M = \lambda y.[y/x]M \text{ if y is not free in M}$$

so that we can rename bound variables. The reduction rules are

(β) $(\lambda x.M)N \rightarrow_{\beta} [N/x]M$,

(η) $\lambda x.Mx \rightarrow_{\eta} M$ if x is not free in M,

where substitution of N for x in M, written [N/x]M, is defined with renaming of bound variables to avoid capture. If a term M is of the form of the left-hand side of rule (β) or (η), then M is a β- or *η-redex*. We say that M *β-reduces to N in one step* if there is a subterm P of M which is a β-redex and N is the result of contracting this redex in M. The term M *β-reduces* to N, written $M \twoheadrightarrow_{\beta} N$, if there is a sequence of β-reductions leading from M to N. The η-reduction relation is defined similarly. The combination of β- and η-reduction is called *β,η-reduction* and written $M \twoheadrightarrow_{\beta,\eta} N$. A term which cannot be reduced is in *normal form*.

The equational proof system for lambda calculus is obtained by taking all instances of (α) and an equational version of (β) as axioms, along with inference rules to make = a congruence with respect to application and lambda abstraction. A *lambda theory* is any set of equations containing (α) and (β) and closed under the inference rules. A theory is *extensional* if it contains all instances of an equational version of (η). *Conversion* is the least congruence relation containing reducibility; $=_{\beta}$ denotes β-conversion and $=_{\beta,\eta}$ denotes β,η-conversion. Thus $M =_{\beta} N$ iff every theory contains M=N, and $M =_{\beta,\eta} N$ iff every extensional theory contains M=N.

One important model is the term model. Given any lambda theory *Th*, we let $[M]_{Th}$ be the set of terms N with $M{=}N \in Th$. The term model $\langle D, \cdot, \varepsilon \rangle$ for *Th* has equivalence classes of terms as elements,

$$D = \{ [M]_{Th} \mid M \text{ an untyped term} \}.$$

Application, $\cdot$, in term models is defined by

$$[M] \cdot [N] = [MN]$$

and choice element defined by

$$\varepsilon = [\lambda x.\lambda y.xy],$$

where we omit the subscript *Th* when it seems to be clear from context. See [Barendregt 84, Meyer 82] for further discussion of term models.

3. Type Expressions, Coercions and Type Assignments

Although product types, lists, and other kinds of types are useful in programming languages, function spaces seem to raise most of the significant typing issues related to subtyping, short of the more involved problems that arise with record types (see [Jategaonkar and Mitchell 88, Cardelli and Mitchell 89]). Therefore, $\rightarrow$ will be the only type connective. We will adopt the notational conventions that

r, s, t, ... denote type variables

ρ, σ, τ, ... denote type expressions.

To be precise, the type expressions are defined by the grammar

$$\tau ::= \ t \mid \sigma \rightarrow \tau.$$

Intuitively, the functional type $\sigma \rightarrow \tau$ consists of the set of functions which take arguments of type σ to results of type τ. We have omitted type constants as a matter of notational convenience, and to eliminate routine cases in inductive proofs. Constants do not alter the inference rules, and require only minor modifications to the supporting algorithms given in the appendix. The necessary modifications will be discussed briefly there.

For grammatical reasons, we will often use "coercion" as a synonym for subtyping. We will use $\sigma \subseteq \tau$ to denote the fact (or assumption) that σ is a subtype of τ or, equivalently from our point of view, values of type σ may be coerced to values of type τ. If we think of subtyping as an ordering, then $\rightarrow$ is monotonic in its second argument:

$$\text{if } \sigma \subseteq \rho \ \text{ then } \ \tau \rightarrow \sigma \subseteq \tau \rightarrow \rho.$$

However, $\rightarrow$ is *antimonotonic* in its first argument, i.e.

$$\text{if } \sigma \subseteq \rho \ \text{ then } \ \rho \rightarrow \tau \subseteq \sigma \rightarrow \tau$$

rather than the reverse inclusion. If every value of type σ can be treated as a value of type ρ, then every function which maps ρ to τ also maps σ to τ. For example, if f is a function of one real argument, and integers are coercible to reals, then f should be applicable to all integer values. Some "domain-theoretic" semantics of $\rightarrow$ are carefully constructed so that $\rightarrow$ is monotonic in both arguments, since this is helpful in solving domain equations [Scott 76, Smyth and Plotkin 82]. However, antimonotonicity in the first argument is the standard, categorical view of function spaces [Lambek and Scott 86, Mac Lane 71], and seems most natural when we think of containment as either substitutivity or the ability to coerce.

Types will be interpreted as arbitrary sets of elements of lambda models, as in previous studies such as [Barendregt, Coppo and Dezani 83, Hindley 83a]. A *type environment* η for a model <D, $\cdot$, ε> is a mapping from type variables to subsets of D. The meaning of a type expression σ in a type environment η is defined inductively by

$$[\![t]\!]\eta = \eta(t)$$

$$[\![\sigma \rightarrow \tau]\!]\eta = \{d \mid \forall\, e \in [\![\sigma]\!]\eta,\ d{\cdot}e \in [\![\tau]\!]\eta\}.$$

This is the "simple semantics" for $\rightarrow$. Note that membership in $\sigma\rightarrow\tau$ is determined only by the applicative behavior of an element d, so that

$$d \in [\![\sigma\rightarrow\tau]\!]\eta \quad \text{iff} \quad \varepsilon{\cdot}d \in [\![\sigma\rightarrow\tau]\!]\eta.$$

The simple semantics will be used because this seems to be a natural and representative interpretation of $\rightarrow$; other semantics for $\rightarrow$ are discussed in [Hindley 83a, Hindley 83b, MacQueen and Sethi 82, Mitchell 88, Scott 76].

A *coercion set* C is a set of subtype assertions $\sigma\subseteq\tau$ between types. A model <D, $\cdot$, ε> and type environment η satisfy a coercion set C if

$$[\![\sigma]\!]\eta \subseteq [\![\tau]\!]\eta \quad \text{for all } \sigma\subseteq\tau \in C.$$

Generally speaking, coercion sets may include statements like $(t\rightarrow t)\subseteq t$, which are closer to "domain equations" than the simple containments like **int$\subseteq$real** which are often given as typical examples of coercions. In fact, some coercion sets allow us to type every pure lambda term. For example, the two coercions

$$(t \rightarrow t) \subseteq t \quad \text{and} \quad t \subseteq (t \rightarrow t),$$

imply that $t=t\rightarrow t$. Since every solution of this equation forms a model of untyped lambda calculus [Barendregt 84, Scott 76, Scott 80, Smyth and Plotkin 82], we would expect to type every term with these two coercions. This is borne out in Lemma 2. In later sections of the paper, we will develop a typing algorithm which only allows coercions between atomic types, and therefore only types expressions which are typable without coercions.

A *type assignment* A is a finite set of basic typing statements of the form $x{:}\sigma$. An environment η mapping type variables to subsets of D and ordinary variables to elements of D satisfies a type assignment A if

$$\eta(x) \in [\![\sigma]\!]\eta \quad \text{for all} \quad x{:}\sigma \in A.$$

If x is a variable, σ a type expression and A a type assignment, then $A[x{:}\sigma]$ is the type assignment given by $A[x{:}\sigma] = (A - \{x{:}\tau\}) \cup \{x{:}\sigma\}$ if we have $x{:}\tau \in A$, and $A[x{:}\sigma] = A \cup \{x{:}\sigma\}$ otherwise.

A *typing statement* describes the type of an expression, given coercions between types and the types of variables. Informally, the statement

$$C,A \supset M{:}\sigma$$

means that if types may be coerced according to C, and free variables have the types assigned by A, then the term M has type σ. More formally, a model $\langle D, \cdot, \varepsilon \rangle$ and environment η mapping term variables to elements of D and type variables to subsets of D *satisfy* a typing $M{:}\sigma$ if

$$[\![M]\!]\eta \in [\![\sigma]\!]\eta.$$

A statement $C,A \supset M{:}\sigma$ holds (or, is satisfied) in a model if every environment which satisfies C and A also satisfies $M{:}\sigma$. A statement is *valid* if it holds in every model.

4. Rules for Type Inference

4.1. Overview of the rules

We will consider six type inference rules. The first three are essentially Curry's rules for functional types [Curry and Feys 58]. The next rule, (coerce), formalizes the property that if a term M has type σ, and the type σ is coercible to the type τ, then M also has type τ. These four rules will be called the *Curry rules with coercions*, or **CC** for short, and will provide the basis for the typing algorithm G in Section 6. As in [Gordon, et. al. 79, Milner 78], rules for other applicative programming language constructs may be added. However, the main issues involved in treating coercions seem adequately illustrated by the system we will consider.

The Curry rules with coercions are not semantically complete. If we want semantic completeness, then untyped terms that have the same meaning must be given the same types. The simplest way to achieve this is to add a fifth rule (equal) based on equality of untyped terms. However, we will see that with the "simple semantics" of $\rightarrow$, rule (equal) still does not give us semantic completeness with respect to certain equational theories. Therefore, we will also consider a sixth rule allowing us to η-reduce terms of functional types. The six typing rules define *Curry typing with containment and equality*, or $\mathbf{CC}_{\mathrm{eq}}$. Although (equal) is not a recursive inference rule, since no nontrivial lambda theory is decidable, we could replace (equal) by a set of recursive, schematic rules based on the usual axioms and inference rules for lambda theories. However, there does not seem to be any advantage of doing so. As we will see later on, the set of valid typings is undecidable.

4.2. Rules for deducing coercions

The rule (coerce) for coercing terms from one type to another will use two subsidiary rules for deducing consequences of coercion sets. The axiom and rules for deriving coercions are

(ref) $\quad \sigma \subseteq \sigma,$

(arrow) $\quad \dfrac{\sigma_1 \subseteq \sigma, \;\; \tau \subseteq \tau_1}{\sigma \rightarrow \tau \subseteq \sigma_1 \rightarrow \tau_1}$

(trans) $\quad \dfrac{\sigma \subseteq \tau, \;\; \tau \subseteq \rho}{\sigma \subseteq \rho}$

It is easy to verify the soundness of these rules. A coercion $\sigma \subseteq \rho$ is *provable from* C, written

$$C \vdash \sigma \subseteq \rho,$$

if $\sigma \subseteq \rho$ can be derived from formulas in C using (ref), (arrow) and (trans). We will write $C \vdash C'$ if $C \vdash \sigma \subseteq \tau$ for every $\sigma \subseteq \tau \in C'$. It is easy to show that $\vdash$ is a transitive relation on sets.

LEMMA 1. *If $C \vdash C'$ and $C' \vdash C''$, then $C \vdash C''$.*

4.3. Curry typing with coercions

Three well-known rules for assigning types to lambda terms [Curry and Feys 58, Damas and Milner 82, Hindley 83a, Hindley 83b] are

(var) $C,A \supset x{:}\sigma$ whenever $x{:}\sigma \in A$,

$$\text{(app)}\quad \frac{C,A \supset M{:}\sigma \to \tau,\quad C,A \supset N{:}\sigma}{C,A \supset MN{:}\tau}$$

$$\text{(abs)}\quad \frac{C,A[x{:}\sigma] \supset M{:}\tau}{C,A \supset \lambda x.M{:}\sigma \to \tau}$$

These three rules are called the Curry typing rules (except that they are usually written without coercion sets). The coercion rule for typing lambda terms, based on the rules for deducing coercions, is

$$\text{(coerce)}\quad \frac{C,A \supset M{:}\sigma,\quad C \vdash \sigma \subseteq \tau}{C,A \supset M{:}\tau}$$

The four rules (var), (app), (abs) and (coerce) are called the Curry rules with coercions, or **CC** for short. We will write $\vdash C,A \supset M{:}\sigma$ if this typing statement is provable from the **CC** rules. The soundness of the rules is left to the reader.

It is easy to show that the **CC** rules are conservative over the Curry rules in the sense that if $\vdash \emptyset,A \supset M{:}\sigma$, then this typing statement may be proved without using rule (coerce). For this reason, we will call a typing statement of the form $\emptyset,A \supset M{:}\sigma$ a *Curry typing*. Whenever a term M has a Curry typing, M must be *strongly normalizing,* which means that there is no infinite sequence of reductions starting from M (cf. [Hindley 83a]). In contrast, every untyped lambda term may be assigned a type using coercions.

LEMMA 2. *Let M be any untyped lambda term and let A be a type assignment with $x{:}t \in A$ for every variable x free in M. Let C be the coercion set $C = \{t \subseteq (t \to t),\ (t \to t) \subseteq t\}$. Then $\vdash C,A \supset M{:}t$.*

The proof is a straightforward induction on the structure of terms, using $t \subseteq (t \to t)$ in the application case, and $(t \to t) \subseteq t$ for an abstraction. A related translation of untyped terms into typed terms with $t = t \to t$ is given in [Scott 80].

A useful fact about the typing rules is that all free variables must be given types, and types given to variables that do not appear free are irrelevant.

LEMMA 3. *If* $\vdash C,A \supset M{:}\sigma$, *and x occurs free in M, then* $x{:}\tau \in A$ *for some* τ. *Furthermore, if* $x{:}\tau \in B$ *for every* $x{:}\tau \in A$ *with x free in M, then* $\vdash C,B \supset M{:}\sigma$.

The lemma is proved by an easy induction on typing derivations.

An interesting property of the Curry and coercion rules is the following generalization of the Subject Reduction Theorem of [Curry and Feys 58]. The lemma shows that types, as defined by **CC**, are closed under β,η-reduction (but *not conversion*). The lemma is interesting in itself, and will be used to derive some useful corollaries to the completeness theorem.

LEMMA 4. *(Subject Reduction Lemma) If* $\vdash C,A \supset M{:}\sigma$ *and M* β,η*-reduces to N, then* $\vdash C,A \supset N{:}\sigma$.

PROOF . Let us assume for the moment that the lemma holds in the special case that M is a redex and N is obtained by contracting M. We will first argue that the lemma as stated follows from this special case, and then justify the assumption. Suppose M is a term with a subterm P which is a β- or η-redex. We can write M = **C**[P] for some context **C**[] with a single "hole". Let N be the result of contracting the redex P in M. It is easy to show by induction on the structure of the context **C**[] that if $C,A \supset M{:}\sigma$ is provable, then so is $C,A \supset N{:}\sigma$. Thus whenever M reduces to N by a single reduction step, the lemma holds. In general, M may reduce to N by more than one reduction step. By induction on the length of the reduction path, we can prove the lemma. It now suffices to prove the lemma in the special case that M is a redex.

We consider η-reduction first. Assume that the statement $C,A \supset \lambda x.Mx : \sigma$ is provable for x not free in M. We wish to show that $C,A \supset M{:}\sigma$ is provable. For some type τ, there is a proof of $C,A \supset \lambda x.Mx{:}\tau$ which ends in a use of rule (abs) and such that $C \vdash \tau \subseteq \sigma$. Since the proof of $C,A \supset \lambda x.Mx{:}\tau$ ends in a use of rule (abs), the type τ must be of the form $\tau_1 \rightarrow \tau_2$. Furthermore, we must have a proof of the antecedent of (abs),

$$C,\ A[x{:}\tau_1] \supset Mx{:}\tau_2.$$

It follows that for some type ρ with $C \vdash \rho \subseteq \tau_2$, there is a proof of

$$C,\ A[x{:}\tau_1] \supset Mx{:}\rho$$

that ends in a use of rule (app). Hence, for some type υ, the statements

$$C, A[x{:}\tau_1] \supset M{:}\upsilon \longrightarrow \rho$$

and

$$C, A[x{:}\tau_1] \supset x{:}\upsilon$$

are both provable.

Since the typing C, $A[x{:}\tau_1] \supset x{:}\upsilon$ is provable, it is easy to argue that $C \vdash \tau_1 \subseteq \upsilon$. Thus, by (arrow),

$$C \vdash \upsilon \longrightarrow \rho \subseteq \tau_1 \longrightarrow \tau_2$$

and so by rule (trans)

$$C \vdash \upsilon \longrightarrow \rho \subseteq \sigma.$$

From this, we may use Lemma 3 to conclude that there is a proof of $C,A \supset M{:}\sigma$.

The remaining case is β-reduction. Assume that $C,A \supset (\lambda x.M)\ N{:}\ \sigma$ is provable, for some terms M and N. We wish to conclude that $C,A \supset [N/x]M{:}\ \sigma$ is provable. There is some type τ with $C \vdash \tau \subseteq \rho$ such that $C,A \supset (\lambda x.M)\ N{:}\ \tau$ has a proof that ends in a use of rule (app). Thus for some type ρ, we have proofs of

$$C,A \supset \lambda x.M{:}\ \rho \longrightarrow \tau$$

and

$$C,A \supset N{:}\ \rho.$$

A straightforward induction on the structure of M shows that if $C,A[x{:}\rho] \supset M{:}\upsilon$ and $C,A \supset N{:}\ \rho$ are provable then so is $C,A \supset [N/x]M{:}\upsilon$. This implies that $C,A \supset [N/x]\ M{:}\sigma$ is derivable, and finishes the proof of the lemma.

4.4. Curry typing with coercions and equality

As mentioned earlier, the **CC** rules are not semantically complete. This follows from the fact that without coercion hypotheses, only the Curry typing rules apply, and these are not semantically complete. More specifically, a term M can have a Curry typing only if there is no infinite sequence of reductions from M (cf. [Hindley 83a]). Therefore, although the expression

$$(\lambda x.\lambda y.y)\ ((\lambda x.xx)\ \ (\lambda x.xx)),$$

is semantically equally to λy.y, we cannot type this term since the subterm (λx.xx)(λx.xx) may be reduced infinitely many times. More precisely, the typing statement

$$\emptyset,\emptyset \supset\ (\lambda x.\lambda y.y)\ ((\lambda x.xx)\ \ (\lambda x.xx)):t \longrightarrow t$$

is semantically valid, but cannot be proved using the **CC** typing rules. Another way to show that the **CC** rules are not semantically complete is to compare Theorem 5, which shows that the valid typing statements are undecidable, against Theorems 13 and 14, which imply that the consequences of the **CC** rules are decidable.

We will give equal terms the same types by adopting the rule

$$\text{(equal)}\quad \frac{C,A \supset M:\sigma,\ \ M=N}{C,A \supset N:\sigma}$$

If we are interested in deducing typings that hold in all models, then we use β-conversion for = in (equal). To consider only extensional models, we would use β,η-conversion instead. In both of these cases, the **CC** rules together with (equal) are semantically complete. If we wish to consider the typing statements that hold in all models of some lambda theory *Th*, then we would use *Th* for equality in (equal).

For certain lambda theories, even **CC**+(equal) may not be semantically complete. The reason is that in the "simple semantics" of $\longrightarrow$, a term M has a functional type $\sigma \longrightarrow \tau$ iff λx.Mx has type $\sigma \longrightarrow \tau$. This implies that the rule

$$\text{(eta)}\quad \frac{C,A \supset \lambda x.Mx:\sigma \longrightarrow \tau}{C,A \supset M:\sigma \longrightarrow \tau}\quad \text{x not free in M}$$

is sound. However, the set of typing statements derivable using **CC**+(equal) from *Th* may not be closed under rule (eta). For example, if we assume that kx=x for some constant k, then we can prove λx.kx:t$\longrightarrow$t. However, we cannot prove k:t$\longrightarrow$t without rule (eta), as explained at the end of this paragraph. Therefore, we will adopt (eta) as an additional inference rule. It is worth remarking that the need for rule (eta) stems only from the choice of simple semantics for $\longrightarrow$, and is not related to the presence of coercions. The reason we cannot prove k:t$\longrightarrow$t without (eta) is that **CC**+(equal) are sound for the F-semantics of $\longrightarrow$, discussed in [Hindley 83a, Hindley 83b], while kx=x does not imply k:t$\longrightarrow$t in the F-semantics. It is worth pointing out that although this example is stated using a constant symbol k, this is for notational simplicity only. The same reasoning applies if we replace the constant k with the closed term (λx.xx)(λx.xx).

The **CC** rules with (equal) and (eta) will be called the *Curry rules with coercions and equality*, and abbreviated $\mathbf{CC}_{eq}$. We will write $\vdash_{eq}$ C,A $\supset$ M:σ if the typing statement is provable using the $\mathbf{CC}_{eq}$ rules, with β-conversion in rule (equal). Similarly, if C,A $\supset$ M:σ is provable from $\mathbf{CC}_{eq}$ using an equational theory *Th*, we will write *Th* $\vdash_{eq}$ C,A $\supset$ M:σ. It is clear that if *Th* is closed under η-conversion, then rule (eta) is not needed. We will also see that if *Th* is the theory of β-conversion, then (eta) is superfluous.

Since we have introduced undecidable equational reasoning into the typing rules, it seems worthwhile to point out that the set of semantically valid typing statements is undecidable.

THEOREM 5. *The set of valid typing statements of the form C,A $\supset$ M:t, with C=$\varnothing$ and A={x:t}, is not recursive.*

PROOF . This theorem follows from the classical undecidability results in lambda calculus, which imply that for any variable x, the set of terms M such that M $=_\beta$ x is undecidable [Barendregt 84]. Specifically, if $\varnothing$,{x:t}$\supset$M:t is valid, then this typing statement must hold in the term model for β-conversion, with type t assigned to the singleton set containing only the equivalence class [x] of x. Therefore, the typing statement $\varnothing$,{x:t}$\supset$M:t is valid iff M $=_\beta$ x.

If we replace (eta) by a more general proof step, then we can generalize the Equality Postponement Lemma for Curry typing [Hindley 83a] to show that in any typing derivation, all uses of (equal) and (eta) may be "postponed" until the end of the proof. One application of this will be to show that rule (eta) is unnecessary if *Th* is β-conversion; some other applications will be discussed at the end of this section. An unsound "inference rule" that will facilitate proof-theoretic analysis is

$$(\text{eta})_{\text{trick}} \quad \frac{\text{C,A} \supset \text{M}:\sigma\to\tau}{\text{C,A} \supset \text{N}:\sigma\to\tau} \quad \text{M} \twoheadrightarrow_\eta \text{N}$$

which includes (eta) as a special case. We will say that *η-reduction passes through Th* if M $\twoheadrightarrow_\eta$ N and *Th* $\vdash$ N=P imply that there exists a term Q with *Th* $\vdash$ M=Q and Q $\twoheadrightarrow_\eta$ P. It follows from Corollary 15.1.6 of [Barendregt 84] that η-reduction passes through the theory of β-conversion. We now have the following useful but rather elaborate lemma.

LEMMA 6. *(Equality and Eta Postponement) Suppose* $Th \vdash_{eq} C,A \supset M{:}\sigma$ *and* Δ_1 *is a derivation of this typing statement using Th. Then there is a derivation* Δ_2 *of the same typing statement, possibly using* $(eta)_{trick}$, *with the following properties.*

(i) Δ_2 has the same number of occurrences of each typing rule as Δ_1, provided we consider $(\mathrm{eta})_{\mathrm{trick}}$ an acceptable replacement for (eta)

(ii) All **CC** rules appear before any occurrences of (equal) or (eta) in Δ_2.

If η-reduction passes through *Th*, then we may choose Δ_2 so that, in addition,

(iii) all occurrences of (equal) appear before any uses of $(\mathrm{eta})_{\mathrm{trick}}$.

If (eta) does not occur in the derivation Δ_1, then from (ii) and the transitivity of equality, we may coalesce all uses of (equal) in Δ_2 into one. Similarly, when (iii) applies, we may assume Δ_2 contains at most one occurrence of (equal) followed by at most one use of $(\mathrm{eta})_{\mathrm{trick}}$.

It is important to emphasize that $(\mathrm{eta})_{\mathrm{trick}}$ is merely a technical device for analyzing typing derivations, and is *not* a sound typing rule with respect to arbitrary equational theories. Therefore, care must be taken in applying this lemma.

PROOF . The proof is a straightforward induction on the derivation of $C,A \supset M{:}\sigma$, and some details will be left to the reader. If we have a use of (equal) preceding any **CC** rule, then it is easy to produce an valid derivation with the order of the two rules reversed. For example, if Δ_1 proves the following sequence of typings

$C,A \supset M{:}\sigma\to\tau$

$C,A \supset N{:}\sigma\to\tau$ by (equal) using $M=N$

$C,A \supset NP{:}\tau$ by (app) using $C,A \supset P{:}\sigma$,

then we can replace this sequence by

$C,A \supset M:\sigma \rightarrow \tau$

$C,A \supset MP:\tau$ by (app) using $C,A \supset P:\sigma$

$C,A \supset NP:\tau$ by (equal) using MP=NP

The other cases are similar.

If we have a use of rule (eta) followed by any **CC** rule, then we may also switch the order of rules, provided we allow $(eta)_{trick}$ in place of (eta). The argument is similar to that for (equal). This shows that we may transform Δ_1 into a derivation Δ_2 satisfying conditions (i) and (ii) of the lemma.

If η-reduction passes through *Th*, then we may further simplify the sequence of $(eta)_{trick}$ and (equal) rules following the **CC** rules in Δ_2. More specifically, if $(eta)_{trick}$ is followed by (equal) to prove a sequence of typing statements of the form

$C,A \supset M:\sigma \rightarrow \tau$

$C,A \supset N:\sigma \rightarrow \tau$ by (eta) using $M \twoheadrightarrow_\eta N$

$C,A \supset P:\sigma \rightarrow \tau$ by (equal) $Th \vdash N=P$

then the assumption that η-reduction passes through *Th* is precisely what we need to reverse the order of the two rules.

An interesting corollary of Lemma 4 and Lemma 6 is that for typing with respect to the theory of β-conversion, rule (eta) is unnecessary.

COROLLARY 7. *If* $\vdash_{eq}$ *C,A* $\supset$ *M:*σ*, then this typing statement may be proved from the theory of* β*-conversion without using rule (eta).*

A similar corollary is proved in [Hindley 83a], following the Subject Reduction Theorem, using similar facts about reduction.

PROOF . Suppose $\vdash_{eq} C,A \supset M:\sigma$. Since η-reduction passes through β-conversion, Lemma 6 implies that there exist terms N and P such that $\vdash C,A \supset N:\sigma$ by the **CC** rules only, $N =_\beta P$ and $P \twoheadrightarrow_\eta M$. Using well-known properties of β- and η-reduction we will show that there is a term V such that $N \twoheadrightarrow_{\beta,\eta} V$ and $V =_\beta M$. This will allow us to apply the subject reduction lemma.

By the Chruch-Rosser property of β-conversion, there is some term U with N $\twoheadrightarrow_\beta$ U and P $\twoheadrightarrow_\beta$ U. Thus P may be reduced to M by (η), or to U by (β). Since (β) and (η) commute (Lemma 3.3.8 of [Barendregt 84]), there is some term V with M $\twoheadrightarrow_\beta$ V and U $\twoheadrightarrow_\eta$ V. Putting the reduction paths together, we obtain N $\twoheadrightarrow_{\beta,\eta}$ V and M $\twoheadrightarrow_\beta$ V, as desired.

By the subject reduction lemma for the **CC** rules, we may conclude $\vdash$ C,A $\supset$ V:σ. Therefore, from M $=_\beta$ V, we have $\vdash_{eq}$ C,A $\supset$ M:σ without using (eta).

4.5. Semantic completeness

We will now prove the semantic completeness theorem for $\mathbf{CC}_{eq}$. Since (coerce) is unnecessary for proving Curry typings, our theorem implies that Curry's rules, augmented with (equal) and (eta), are complete for Curry typing with respect to any equational theory. In addition, by Corollary 7, Hindley's completeness theorem for Curry typing with β- or β,η-conversion as equality, but without rule (eta), also follows.

THEOREM 8. *Let Th be any lambda theory. There is a lambda model* **D** *for Th such that Th* $\vdash$ *C,A* $\supset$ *M:*σ *iff the typing statement C,A* $\supset$ *M:*σ *holds in every environment for* **D**.

In contrast to the theorem stated earlier in [Mitchell 84a], this completeness theorem applies to typing with unrestricted coercions. However, the proof is quite similar.

PROOF . The first step is to construct a model **D** for *Th*. Let <D, $\cdot$, ε> be the term model for *Th*, so that D is the collection of all equivalence classes $[M]_{Th}$ of terms modulo *Th*. As mentioned earlier, we will write [M] for the equivalence class $[M]_{Th}$. A standard property of term models [Barendregt 84, Meyer 82] is that if η is any environment, and S is a substitution such that η(x) = [Sx] for every term variable x, then

$$[\![M]\!]\eta = [SM].$$

It turns out that we will only need to consider one environment in the proof, namely an environment mapping each term variable x to its equivalence class.

Let $C_0,A_0 \supset M_0{:}\sigma_0$ be any typing statement. It is easy to verify that the typing rules are sound, so that if *Th* $\vdash_{eq}$ $C_0,A_0 \supset M_0{:}\sigma_0$, then this statement must hold in every environment for <D, $\cdot$, ε>. Therefore, we will assume that $C_0,A_0 \supset M_0{:}\sigma_0$ is not provable.

The rest of the proof will be devoted to showing that there is some environment η satisfying C_0 and A_0 but with $[\![M_0]\!]\eta \notin [\![\sigma_0]\!]\eta$. We will do this by choosing an infinite type assignment A containing A_0 and using the proof system to define an environment η from A.

Let A be any set of basic typing statements $x{:}\sigma$, with no x appearing twice in A, such that

(i) $A_0 \subseteq A$

(ii) for every type σ, there are infinitely many variables x with $x{:}\sigma \in A$.

The reason for having infinitely many variables of each type is so that given any term M and type σ, we can find some $x{:}\sigma \in A$ with x not free in M. We will extend our notation slightly and write $Th \vdash C,A \supset M{:}\sigma$ if $Th \vdash_{eq} C,A_1 \supset M{:}\sigma$ for some finite subset A_1 of A. Let η be an environment which maps each term variable x to its equivalence class [x], and each type variable t to the subset of D given by

$$\eta(t) = \{\, [M] \mid Th \vdash_{eq} C,A \supset M{:}t \}.$$

We will see that η satisfies A_0 and C_0, and that the rules are complete, by showing that

$$(*) \qquad [M] \in [\![\sigma]\!]\eta \ \text{ iff } \ Th \vdash_{eq} C,A \supset M{:}\sigma.$$

The argument will proceed by induction on the structure of type expressions.

For a type variable t, the equivalence is a trivial consequence of the definition. For any functional type $\sigma \rightarrow \tau$, suppose that the statement

$$C,A \supset M{:}\ \sigma \rightarrow \tau$$

is provable from *Th*. We must show that [M] belongs to $[\![\sigma \rightarrow \tau]\!]\eta$. For any term N, if $[N] \in [\![\sigma]\!]\eta$, then by the inductive hypothesis there is a proof of $C,A \supset N{:}\sigma$, and so we can prove $C,A \supset MN{:}\tau$ by rule (app). Therefore, by the inductive hypothesis, $[MN] \in [\![\tau]\!]\eta$. Thus, by definition of $[\![\sigma \rightarrow \tau]\!]\eta$, we have $[M] \in [\![\sigma \rightarrow \tau]\!]\eta$.

For the converse, assume that $[M] \in [\![\sigma \rightarrow \tau]\!]\eta$. For any term N, if $C,A \supset N{:}\sigma$ is provable, then $[N] \in [\![\sigma]\!]\eta$ by the inductive hypothesis, and so

$$[M] \cdot [N] = [MN] \in [\![\tau]\!]\eta.$$

Thus $Th \vdash C,A \supset MN{:}\tau$, again by the induction hypothesis. In particular, if x is any variable with x:σ∈A, we can use this argument to show that

$$C,A \supset Mx:\tau.$$

and so, by rule (abs),

$$Th \vdash_{eq} C,A \supset \lambda x.Mx : \sigma \longrightarrow \tau.$$

By the construction of A, we may choose x to be a variable not free in M. Therefore, we may use rule (eta) to derive $Th \vdash_{eq} C,A \supset M{:}\sigma \longrightarrow \tau$, which finishes the proof of (*).

It is now easy to see that η satisfies A_0 and C_0. If x:$\sigma \in A_0$, then x:σ∈A and so

$$[\![x]\!]\eta = [x] \in [\![\sigma]\!]\eta$$

by (*). If $\sigma \subseteq \tau \in C_0$, then for every $[M] \in [\![\sigma]\!]\eta$, we have $Th \vdash_{eq} C_0,A \supset M{:}\sigma$ by (*). Therefore, by rule (coerce), we have $Th \vdash_{eq} C_0,A \supset M{:}\tau$ and so $[M] \in [\![\tau]\!]\eta$. Thus η satisfies C_0. Finally, using (*) again, we have $[M_0] \notin [\![\sigma_0]\!]\eta$, and so the unprovable typing statement $C_0,A_0 \supset M_0{:}\sigma_0$ does not hold in environment η. This proves the theorem.

The completeness theorem has two important implications for **CC** typing: **CC** is semantically complete for terms in normal form, and the three containment rules are complete for deducing consequences of coercion sets.

COROLLARY 9. *If M is in β-normal form and C,A ⊃ M:σ holds in all lambda models, then* $\vdash C,A \supset M{:}\sigma$.

PROOF . Suppose C,A ⊃ M:σ holds in all models. Then by Corollary 7, $\vdash_{eq} C,A \supset M{:}\sigma$ without using rule (eta). By the equality postponement lemma, there is some N which is β-equivalent to M with $\vdash C,A \supset N{:}\sigma$. But since M is in β-normal form, N must reduce to M. Therefore, by the subject reduction lemma, it follows that $\vdash C,A \supset M{:}\sigma$.

COROLLARY 10. *If* $\sigma \subseteq t$ *holds in every model and environment satisfying coercion set C, then* $C \vdash \sigma \subseteq \tau$.

PROOF . Note that if C semantically implies $\sigma \subseteq \tau$, then C,{x:σ} ⊃ x:τ must be valid for any variable x. Since x is in normal form, this typing statement is provable using rule (var), (app), (abs) and (coerce). But then it is easy to see that the only applicable rules are (var) and (coerce). Thus $C \vdash \sigma \subseteq \tau$.

The proofs of both corollaries rely on equality postponement and the subject reduction lemma. Although rule (arrow) is not used in the proof of Theorem 8, it is used critically in the proof of subject reduction.

5. Typing Algorithm with Unrestricted Coercions

5.1. Introduction

Since semantic typing characterized by $\mathbf{CC}_{eq}$ and any nontrivial lambda theory is undecidable, it is impossible to build a practical type checker based on $\mathbf{CC}_{eq}$. However, **CC** forms a natural subset of the $\mathbf{CC}_{eq}$ system, and we will see that there is an efficient algorithm for **CC** typing. While it might seem that **CC**+(eta) would also provide a reasonable basis for practical type checking, recall that by Lemma 4, the consequences of the **CC** rules are closed under rule (eta). Therefore, we will study algorithmic properties of **CC** typing in the remainder of the paper. Section 5 will be concerned with unrestricted **CC** typing. In Section 6 we will consider a restriction of **CC** typing in which only atomic containment hypotheses are allowed.

One plausible approach to type inference with subtyping might be to provide a term M and coercion set C as input to a typing algorithm, and then compute a description of the set of all A and σ such that $\vdash C, A \supset M{:}\sigma$. However, this approach does not seem fruitful. The main problem is that there does not seem to be any succinct, understandable description of all suitable types and type assignments. If every term had a semantically minimal type, in each context, then this would be a natural way of characterizing all other types. However, typable terms do not have have minimal types in each context. To give a simple example, let us suppose we have expression and type constants; the same phenomenon occurs without this assumption, but slightly less obviously. Now suppose the only coercion is $int \subseteq real$, and that function constant f has type $real \rightarrow bool$. In this context, the expression

$$\lambda x.K\ x\ (fx),$$

where $K = \lambda u.\lambda v.u$ returns its first argument and discards its second, has types $int \rightarrow int$, $int \rightarrow real$, and $real \rightarrow real$. It is easy to see that $int \rightarrow int$ is contained in $int \rightarrow real$, but from Corollary 10 we can see that, neither $int \rightarrow int \subseteq real \rightarrow real$, nor the reverse containment, is semantically valid. Therefore, there is no semantically minimal type to use as a representation of all other types. This suggests that the seemingly natural approach of leaving the set of coercions fixed is impractical. Instead, our type inference algorithm will compute a minimal set of coercions necessary to type a given term. Although there is a certain computational cost associated with this, since sets of coercions must be manipulated, there is an added generality which provides some insight into typing.

The general typing algorithm presented in this section is a straightforward generalization of the special case presented in [Mitchell 84a] to arbitrary coercion sets. Algorithm GA of this paper, which only allows a restricted form of coercion set, corresponds to the original Algorithm TYPE of [Mitchell 84a].

5.2. Substitutions, instances and most general typings

A useful property of Curry typing is that the provable typings are closed under substitution. **CC** typings are not only closed under substitution, but also a more general relation involving entailment of coercion sets. After a brief discussion of substitution, we will define "instance" and show that every instance of a provable typing is provable. The correctness proof for algorithm G will later establish that every term has a "most general typing" with all alternative typings as instances.

A *substitution* is a function from type variables to type expressions. We write $[\sigma_1,...,\sigma_n/t_1,...,t_n]$ for the substitution mapping t_i to σ_i, for $1 \le i \le n$, and mapping every other type variable to itself. If σ is a type expression and S is a substitution, then $S\sigma$ is the type expression obtained by replacing each variable t in σ with S(t). The composition $S \circ T$ of substitutions S and T is defined by $(S \circ T)\sigma = S(T\sigma)$.

A substitution S applied to a type assignment A is the assignment SA with

$$SA = \{x{:}S\sigma \mid x{:}\sigma \in A\}.$$

Similarly, the application SC of substitution S to coercion set C is defined to be the following set of subtype assertions.

$$SC = \{S\sigma \subseteq S\tau \mid \sigma \subseteq \tau \in C\}.$$

An instance of a typing statement may be obtained by applying a substitution to all of its type expressions, and possibly choosing a "stronger" coercion hypothesis or type assignment. More precisely, a typing statement $C', A' \supset M{:}\sigma'$ *is an instance of* $C, A \supset M{:}\sigma$ if there exists a substitution S such that

$$C' \vdash SC, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma.$$

In this case we say C',A' $\supset$ M:σ' *is an instance of* C,A $\supset$ M:σ *by substitution* S. Note that coercion sets are compared using the entailment relation, rather than syntactically. One important fact about instances is that every instance of a provable typing statement is provable.

LEMMA 11. *Suppose C',A' $\supset$ M:σ' is an instance of C,A $\supset$ M:σ. If $\vdash$ C,A $\supset$ M:σ, then the instance C',A' $\supset$ M:σ' is provable also.*

PROOF . By Lemmas 1 and 3, it suffices to show that if $\vdash$ C,A $\supset$ M:σ, then $\vdash$ SC,SA $\supset$ M:Sσ for any substitution S. An easy induction on coercion proofs shows that if C $\vdash$ $\sigma \subseteq \tau$, then SC $\vdash$ S$\sigma \subseteq$Sτ. Using this fact for the (coerce) case, a straightforward induction on the derivation of C,A $\supset$ M:σ proves the lemma. The details are left to the reader.

A *most general typing for term* M is a provable typing statement which has every other provable typing for M as an instance. More specifically, C,A $\supset$ M:σ is a most general typing for M if $\vdash$ C,A $\supset$ M:σ and, whenever $\vdash$ C',A' $\supset$ M:σ', the latter typing is an instance of C,A $\supset$ M:σ. Consequently, if C,A $\supset$ M:σ is a most general typing for M, then $\vdash$ C',A' $\supset$ M:σ' iff C',A' $\supset$ M:σ' is an instance of C,A $\supset$ M:σ. Since the instance relation is easily seen to be decidable, the decision problem for **CC**-typing is effectively reducible to the problem of computing most general typings.

Without coercions, a most general Curry typing is unique except for the names of type variables. In addition, since substitutions cannot decrease the size of expressions, every most general Curry typing is a Curry typing of minimal length (when written out symbolically). However, because coercion sets are compared using entailment, there may be most general **CC** typings of differing lengths. For example, both

$$\{s\subseteq t,\ u\subseteq v\},\ \emptyset \supset \lambda x.x{:}u\rightarrow v \quad \text{and} \quad \{u\subseteq v\},\ \emptyset \supset \lambda x.x{:}u\rightarrow v$$

are most general **CC** typings for the identity function. The first is easily seen to be an instance of the second (by the identity substitution), since

$$\{s\subseteq t,\ u\subseteq v\} \vdash u\subseteq v.$$

Conversely, the second may be obtained as an instance of the first by substituting s for t. This reduces s$\subseteq$t to an instance of the reflexivity axiom s$\subseteq$s.

5.3. Unification

A unifier is substitution which makes two expressions syntactically equal. More generally, if E is a set of pairs of expressions, then substitution S *unifies* E if $S\sigma = S\tau$ for every pair $\langle\sigma,\tau\rangle \in E$. Since such a set of pairs may be regarded as set of equations to be solved, we often write the pairs $\langle\sigma,\tau\rangle \in E$ in the form $\sigma = \tau$. As in ML typing [Milner 78], we will use unification to combine typing statements about subexpressions. Although we will see that the unification problems involved in Algorithm G have very simple solutions, more difficult unification problems will occur in the specialized versions of the algorithm considered in later sections of the paper. Since we will need a general unification algorithm eventually, it makes sense to start right off with one here.

The unification algorithm computes most general unifying substitutions, where we say substitution S *is more general than* R, and write $S \le R$, if there is a substitution T with $R = T \circ S$.

LEMMA 12.

[Robinson 65] Let E be any set of equations between type expressions. There is an algorithm UNIFY *such that if E is unifiable, then* UNIFY(E) *computes a most general unifier. Furthermore, if E is not unifiable, then* UNIFY(E) *fails.*

If A_1 and A_2 are type assignments, then unification can be used to find a most general substitution S such that $SA_1 \cup SA_2$ is a well-formed type assignment. Generally speaking, the union of two type assignments is a type assignment precisely when both give each variable in common the same type. Therefore, to find a most general S with $SA_1 \cup SA_2$ well-formed, we simply unify the set of all equations $\sigma = \tau$ such that $x{:}\sigma \in A$ and $x{:}\tau \in B$.

To facilitate comparisons between various containment theories, a proof of Lemma 12 is sketched in the Appendix. There are efficient, even linear, implementations of unification [Paterson and Wegman 78]. A parallel lower bound is given in [Dwork, Kanellakis and Mitchell 84].

5.4. Algorithm G for most general typings

Given any term M, the algorithm G(M) produces a provable typing $C,A \supset M{:}\sigma$ for M. The algorithm is written below in an applicative, pattern-matching style. There are three mutually recursive clauses, one for each possible form of lambda term. In the abstraction clause, we use A -- $\{x{:}\sigma\}$ to denote the set difference, i.e., the type assignment defined by removing $x{:}\sigma$ from A.

$G(x) = \{\ s \sqsubseteq t\ \}, \{x{:}s\} \supset x{:}t$

$G(MN) =$

let $C_1, A_1 \supset M{:}\sigma = G(M)$

$C_2, A_2 \supset N{:}\tau = G(N)$,

with type variables renamed to be disjoint from those in G(M)

$S\ =\ \text{UNIFY}(\{\alpha{=}\beta \mid x{:}\alpha{\in}A_1 \text{ and } x{:}\beta{\in}A_2\} \cup \{\sigma{=}\tau{\longrightarrow}t\})$

where t is a fresh type variable

in

$SC_1{\cup}SC_2{\cup}\{St{\sqsubseteq}u\},\ \ SA_1{\cup}SA_2 \supset MN{:}u$

where u is a fresh type variable

$G(\lambda x.M) =$

let $C, A \supset M{:}\tau = G(M)$

in if $x{:}\sigma{\in}A$ for some σ

then $C{\cup}\{\sigma{\longrightarrow}\tau{\sqsubseteq}u\}, (A \text{ -- } \{x{:}\sigma\}) \supset \lambda x.M{:}u$

else $C{\cup}\{s{\longrightarrow}\tau{\sqsubseteq}u\}, A \supset \lambda x.M{:}u$,

where s, u are fresh type variables

The algorithm could conceivably *fail* in the application case if the call to UNIFY *fails*. However, we will see that this does not happen. It is not to hard to prove that if G(M) succeeds, then it produces a provable typing for M.

THEOREM 13. *If $G(M) =\ C,A \supset M{:}\sigma$, then $\vdash C,A \supset M{:}\sigma$.*

It follows, by Lemma 11, that every instance of G(M) is a provable typing for M. Conversely, every provable typing for M is an instance of G(M).

THEOREM 14. *Suppose $\vdash C,A \supset M{:}\sigma$. Then G(M) succeeds and produces a typing with $C,A \supset M{:}\sigma$ as an instance.*

Both theorems are proved below.

From Lemma 2, we know that every term has a provable **CC** typing. Therefore, Theorem 14 implies G(M) always succeeds.

COROLLARY 15. *For every untyped lambda term M, Algorithm G(M) succeeds in finding a most general typing for M.*

In contrast to the typing algorithm given in [Milner 78], Algorithm G takes a term, but no type assignment, as input. This style of typing algorithm for lambda terms seems to have originated with [Leivant 83a]. The advantage of Algorithm G over Milner's Algorithm W is that the algorithm and the correctness condition are simpler to state. In addition, since the type of a lambda term is determined without regard to context, this style of algorithm facilitates the extension to ML **let** declarations, as presented in Section 7. The disadvantage is that in practice, Algorithm G may compute rather large type assignments which must be unified. In contrast, Milner's Algorithm W may be implemented so that entire type assignments need not be unified or returned as results of function calls. However, it is not very difficult to use Algorithm G to develop an algorithm for **CC** typing in the style of Milner's Algorithm W.

PROOF OF THEOREM 13. The proof is by induction on the structure of terms. It is easy to see that G(x) is always a well-typing, so we move on to application and abstraction.

Consider G(MN). By the inductive assumption, both

$$G(M) = C_1, A_1 \supset M{:}\sigma$$

$$G(N) = C_2, A_2 \supset N{:}\tau$$

are provable. (We will assume that the type variables in G(N) have been renamed to avoid duplicating type variables in G(M), as specified in Algorithm G.) Since S unifies $\{\alpha=\beta \mid x{:}\alpha \in A_1 \text{ and } x{:}\beta \in A_2\}$, the set $SA_1 \cup SA_2$ is a well-formed type assignment. By Lemma 11, it follows that the two typings

$$SC_1 \cup SC_2 \cup \{St \subseteq u\}, SA_1 \cup SA_2 \supset M{:}S\sigma$$

$$SC_1 \cup SC_2 \cup \{St \subseteq u\}, SA_1 \cup SA_2 \supset N{:}S\tau$$

are both provable. Since S unifies σ and $\tau \rightarrow t$, we have

$$SC_1 \cup SC_2 \cup \{St \subseteq u\}, SA_1 \cup SA_2 \supset MN{:}St$$

by rule (app), and hence G(MN) is provable using (coerce).

The third case is an abstraction λx.M. By the inductive assumption,

$$G(M) \;=\; C, A \supset M{:}\tau$$

is provable. If x:$\sigma \in$A for some type σ, then A = (A -- {x:σ})[x:σ] and so by rule (abs) we may derive

$$C, (A \text{ -- } \{x{:}\sigma\}) \supset \lambda x.M{:}\sigma \longrightarrow \tau.$$

If x does not occur in A, then by Lemma 3, the typing C, A[x:t] $\supset$ M:τ is provable, for any type variable t, and so

$$C, A \supset \lambda x.M{:}t \longrightarrow \tau$$

follows by rule (abs). In either case, augmenting the coercion set with $\sigma \longrightarrow t \subseteq u$ or $s \longrightarrow t \subseteq u$ preserves provability (by Lemma 11), and so by rule (coerce) we may prove that λx.M has type u, as desired. This proves the theorem.

PROOF OF THEOREM 14. An easy induction on the structure of terms shows that if G(M) succeeds, then it produces a typing C,A $\supset$ M:σ in which A assigns a type to x iff x occurs free in M. This will be useful later in the proof. The main argument now proceeds by induction on the structure of terms.

For a variable x, suppose $\vdash$ C,A $\supset$ x:τ. Without loss of generality, we may assume the proof uses axiom (var) followed by rule (coerce). Since x must appear in A, we let σ be the type with x:$\sigma \in$A, and note that C must prove $\sigma \subseteq \tau$. Algorithm G returns the typing

$$G(x) \;=\; \{s \subseteq t\}, \{x{:}s\} \supset x{:}t.$$

To show that C,A $\supset$ x:τ is an instance of G(x), let T be the substitution [σ,τ/s,t]. It is easy to check that

$$A \supseteq T\{x{:}s\} \quad \text{and} \quad Tt = \tau.$$

Furthermore, since C $\vdash$ T{s$\subseteq$t}, it follows that C,A $\supset$ x:τ is an instance of G(x).

Suppose $\vdash$ C, A $\supset$ MN:ρ. This typing must follow from provable typings

$$C, A \supset M{:}\mu \longrightarrow \nu$$

$$C, A \supset N{:}\mu$$

by rules (app) and (coerce), where C $\vdash$ $\nu \subseteq \rho$. By the inductive hypothesis, G(M) and G(N) are most general typings for M and N. This means that there exist substitutions T_1 and

T_2 such that

$$C \vdash T_1C_1, \quad A \supseteq T_1A_1, \quad T_1\sigma = \mu \longrightarrow \nu,$$

$$C \vdash T_2C_2, \quad A \supseteq T_2A_2, \quad T_2\tau = \mu,$$

where C_1, C_2, etc., are as in the application case of Algorithm G. Because G renames variables, no type variables in $C_2, A_2 \supset N{:}\tau$ appear in $C_1, A_1 \supset M{:}\sigma$. This allows us to combine substitutions T_1 and T_2. Anticipating the need for a substitution that behaves properly on the fresh variables t and u introduced in Algorithm G, we let T be any substitution such that

$$Ts = T_1 s \text{ if s appears in the typing of M,}$$

$$Ts = T_2 s \text{ if s appears in the typing of N,}$$

$$Tt = \nu,$$

$$Tu = \rho.$$

Without considering the effect of T on t or u, it is easy to see that

$$C \vdash TC_1, \quad A \supseteq TA_1, \quad T\sigma = \mu \longrightarrow \nu,$$

$$C \vdash TC_2, \quad A \supseteq TA_2, \quad T\tau = \mu,$$

so that both instances are by the single substitution T. By Lemma 3, the assignment A must give types to all free variables of M and N and, as noted earlier, an assignment produced by G always contains exactly the variables that occur free. Therefore, T must unify $\{\alpha{=}\beta \mid x{:}\alpha \in A_1 \text{ and } x{:}\beta \in A_2\}$. In addition, since $T\sigma = \mu \longrightarrow \nu = T\tau \longrightarrow Tt$, the substitution T unifies $\sigma{=}\tau{\longrightarrow}t$. Since S is a most general unifier for these equations, there is a substitution V with

$$T = V \circ S.$$

This implies that $C, A \supset MN{:}\nu$ is an instance of $SC_1 \cup SC_2,\ SA_1 \cup SA_2 \supset MN{:}St$ by V. It remains to consider the fresh type variable u and coercion $\nu \subseteq \rho$.

Since S is a most general unifier for a set of equations not containing u, we know Su = u. Since we chose Tu = ρ, it follows that Vu=ρ. We have already seen that V(St) = ν, and so putting the pieces together gives us C$\vdash$ V(St $\rightarrow$ u). This shows that

$$C \vdash V[SC_1 \cup SC_2 \cup \{St \rightarrow u\}],$$

which completes the proof that C,A $\supset$MN:ρ is an instance of G(MN) by substitution V.

The final case to consider is an abstraction λx.M. Suppose$\vdash$ C',A' $\supset$ λx.M:ρ. This must follow from a provable typing C',A'[x:μ] $\supset$ M:ν by (abs) and (coerce), where C'$\vdash$ $\mu \rightarrow \nu \subseteq \rho$. By the inductive hypothesis, C',A'[x:μ] $\supset$ M:ν is an instance of

$$G(M) = C,A \supset M{:}\tau.$$

This means that there is a substitution S such that

$$C' \vdash SC, \quad A'[x{:}\mu] \supseteq SA \quad \text{and} \quad \nu = S\tau.$$

Without loss of generality we may assume Su=ρ. If x:σ occurs in A, then Sσ must be μ, and so C',A' $\supset$ λx.M:$\mu \rightarrow \nu$ is an instance of C,A--{x:σ} $\supset$ λx.M:$\sigma \rightarrow \tau$ by substitution S. In addition, since C'$\vdash$ $\mu \rightarrow \nu \subseteq \rho$ and Su = ρ, the typing C',A' $\supset$ λx.M:ρ is an instance of G(λx.M) by S.

If x does not occur in A, then we may further assume without loss of generality that Ss = μ, where s is the fresh type variable introduced in Algorithm G. It is easy to check the definition and verify that C',A' $\supset$ λx.M:$\mu \rightarrow \nu$ must be an instance of C,A $\supset$ λx.M:s$\rightarrow \tau$ by substitution S. Furthermore, reasoning about coercions as above, we again conclude C',A' $\supset$ λx.M:ρ is an instance of G(λx.M) by substitution S. This proves the theorem.

6. Typing with Atomic Coercions

6.1. Introduction

In this section, we will study **CC** typing with atomic coercions and give an algorithm GA for finding the corresponding form of most general typing. An *atomic coercion* is a containment s$\subseteq$t between type variables, or atomic type names if we were to extend the syntax of type expressions to include constants. This class of containments is practically interesting, since many common coercions like *int*$\subseteq$*real* are atomic. It also seems that, in light of Wand's treatment of labeled record types [Wand 87], the kind of subtyping that arises in "object-oriented" languages with class hierarchies may be characterized by subtyping axioms about atomic type names. The reader may wish to consult [Jategaonkar and Mitchell 88], which is based on the present paper.

By concentrating on atomic types, we will eliminate coercions like

$$(r \rightarrow s) \subseteq t,$$

which allow terms without Curry typings to be given types. In fact, with atomic coercions, we will see that every pure term that is typable with coercions is also typable without. This means that when we extend Algorithm GA to ML **let** declarations, we will have an algorithm for typing with subtypes that rejects precisely the same pure terms (terms without constants) as the ML type checker. Of course, this does not mean that atomic coercions have no effect; most typable terms will have more typings when coercions are considered. For example, the application fx of f to x has typing

$$\{\mathit{int} \subseteq \mathit{real}\}, \{f{:}\mathit{real} \rightarrow \mathit{real}, x{:}\mathit{int}\} \supset fx{:}\mathit{real},$$

while the application of a real function to an integer argument would not be typable without coercions.

Another interest in atomic coercions stems from the normalization theorem for typing derivations given in Section 6.2. This theorem (Lemma 20) shows that whenever a typing statement $C,A \supset M{:}\sigma$ with only atomic coercions is provable, there is a typing derivation in which (coerce) is only applied to variables. This means that coercions only enter into the base case of the typing algorithm, and so we may optimize the remaining cases. In addition, the restriction to atomic coercions allows various optimizations in the representation of coercion sets, and related algorithms (which we will not go into in much detail). Before analyzing the structure of typing derivations with atomic coercions, we will discuss some useful properties of entailment with atomic coercions.

6.2. Atomic coercions and "matching"

An *atomic coercion set* C is a set of coercions $s \subseteq t$ between type variables. Although the phrase is slightly inaccurate, we will call a typing statement $C,A \supset M{:}\sigma$ with C atomic an *atomic typing statement*.

When coercion sets only contain atomic coercions, it is easy to see that C $\vdash \sigma \subseteq \tau$ only if these two expressions have essentially the same "shape," or pattern of type constructors (in our case, $\rightarrow$'s). To be more precise, we define the *matching* relation on type expressions by

(i) if σ is a type variable, then σ matches τ iff τ is a type variable

(ii) if $\sigma = \sigma_1 \rightarrow \sigma_2$, then σ matches τ iff $\tau = \tau_1 \rightarrow \tau_2$ and σ_i matches τ_i (i=1, 2).

It is easy to verify that matching is an equivalence relation on types. In addition we have

LEMMA 16. *If C $\vdash \sigma \subseteq \tau$, where C is an atomic coercion set, then σ matches τ.*

This is easily proved by induction on the derivation of $\sigma \subseteq \tau$ from C.

The following lemma about the structure of proofs from atomic coercion sets will be useful for analyzing derivations of typing statements.

LEMMA 17. *Let σ and τ be type expressions with $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$. Then C $\vdash \sigma \subseteq \tau$ iff C $\vdash \tau_1 \subseteq \sigma_1$ and C $\vdash \sigma_2 \subseteq \tau_2$.*

PROOF . One direction is a direct consequence of rule (arrow): if C $\vdash \tau_1 \subseteq \sigma_1$ and C $\vdash \sigma_2 \subseteq \tau_2$, then C $\vdash \sigma \subseteq \tau$. It remains to prove the converse.

We show that if C $\vdash \sigma \subseteq \tau$ for any σ and τ of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$, then there is a proof of $\sigma \subseteq \tau$ from C that ends with an application of rule (arrow). We argue by induction on the length of the proof of $\sigma \subseteq \tau$ from C. If the proof is one step, then this is either an application of rule (arrow), in which case the lemma obviously holds, or an instance of (ref). If σ and τ are identical, then we can also prove both $\sigma_1 \subseteq \tau_1$ and $\sigma_2 \subseteq \tau_2$ by (ref). This lets us prove $\sigma \subseteq \tau$ by (arrow).

For the inductive step, assume that we have a proof whose final step is a use of rule (trans) from antecedents $\sigma \subseteq \rho$ and $\rho \subseteq \tau$. By Lemma 16, we know that ρ has the form $\rho = \rho_1 \rightarrow \rho_2$. Since the proofs of $\sigma \subseteq \rho$ and $\rho \subseteq \tau$ are shorter, we may assume we have proofs of these inclusions ending in applications of rule (arrow). Thus

$$\text{C} \vdash \rho_1 \subseteq \sigma_1,\ \ \sigma_2 \subseteq \rho_2,\ \ \tau_1 \subseteq \rho_1,\ \ \rho_2 \subseteq \tau_2.$$

By rule (trans), we have C $\vdash \tau_1 \subseteq \sigma_1$ and C $\vdash \sigma_2 \subseteq \tau_2$, which proves the lemma.

Given any coercion $\sigma \subseteq \tau$ between matching type expressions σ and τ, there is a minimal atomic coercion set that implies $\sigma \subseteq \tau$.

LEMMA 18. *Let σ and τ be matching type expressions. There is an atomic coercion set C = ATOMIC($\sigma \subseteq \tau$) with $C \vdash \sigma \subseteq \tau$ and such that if C' is any atomic coercion set with $C' \vdash \sigma \subseteq \tau$, then $C' \vdash C$. Furthermore, C may be calculated from a graph representation of σ and τ in linear time.*

PROOF . It follows from Lemma 17 that we can compute ATOMIC($\sigma \subseteq \tau$) recursively by

$$\text{ATOMIC}(s \subseteq t) = \{\ s \subseteq t\ \}$$

$$\text{ATOMIC}(\sigma_1 \rightarrow \sigma_2 \subseteq \tau_1 \rightarrow \tau_2) = \text{ATOMIC}(\tau_1 \subseteq \sigma_1) \cup \text{ATOMIC}(\sigma_2 \subseteq \tau_2)$$

This can be implemented efficiently using a graph representation of type expressions simply by marking "positive" and "negative" occurrences of variables in σ and τ. Since comparing corresponding positive and negative positions is entirely straightforward, this can be programmed to run in linear time. (The notion of positive and negative occurrences is commonly used in logic. Putting an expression on the right of an $\rightarrow$ preserves sign, while putting an expression on the left reverses the signs of all subexpressions.)

Using much the same idea, it is also easy to decide whether an atomic coercion set C implies $\sigma \subseteq \tau$.

LEMMA 19. *The predicate $C \vdash \tau \subseteq \sigma$ is decidable in linear time, given a subroutine for the transitive closure of C.*

The proof is straightforward using Lemma 17. Since entailment from atomic coercion sets is easily reduced to transitive closure, a reasonable representation for atomic coercion sets might be directed graphs or adjacency matrices. This would allow transitive closure, and hence entailment, to be computed by standard means [Aho, Hopcroft and Ullman].

6.3. A normalization theorem for typing derivations

In general, a typing derivation may apply rule (coerce) to a term of any form. For this reason, Algorithm G for unrestricted **CC** typing, includes coercions in every case. However, with atomic coercions, we can show that every provable typing can be transformed into a typing derivation in which coercions are only applied to variables. This will be used to simplify Algorithm GA.

LEMMA 20. *For every provable atomic typing statement, $C, A \supset M{:}\sigma$, there is a proof in which rule (coerce) is only used immediately after the typing axiom (var).*

The proof of this lemma appears below.

A simple example shows that Lemma 20 fails for unrestricted coercions. Consider any derivation of the typing

$$\{(s \rightarrow s) \subseteq t\},\ \varnothing \supset \lambda x.x : t.$$

Since we must use (abs) to give a type to $\lambda x.x$, the typing derivation must use a statement of the form

$$\{(s \rightarrow s) \subseteq t\},\ \varnothing \supset \lambda x.x : \sigma \rightarrow \tau,$$

possibly followed by (coerce). Since (coerce) is needed to remove the $\rightarrow$ from the type of $\lambda x.x$, we must use (coerce) after (abs). Thus the lemma fails.

Another class of counterexamples to Lemma 20 is illustrated by the typing statement

$$\{(s \rightarrow s) \subseteq (t \rightarrow t)\},\ y{:}s \supset \lambda x.y : t \rightarrow t,$$

where although the coercion set is not atomic, the only coercions assumed are between matching type expressions. This typing statement may be derived by first proving $\lambda x.y{:}s \rightarrow s$, and then using (coerce). We cannot apply (coerce) to the variable y earlier in the derivation since $s \subseteq t$ is not provable from the coercion hypothesis. However, for typing with coercions between matching type expressions, we can strengthen the coercion inference rules so that Lemma 20 holds. Specifically, it suffices to add the inference rule

$$\text{(arrow-inverse)} \quad \frac{\sigma \rightarrow \tau \subseteq \sigma_1 \rightarrow \tau_1}{\sigma_1 \subseteq \sigma,\ \tau \subseteq \tau_1}$$

Although this rule is not sound without further restrictions on our semantics, it does seem fairly plausible. In addition, the algorithmic aspects of typing with coercions between matching type expressions and rule (arrow-inverse) seem quite similar to typing with atomic coercions.

PROOF OF LEMMA 20. Note that (var) is the only axiom scheme and so every proof is essentially a tree with an instance of (var) at each leaf. We think of each node as labeled by both the statement proved at that node and the final rule used in that proof. Given a proof of a statement C,A $\supset$ M:σ, define the *degree* of the proof to be the number of pairs of internal tree nodes $<\alpha, \beta>$ such that there is a path from a leaf through α to β, node α is labeled with a rule different from (coerce), and node β is labeled with rule (coerce). Note that α and β do not need to be adjacent. Intuitively, the degree gives us a measure of how far the occurrences of (coerce) are from the leaves. We show by induction on the degree of a proof that every provable statement has a proof of degree zero.

We need a preliminary fact about proofs for the case in which a node labeled (coerce) follows a node labeled (abs). Suppose we are given a proof of C,A[x:σ] $\supset$ M:τ and that C $\vdash \rho \subseteq \sigma$. Then we can produce a proof of C,A[x:ρ] $\supset$ M:τ by replacing every leaf labeled with the statement C,A[x:σ] $\supset$ x:σ by a short proof of this statement beginning with C,A[x:ρ] $\supset$ x:ρ and then using (coerce). Note that the proof we produce has the same degree as the proof we start with.

It is now a simple matter to prove by induction on the degree of proofs that every provable statement has a proof of degree zero. The three possibilities to consider are that rule (coerce) may follow a use of rule (app), (abs) or another use of rule (coerce). If we have a node labeled (coerce) following another node labeled (coerce), then we can collapse these two proof steps into one using rule (trans) for inclusions. So it remains to consider (abs) and (app).

For the (app) case, suppose C,A $\supset$ MN:τ follows from C,A $\supset$ M: $\sigma \longrightarrow \tau$ and C,A $\supset$ N:σ by rule (app) and then C,A $\supset$ MN:ρ follows by (coerce). We have C $\vdash \tau \subseteq \rho$. Therefore, by rule (arrow),

$$C \vdash (\sigma \longrightarrow \tau) \subseteq (\sigma \longrightarrow \rho).$$

So we can derive C,A $\supset$ M:$\sigma \longrightarrow \rho$ from C,A $\supset$ M:$\sigma \longrightarrow \tau$ by rule (coerce) and then proceed to use rule (app) to derive C,A $\supset$ MN: ρ. This reduces the degree of the proof by one.

The final case is a node labeled (coerce) following a node labeled (abs). Suppose the proof has a path with nodes labeled

$$C,A[x{:}\sigma_1] \supset M{:}\sigma_2$$

$$C,A \supset \lambda x.M{:}\sigma_1 \rightarrow \sigma_2 \quad \text{(by rule abs)}$$

$$C,A \supset \lambda x.M{:}\rho_1 \rightarrow \rho_2 \quad \text{(by rule coerce)},$$

where we have used Lemma 16 to assume without loss of generality that the final type has the form $\rho_1 \rightarrow \rho_2$.

We would like to move rule (coerce) above (abs). Note that since

$$C \vdash \sigma_1 \rightarrow \sigma_2 \subseteq \rho_1 \rightarrow \rho_2,$$

we have $C \vdash \rho_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \rho_1$ by Lemma 17. By the preliminary fact noted above, there is a proof of C, $A[x{:}\rho_1] \supset M{:}\sigma_2$ with the same degree as the proof of $C,A[x{:}\sigma_1] \supset M{:}\sigma_2$. Now, applying rule coerce, we can prove $C,A[x{:}\rho_1] \supset M{:}\rho_2$ and so by rule (abs) we have $C,A \supset \lambda x.M{:}\rho_1 \rightarrow \rho_2$. This reduces the degree of the proof by one and finishes the proof of the lemma.

6.4. Substitutions, instances and most general atomic typings

We will apply substitutions to atomic coercion sets by computing the least atomic coercion set that implies the substitution instance. A substitution S *respects* coercion set C if, for every $\sigma \subseteq \tau$ in C, the substitution instances $S\sigma$ and $S\tau$ match. If S respects C, then we define the action of S on C by

$$S{\bullet}C = \cup_{\sigma \subseteq \tau \in C} \ \text{ATOMIC}(S\sigma \subseteq S\tau).$$

The instance relation on typings with atomic coercions is defined just as with unrestricted coercions, except that we interpret the application SC of substitution S to coercion set C using ATOMIC as above. More precisely, a typing statement $C',A' \supset M{:}\sigma'$ *is an atomic instance of* $C,A \supset M{:}\sigma$ if there exists a substitution S respecting C such that

$$C' \vdash S{\bullet}C, \quad A' \supseteq SA, \quad \text{and} \quad \sigma' = S\sigma.$$

Note that by Lemma 18, $S{\bullet}C \vdash SC$, and so by Lemma 1 any C' with $C' \vdash S{\bullet}C$ also satisfies $C' \vdash SC$. Therefore, it follows from Lemma 11 that every atomic instance of a provable atomic typing is also a provable atomic typing.

LEMMA 21. *Suppose $C',A' \supset M{:}\sigma'$ is an atomic instance of $C,A \supset M{:}\sigma$. If $\vdash C,A \supset M{:}\sigma$, then $\vdash C',A' \supset M{:}\sigma'$.*

An interesting corollary is that every pure lambda term with an atomic typing also has a Curry typing.

COROLLARY 22. *Suppose C is atomic and M is a pure lambda term without constants. If $\vdash C,A \supset M{:}\sigma$, then M has a Curry typing.*

This follows from the fact that if we instantiate $C,A \supset M{:}\sigma$ using a substitution that maps all type variables to a single variable t, then we have a typing for M with all coercions following from reflexivity of containment. Therefore, this typing may be proved without using rule (coerce). The details of are left to the reader. For the reader interested in type constants, it is worth pointing out that if M has a **CC** typing with type constants, the constants may be replaced by type variables to yield a provable typing without type constants. Then, by applying a substitution as above, we can produce a Curry typing.

6.5. Matching substitutions for atomic coercion sets

In the typing algorithm for atomic coercions, we will use an algorithm similar to unification to maintain the "atomicity" constraint. The need for this arises when we have typings $C_1,A_1 \supset M{:}\sigma$ and $C_2,A_2 \supset N{:}\tau$ for terms M and N, and wish to find a typing for MN. Using unification, we can find common substitution instances of the typing assumptions in A_1 and A_2 which could make MN well-typed (if, in fact MN is typable). However, the substitution giving us a typing of MN may not respect the coercion sets C_1 and C_2. Therefore, given a set C derived from C_1 and C_2, we will need to find the most general substitution S which, for every $\sigma \subseteq \tau \in C$, produces matching type expressions $S\sigma$ and $S\tau$. We will say that S *is a matching substitution for* C if, for every $\sigma \subseteq \tau \in C$, the substitution instances $S\sigma$ and $S\tau$ match.

LEMMA 23. *Let C be a set of containment expressions of the form $\sigma \subseteq \tau$, where σ and τ may not necessarily match. There is an algorithm MATCH such that whenever there is a matching substitution for C, then MATCH(C) produces a most general matching substitution. If C has no matching substitution, then MATCH(C) fails.*

Algorithm MATCH, which is similar to unification, is discussed in the Appendix.

6.6. Algorithm GA for most general atomic typings

Given any term M, the algorithm GA(M) below produces an atomic typing C,A $\supset$M:σ for M, or *fails*. The algorithm is written below in the same applicative, pattern-matching style as algorithm G, using similar notation.

GA(x) = { s$\sqsubseteq$t }, {x:s} $\supset$ x:t

GA(MN) =
let C_1, $A_1 \supset$ M:σ = GA(M)
C_2, $A_2 \supset$ N:τ = GA(N),
with type variables renamed to be disjoint from those in GA(M)
R = UNIFY({$\alpha{=}\beta$ | x:$\alpha{\in}A_1$ and x:$\beta{\in}A_2$} $\cup$ {$\sigma{=}\tau{\longrightarrow}$t})
where t is a fresh type variable
S = MATCH($RC_1{\cup}RC_2$) $\circ$ R
in
S•($C_1{\cup}C_2$), $SA_1{\cup}SA_2 \supset$ MN:St

GA(λx.M) =
let C, A $\supset$ M:τ = GA(M)
in if x:$\sigma \in$ A for some σ
then C, (A -- {x:σ}) $\supset$ λx.M:$\sigma{\longrightarrow}\tau$
else C, A $\supset$ λx.M:t$\longrightarrow\tau$,
where t is a new type variable

While Algorithm G always succeeds, Algorithm GA may *fail* in the application case if the call to UNIFY or MATCH *fails*. This is to be expected since, by Corollary 22, Algorithm GA must fail on every term that does not have a Curry type. In particular, GA(M) must fail if any subterm of M has no normal form. We can prove that if GA(M) succeeds, then it produces a provable typing for M.

THEOREM 24. *If GA(M) = C,A $\supset$ M:σ, then C,A $\supset$ M:σ is a provable atomic typing statement.*

It follows, by Lemma 21, that every atomic instance of GA(M) is a provable typing for M. Conversely, every provable atomic typing for M is an atomic instance of GA(M).

THEOREM 25. *Suppose* $\vdash C,A \supset M{:}\sigma$ *is a provable atomic typing. Then GA(M) succeeds and produces an atomic typing with* $C,A \supset M{:}\sigma$ *as an atomic instance.*

Both theorems are proved at the end of this section. One consequence of Theorem 25 is that if GA(M) succeeds, then we may compute a most general Curry typing from GA(M). In stating and discussing this corollary, it is useful to introduce some notation. If C is an atomic coercion set, let E_C be the set of equations

$$E_C = \{s{=}t \mid s \subseteq t \in C\},$$

and for t appearing in C, let $[t]_C$ be the set of all s with s=t in the reflexive, symmetric and transitive closure of E_C. In other words, we write $[t]_C$ for the equivalence class of t with respect to the least equivalence relation containing E_C.

COROLLARY 26. *Let* $GA(M) = C,A \supset M{:}\sigma$ *be a most general atomic typing for M. Let S be a substitution that is a choice function on equivalence classes* $[t]_C$ *of type variables appearing in C, so that whenever* $t_1, t_2 \in [t]_C$*, we have* $St_1 = St_2 \in [t]_C$*. Then* $\emptyset,SA \supset M{:}S\sigma$ *is a most general Curry typing for M.*

PROOF . It should be clear from the definition of S that $\emptyset,SA \supset M{:}S\sigma$ is an atomic instance of GA(M). By Theorem 25, every Curry typing is an atomic instance of GA(M), so it suffices to show that every Curry instance of GA(M) is an instance of $\emptyset,SA \supset M{:}S\sigma$.

If $\emptyset,A' \supset M{:}\sigma'$ is an instance of $GA(M) = C,A \supset M{:}\sigma$ by substitution T, then $\emptyset$ must prove Ts = Tt for every $s \subseteq t \in C$. Consequently, T must unify E_C. But since S is a most general unifier for E_C, as is easily verified, there is some substitution R with $T = R \circ S$. It follows that $\emptyset,A' \supset M{:}\sigma'$ is an instance of $\emptyset,SA \supset M{:}S\sigma$ by R, proving the corollary.

Further discussion of the relationship between coercion sets and unification is given in the appendix. The remainder of this section is devoted to proving Theorems 24 and 25.

PROOF OF THEOREM 24. It is easy to see that G(x) is always a well-typing, so we move on to application and abstraction.

Consider GA(MN). By the inductive assumption, both

$$GA(M) \; = \; C_1, A_1 \supset M{:}\sigma$$

$$GA(N) \; = \; C_2, A_2 \supset N{:}\tau$$

are provable. (As in the proof of Theorem 13, we assume that the type variables in GA(N) have been renamed.) Since S is defined from the unifier R by composition, S must unify $\{\alpha{=}\beta \mid x{:}\alpha{\in}A_1$ and $x{:}\beta{\in}A_2\}$, and $\sigma{=}\tau{\longrightarrow}t$. This implies that $SA_1{\cup}SA_2$ is a well-formed type assignment. Since S is a matching substitution for $C_1{\cup}C_2$, we know that $S{\bullet}(C_1{\cup}C_2)$ is a well-defined atomic coercion set. Therefore, arguing as in the proof of Theorem 13, Lemma 11, implies that the two typings

$$S{\bullet}(C_1{\cup}C_2)SA_1{\cup}SA_2 \supset M{:}S\sigma$$

$$S{\bullet}(C_1{\cup}C_2)SA_1{\cup}SA_2 \supset N{:}S\tau$$

are both provable and so

$$S{\bullet}(C_1{\cup}C_2)SA_1{\cup}SA_2 \supset MN{:}St$$

follows by rule (app). Therefore GA(MN) is a provable atomic typing statement.

The abstraction case is similar to the case considered in the proof of Theorem 13, except that no additional coercions are introduced. Since the details are may be checked quite easily, we leave this task to the reader. This proves the theorem.

PROOF OF THEOREM 25. As in the proof of Theorem 14, an easy induction shows that when GA(M) succeeds, it produces a typing which A assigns a type to x iff x occurs free in M. The main argument proceeds by induction on the structure of terms, and is essentially similar to the proof of Theorem 14 in the variable and lambda abstraction cases. For this reason, we will only check the application case.

Suppose $\vdash$ C, A $\supset$ MN:ν is a provable atomic typing statement. By Lemma 20, this must follow from provable atomic typings

$$C, A \supset M{:}\mu{\longrightarrow}\nu$$

$$C, A \supset N{:}\mu$$

by rules (app). By the inductive hypothesis, GA(M) and GA(N) are most general atomic typings for M and N. This means that there exist substitutions T_1 and T_2 such that

$$C \vdash T_1 \bullet C_1 \quad A \supseteq T_1 A_1 \quad T_1 \sigma = \mu \rightarrow \nu$$

$$C \vdash T_2 \bullet C_2 \quad A \supseteq T_2 A_2 \quad T_2 \tau = \mu,$$

where C_1, C_2, etc., are as in the application case of Algorithm GA. Because GA renames type variables, no type variables in $C_2, A_2 \supset N{:}\tau$ appear in $C_1, A_1 \supset M{:}\sigma$. This allows us to combine substitutions T_1 and T_2. Anticipating the need for a substitution that behaves properly on the fresh variable t introduced in the algorithm, we let T be any substitution such that

$$Ts = T_1 s \text{ if s appears in the typing of M,}$$

$$Ts = T_2 s \text{ if s appears in the typing of N,}$$

$$Tt = \nu$$

Without considering the effect of T on t, it is easy to see that

$$C \vdash TC_1 \quad A \supseteq TA_1 \quad T\sigma = \mu \rightarrow \nu$$

$$C \vdash TC_2 \quad A \supseteq TA_2 \quad T\tau = \mu$$

so that both instances are by the single substitution T. By Lemma 3, the assignment A must give types to all free variables of M and N and, as noted earlier, an assignment produced by G always contains exactly the variables that occur free. Therefore, T must unify $\{\alpha{=}\beta \mid x{:}\alpha \in A_1 \text{ and } x{:}\beta \in A_2\}$. In addition, since $T\sigma = \mu \rightarrow \nu = T\tau \rightarrow Tt$, the substitution T unifies $\sigma{=}\tau \rightarrow t$. Since R is a most general unifier for these equations, there is a substitution V with

$$T = V \circ R.$$

Since C is atomic, V must be a matching substitution for $RC_1 \cup RC_2$. But since MATCH computes most general matching substitutions, this implies that

$$V = W \circ \text{MATCH}(RC_1 \cup RC_2)$$

for some W. It follows that $C, A \supset MN{:}\nu$ is an instance of $S \bullet (C_1 \cup C_2),\ SA_1 \cup SA_2 \supset MN{:}St$ by W. This proves the theorem.

7. Variations and extensions of the typing algorithms

7.1. Inserting Conversion Functions

Algorithm G and GA calculate the set of subtyping assumptions needed to type a given term, but do not insert any type conversion functions. This is consistent with the view that whenever σ is a subtype of τ, every values of type σ is also a value of type τ. In practice, however, it may be useful to represent elements of σ is some way that takes advantage of the particular features that distinguish σ from τ. For example, even though the subrange of integers from 1 to 12 is most naturally regarded as a subset of the integers, it may be useful to save space by allocating fewer bytes to the representation of each element of the subtype. Then, when an element of the subtype is used as an element of the supertype, it may be desirable to convert from one representation to another. (Otherwise, it would be necessary to discriminate between representations at run-time.) This may be accomplished by making relatively minor changes in either typing algorithm, as sketched briefly below. To simplify the discussion, we will only consider Algorithm GA. The modifications to Algorithm G are similar and left to the interested reader.

We assume that whenever $\sigma \subseteq \tau$, we are given a conversion function $h_{\sigma,\ \tau}$ mapping values of type σ into type τ. Given conversion functions for each subtyping assertion in C, we may construct conversion functions for every $\sigma \subseteq \tau$ provable from C. Unfortunately, the only way to do this seems to depend on the way we prove $\sigma \subseteq \tau$ from C. This illustrates a general problem with user-supplied type conversion functions.

If $C \vdash \sigma \subseteq \tau$, then we define the untyped lambda term $h_{\sigma,\ \tau}$ by induction on the proof of $\sigma \subseteq \tau$ from C, as follows. We use the standard abbreviation $M \circ N$ for the term $\lambda x.M(Nx)$.

(i) If $\sigma \subseteq \tau$ follows from $\sigma \subseteq \rho$ and $\rho \subseteq \tau$, then $h_{\sigma,\ \tau} ::= h_{\rho,\ \tau} \circ h_{\sigma,\ \rho}$

(ii) If $(\sigma_1 \rightarrow \sigma_2) \subseteq (\tau_1 \rightarrow \tau_2)$ follows from
$\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$, then $h_{\sigma,\ \tau} ::= \lambda x.\ h_{\sigma_2,\ \tau_2} \circ x \circ h_{\tau_1,\ \sigma_1}$.

The conditions in [Reynolds 80] may viewed as a natural way of guaranteeing that the function $h_{\sigma,\tau}$ is determined by the coercions associated with the hypotheses C, and independent the proof used to construct $h_{\sigma,\tau}$.

Two modifications to Algorithm GA are needed. The first is in the variable case, where coercions are used. Instead of returning

$$GA(x) = \{s \subseteq t\}, \{x:s\} \supset x:t$$

the algorithm inserts a coercion

$$GA(x) = \{s \subseteq t\}, \{x:s\} \supset h_{s,\, t}(x):t$$

The second modification is in the use of substitutions in the application case.

While substitutions are only applied to types in Algorithm GA, we must now apply substitutions to terms as well. The reason is that when the type of a variable is changed, we must also change the conversion function associated with it. Therefore, we replace the last line of the application case of Algorithm GA by

$$S \bullet (C_1 \cup C_2),\ SA_1 \cup SA_2 \supset S(MN):St,$$

with S now applied to the term MN, and define the application of a type substitution to an untyped term with conversion functions as follows.

The effect of applying a type substitution S to an untyped lambda term M with conversion functions is to replace each conversion function $h_{\sigma,\tau}$ with $h_{S\sigma,S\tau}$. Recall that the definition of $h_{\sigma,\tau}$ depends on the proof of $\sigma \subseteq \tau$ from C. We will produce a conversion function $h_{S\sigma,S\tau}$ corresponding to a proof of $S\sigma \subseteq S\tau$ from S•C. Specifically, we will use the proof of $S\sigma \subseteq S\tau$ obtained by replacing each nonlogical axiom $a \subseteq b \in C$ in the proof of $\sigma \subseteq \tau$ with a proof of $Sa \subseteq Sb$ from S•C. (Lemma 18 and the definition of S•C guarantee that there is a proof of $Sa \subseteq Sb$ from S•C.) Now that we have fixed $h_{S\sigma,S\tau}$, it is easy to see from the inductive definition of $h_{\sigma,\tau}$ that $h_{S\sigma,S\tau}$ may be obtained by substituting an untyped lambda term $h_{Sa,Sb}$ for each basic conversion function $h_{a,b}$ in $h_{\sigma,\tau}$.

The inductive proof of Theorem 13 may be modified to show that if the modified version of GA(M) succeeds, it produces a typing C, A $\supset$ N:σ such that A $\supset$ N:σ is a Curry typing (i.e., provable from rules (var), (app) and (abs) only), provided we assume $h_{\sigma,\,\tau}:\sigma \rightarrow \tau$. Furthermore, if each $h_{\sigma,\tau}$ in N is replaced by the identity function, then N reduces to M. The proof of Theorem 14 may also be modified to show that the new algorithm is also guaranteed to find a typing whenever there is a method for inserting conversion functions.

7.2. ML Polymorphism

Algorithms G and GA can also be extended to lambda calculus with a polymorphic **let** construct as in ML [Gordon, et. al. 79, Milner 78]. For notational simplicity, we discuss only Algorithm G below. From a theoretical point of view, the simplest way to extend the algorithm is to consider **let** an abbreviation in lambda terms. If we define **let** by

let x=M **in** N ::= [M/x]N,

then it follows immediately that our typing algorithms may be used to find most general typings for terms with **let**. From a practical point of view, it is more useful to extend G to type terms with **let** directly. However, we can use this fact that G(**let** x=M **in** N) should be equivalent to G([M/x]N) to provide some intuition for the extension of G to **let**.

Since Algorithm G deduces a typing for each subterm independently, the algorithm will type every occurrence of M in [M/x]N by precisely the same process. If we wish to type an expression of the form

let x=M **in** N

without substituting M for x in N, then we may compute G(M) once and begin to type N as usual. When we see an x inside N, we then substitute the typing for M. More specifically, if G(M) = C,A $\supset$ M:σ is the typing for M, then we would like to use C,A $\supset$ x:σ as the typing for x inside N. However, since substitution [M/x]N involves some renaming of bound variables in N to avoid capture of free variables in M, there are some minor complications regarding variables that occur free in M. These details are easily resolved, as described in detail in [Kanellakis and Mitchell 89, Kanellakis, Mairson and Mitchell 9+, Mitchell 90]; see especially the appendix of [Kanellakis, Mairson and Mitchell 9+].

8. Conclusion and Future Directions

As with Curry typing without coercions, a relatively simple set of inference rules is sufficient to deduce all semantically valid typing statements. However, semantic completeness is achieved at the cost of making the set of types of a term undecidable. Two type inference algorithms for the decidable set of inference rules (without term equality) are presented, one using arbitrary subtyping assumptions, and the other restricted to subtyping assumptions between atomic types. These algorithms could be used to extend the programming language ML with simple forms of subtyping. We have also addressed the algorithminc problem of inserting calls to type conversion functions at compile time, but not the semantics of type conversion.

We have only considered one semantic interpretation for the type connective $\rightarrow$. Two additional possibilities are the quotient-set semantics [Hindley 83a] and the F-semantics [Hindley 83b]. It seems likely that the techniques of [Hindley 83a, Hindley 83b] will suffice to prove completeness theorems for typing with coercions for both of these semantics. Some discussion of the relationships between these semantics, and further references, are given in [Mitchell 88].

Acknowledgements: Thanks to Ravi Sethi for originally suggesting the study of type inference with coercions and to Lalita Jategaonkar for many helpful suggestions.

Appendix. Algorithms UNIFY and MATCH

8.1. Unification

Since matching is an extension of unification, we begin with a short review of a unification algorithm. Unification may be programmed in the same functional, pattern-matching style as the typing algorithms. In the clauses below, we assume that any non-empty set matches a pattern consisting of the union of two sets. Although the matching of a set to a pattern is nondeterministic, this does not affect the correctness of the algorithm. We also assume a form of pattern-matching for equations which takes commutativity into account. For example, we assume that an equation $\sigma_1 \rightarrow \sigma_2 = s$ matches the pattern $t = \tau$.

Given a set of equations E and a substitution S, the algorithm UNIFY(E,S) attempts to find the most general substitution T≥S unifying E. While we are generally interested in calling UNIFY with the identity substitution to begin with, the substitution parameter is useful on recursive calls.

UNIFY(∅, S) = S;

UNIFY(E∪{t=τ}, S) =
    if τ is the variable t **then** UNIFY(E, S)
    else if t does not occur in τ **then** UNIFY([τ/t]E, [τ/t]∘S) **else** *fail*

UNIFY(E∪{$\sigma_1 \rightarrow \sigma_2 = \tau_1 \rightarrow \tau_2$}, S) = UNIFY(E∪{$\sigma_1 = \tau_1$, $\sigma_2 = \tau_2$}, S}

Intuitively, if UNIFY(E, S) is computed using a recursive call UNIFY(E_1, S_1∘S), then S_1 is a partial solution to E, with E_1 the "simpler" problem remaining. To prove more rigorously that UNIFY is correct, we first show that the algorithm always terminates.

LEMMA 27. *For every finite set of equations E and substitution S, the algorithm UNIFY(E, S) terminates.*

PROOF . Termination is proved by associating a "degree" with each set E of equations, and showing that the degree decreases with each recursive call. Although several other definitions will do equally well, we will say that the *degree* of set E of equations is the pair of natural numbers degree(E) = <# of occurrences of $\longrightarrow$, # of equations in E>.

We order degrees lexicographically, so that <m,n> is less than <i,j> if either m<i or m=i and n<j. It is easy to check that each recursive call involves a set E of lower degree. Since there is no infinite decreasing sequence of pairs of natural numbers, it follows that UNIFY(E,S) always terminates.

LEMMA 28. *Let E be a finite set of equations and S be any substitution. If any $T \geq S$ unifies E, then UNIFY(E, S) computes a unifier R for E with $T \geq R \geq S$. Otherwise, UNIFY(E, S) fails.*

PROOF . We use induction on the degree of E, using the same degree function as in the proof of Lemma 27. If E is a set with degree <0, 0>, then E must be empty and so it is easy to verify that the lemma holds.

Now suppose that E has degree <m, n> with at least one of these numbers greater than 0. Since E cannot be empty, E must be of the form $E_1 \cup \{\sigma = \tau\}$ for some equation $\sigma = \tau$. If one of these, say σ, is a variable t, then the second clause of the algorithm applies. It is easy to see that any unifier must map t to some substitution instance of τ (since the equation $t = \tau$ must be satisfied), and must satisfy the remaining equations $[\tau/t]E$. The remaining case, with $E = E_1 \cup \{\sigma_1 \longrightarrow \sigma_2 = \tau_1 \longrightarrow \tau_2\}$, is straightforward.

8.2. Most general matching substitutions

Let C be a set of coercion expressions $\sigma \sqsubseteq \tau$, where σ and τ may not match. Then S is a *matching substitution for* C if, for every $\sigma \sqsubseteq \tau$ in C, the expressions $S\sigma$ and $S\tau$ match. A matching substitution S is a *most general matching substitution* if every other matching substitution R may be obtained as the composition of S with some substitution T. Matching substitutions are related to unification by the following lemma. A substitution is *variable-to-variable* if it is a function from variables to variables.

LEMMA 29. *Let C be a set of possibly non-matching containment expressions $\sigma \sqsubseteq \tau$ and let E be the set of equations*

$$E = \{\sigma = \tau \mid \sigma \sqsubseteq \tau \in C\}.$$

Then S is a matching substitution for C iff there is a variable-to-variable substitution T such that $T \circ S$ unifies E.

PROOF . Suppose T is variable-to-variable and T∘S unifies E. Since T is variable-to-variable, $S\sigma$ matches $T(S\sigma)$ and $S\tau$ matches $T(S\tau)$ for every $\sigma \subseteq \tau$ in C. But since $T(S\sigma)=T(S\tau)$, S must be a matching substitution for C.

To prove the converse, suppose S is a matching substitution for C, and let T be the substitution which maps all variables to some arbitrarily chosen variable t. Then, for every $\sigma=\tau$ in E, we know that $S\sigma$ and $S\tau$ differ only in the names of variables, and so $T(S\sigma)=T(S\tau)$. Thus T∘S unifies E.

We will use this lemma to design a matching algorithm. Essentially, algorithm MATCH will first unify a set of equations, and then extract a most general matching substitution from the most general unifier. While this may seem an unnecessarily indirect way of computing a most general matching substitution, it is actually quite efficient when we implement unification using the usual graph representation of terms [Paterson and Wegman 78, Aho, Sethi and Ullman 86].

It is worth mentioning that Lemma 29 fails when constants are added to type expressions. For example, C={s⊆a, s⊆b} has a matching substitution (namely, the identity substitution), but E={s=a, s=b} cannot be unified since a and b are different constants. However, by treating constants as variables, we may still use Lemma 29 to reduce matching with constants to unification with constants.

The reduction of matching to unification requires a few preliminary definitions and lemmas. If σ is any type expression, it is easy to construct a most general type expression τ matching σ simply by replacing each variable occurrence in σ by a distinct fresh variable. By a similar process, we can "factor" any substitution S into the composition of a substitution S_1 which produces a most general type matching St for each t we choose, and a substitution S_2 that replaces variables to make $(S_2 \circ S_1)t=St$.

Let **V** be a set of type variables. A substitution S *chooses variables freely on* **V** if (i) for each v∈**V**, no type variable appears twice in Sv

(ii) for distinct u, v∈**V**, no type variable in Su appears in Sv. Essentially, this means that if v_1, v_2, ... is an enumeration of **V**, then no type variable appears twice in the list Sv_1, Sv_2, ... The following lemma is easy to prove.

LEMMA 30. *Suppose substitution S chooses variables freely on* **V** *and that T chooses variables freely on the set of all type variables occurring in Sv, for*

$v \in \mathbf{V}$. Then $S \circ T$ chooses variables freely on $\mathbf{V}$.

The proof is straightforward and is omitted.

We will now show how to factor any substitution into the composition of one that chooses variables freely and one that replaces the freely-chosen variables to produce the original substitution. It will be useful to write $S =_{\mathbf{V}} T$ if substitutions S and T agree on all variables from $\mathbf{V}$.

LEMMA 31. *Let S be any substitution and let* $\mathbf{V}$ *be a set of type variables such that there are infinitely many type variables not in* $\mathbf{V}$. *There are substitutions S_1 and S_2, computable from a symbolic representation of S in linear time, such that S_1 chooses variables freely on* $\mathbf{V}$, *substitution S_2 is variable-to-variable, and $S =_{\mathbf{V}} S_2 \circ S_1$. Furthermore, if $S =_{\mathbf{V}} T_2 \circ T_1$ for some variable-to-variable substitution T_2, then there exists a variable-to-variable substitution R with $T_1 =_{\mathbf{V}} R \circ S_1$.*

PROOF . To define S_1 and S_2, let $v_1, v_2, \ldots$ be an enumeration $\mathbf{V}$ and let us partition the complement of $\mathbf{V}$ into disjoint infinite sets $\mathbf{V}_1, \mathbf{V}_2, \ldots$ This is a technical device for associating a different set of type variables with each element of $\mathbf{V}$. It will also be convenient to choose some enumeration of each $\mathbf{V}_i$, say $\mathbf{V}_i = \{v_{i,1}, v_{i,2}, \ldots\}$.

For each $v_i \in \mathbf{V}$, let $S_1 v_i$ be the type expression derived from Sv_i by replacing the j-th variable occurrence in Sv_i (reading the expression from left-to-right, say), with the j-th variable $v_{i,j}$ from $\mathbf{V}_i$. Let S_2 map $v_{i,j}$ back to the variable occurring in the j-th position in Sv_i. Since the $\mathbf{V}_i$'s are disjoint, and each variable occurrence in $S_1 v_i$ contains a different $v_{i,j}$, substitution S_1 chooses variables freely on $\mathbf{V}$. It should be clear from the definition that S_2 is variable-to-variable and $S =_{\mathbf{V}} S_2 \circ S_1$. Since S_1 and S_2 may be constructed using a single left-to-right scan of a symbolic representation of S, both may be computed in linear time.

For the second part of the lemma, suppose $S =_{\mathbf{V}} T_2 \circ T_1$, with T_2 variable-to-variable. Since S_2 and T_2 are both variable-to-variable substitutions, the three type expressions Sv, $S_1 v$ and $T_1 v$ must match, for each $v \in \mathbf{V}$. Therefore, since S_1 chooses variables freely on $\mathbf{V}$, there is a function (substitution) R mapping the variable occurring in the j-th position of $S_1 v$ to the variable occurring in the j-th position of $T_1 v$. Since $T_1 =_{\mathbf{V}} R \circ S_1$, this proves the lemma.

We may now combine Lemmas 31 and 29 to reduce matching to unification.

LEMMA 32. *There is an algorithm MATCH which, given a finite set of coercion expressions C, produces a most general matching substitution for C if any matching substitution exists, and fails otherwise.*

It will be clear from the proof that MATCH has the same complexity as unification, provided that unification produces a representation of the most general unifier S that allows us to read off St for each type variable t. This is actually a nontrivial assumption, since many implementations of unification will produce a composition $S_1 \circ S_2 \circ ... \circ S_k$ of several substitutions, and the number of operations involved in simplifying such a result may be quadratic in the length of the input. However, when terms are represented using graphs, the unifying substitution is generally represented as an equivalence relation. Algorithm MATCH may then be implemented efficiently as an algorithm for accessing the graph data structure.

PROOF . Let $E = \{\sigma = \tau \mid \sigma \subseteq \tau \in C\}$ be the set of equations determined by C and let **V** be the set of type variables occurring in C. Given C, algorithm MATCH first computes a most general unifier S for E. If there is no unifier, then MATCH *fails.* Otherwise, the algorithm computes a substitution S_1 choosing variables freely on **V** such that $S =_{\mathbf{V}} S_2 \circ S_1$ for some variable-to-variable substitution S_2, and returns S_1 as the result. By Lemma 29 and the properties of unification, we know that MATCH succeeds with S_1 iff there is a matching substitution. It remains to be shown that when MATCH succeeds, S_1 is in fact a most general matching substitution for C.

Let R be any matching substitution for C. By Lemma 29, there exists a variable-to-variable substitution T with $T \circ R$ unifying **E**. Therefore, since S is a most general unifier, there is a substitution U with

$$U \circ S = U \circ S_2 \circ S_1 = T \circ R.$$

By Lemma 31 we can "factor" $U \circ S_2$ into U_1 choosing variables freely, and a variable-to-variable substitution U_2. This gives us

$$U_2 \circ U_1 \circ S_1 = T \circ R.$$

But by Lemma 30, we know that $U_1 \circ S_1$ chooses variables freely on **V**. Therefore, by the second part of Lemma 31, we may conclude $V \circ U_1 \circ S_1 = R$ for some variable-to-variable substitution V. This shows that S_1 is a most general matching for C.

References

[Aho, Hopcroft and Ullman] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Aho, Sethi and Ullman 86] Aho, A.V. and Sethi, R. and Ullman, J.D., *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[Barendregt 84] Barendregt, H.P., *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984 (revised edition).

[Barendregt, Coppo and Dezani 83] Barendregt, H., Coppo, M. and Dezani-Ciancaglini, M., A Filter Lambda Model and the Completeness of Type Assignment. *J. Symbolic Logic* 48, 4; 1983, pages 931-940.

[Cardelli 88] Cardelli, L., A Semantics of Multiple Inheritance. *Information and Computation* 76,1988, pages 138-164. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Cardelli and Mitchell 89] Cardelli, L. and Mitchell, J.C., Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, 1989. To appear. Also available as DEC SRC Technical Report 48, August 1989, 60 pages.

[Coppo, Dezani and Venneri] Coppo, M. and Dezani-Ciancaglini, M. and Venneri, B., Principal type schemes and lambda calculus semantics. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535-560. Academic Press, 1980.

[Coppo 83] Coppo, M., On the semantics of polymorphism. *Acta Informatica* 20,1983, pages 159-170.

[Curry and Feys 58] Curry, H.B and Feys, R., *Combinatory Logic I*. North-Holland, 1958.

[Damas and Milner 82] Damas, L. and Milner, R., Principal Type Schemes for Functional Programs. In *9-th ACM Symposium on Principles of Programming Languages*, 1982, pages 207-212.

[Dwork, Kanellakis and Mitchell 84] Dwork, C., Kanellakis, P. and Mitchell, J.C., On the Sequential Nature of Unification. *J. of Logic Programming* 1,1984, pages 35-50.

[Fuh and Mishra 88] Fuh, Y.-C. and Mishra, P., Type Inference with Subtypes. In *ESOP-88*, March 1988.

[Gordon, et. al. 79] Gordon, M.J. and R. Milner and C.P. Wadsworth, *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78 Springer-Verlag, 1979.

[Hindley 69] Hindley, R., The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. AMS* 146,1969. pages 29-60.

[Hindley 83a] Hindley, R., The Completeness Theorem for Typing Lambda Terms. *Theor. Comp. Sci.* 22,1983. pages 1-17.

[Hindley 83b] Hindley, R., Curry's Type Rules Are Complete with Respect to the F-Semantics Too. *Theor. Comp. Sci.* 22,1983. pages 127-133.

[Jategaonkar and Mitchell 88] Jategaonkar, L. and Mitchell, J.C., ML with extended pattern matching and subtypes. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, July 1988, pages 198-212.

[Kanellakis and Mitchell 89] Kanellakis, P.C. and Mitchell, J.C., Polymorphic unification and ML typing. In *16-th ACM Symposium on Principles of Programming Languages*, 1989, pages 105-115.

[Kanellakis, Mairson and Mitchell 9+] Kanellakis, P.C. and Mairson, H.G. and Mitchell, J.C., Unification and ML type reconstruction. In *Computational Logic, essays in honor of {Alan Robinson}*, pages . MIT Press, 199+. to appear

[Lambek and Scott 86] Lambek, J. and Scott, P.J., *Introduction to Higher-Order Categorical Logic*. Cambridge studies in advanced mathematics 7, 1986.

[Leivant 83a] Leivant, D., Polymorphic Type Inference. In *Proc. 10-th ACM Symp. on Principles of Programming Languages*, 1983, pages 88-98.

[Mac Lane 71] Mac Lane, S., *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5 Springer-Verlag, 1971.

[MacQueen and Sethi 82] MacQueen, D. and Sethi, R., A Semantic Model of Types for Applicative Languages. In *ACM Symp. on Lisp and Functional Programming*, 1982, pages 243-252..

[MacQueen, Plotkin and Sethi 86] MacQueen, D., Plotkin, G and Sethi, R., An ideal model for recursive polymorphic types. *Information and Control* 71, 1/2; 1986, pages 95-130.

[Meyer 82] Meyer, A.R., What Is A Model of the Lambda Calculus ?. *Information and Control* 52, 1; 1982, pages 87-122.

[Milner 78] Milner, R., A Theory of Type Polymorphism in Programming. *JCSS* 17,1978. pages 348-375.

[Mitchell 84a] Mitchell, J.C., Coercion and Type Inference (Summary). In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, January 1984, pages 175-185.

[Mitchell 88] Mitchell, J.C., Polymorphic type inference and containment. *Information and Computation* 76, 2/3; 1988.

[Mitchell 90] Mitchell, J.C., Type systems for programming languages. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science, Volume B*, pages 365--458. North-Holland, 1990.

[Paterson and Wegman 78] Paterson, M.S. and Wegman, M.N., Linear Unification. *JCSS* 16,1978. pages 158-167.

[Remy 88] D. Remy, Typechecking records and variants in a natural extension of ML. In *16-th ACM Symposium on Principles of Programming Languages*, 1989, pages 60-76.

[Reynolds 80] Reynolds, J.C., *Using Category Theory to Design Implicit Conversions and Generic Operators*, in *Semantics-Directed Compiler Generation* , pages 211-2580.Springer-Verlag Lecture Notes in Computer Science, Vol. 94 1980.

[Robinson 65] Robinson, J.A., A Machine Oriented Logic Based on the Resolution Principle. *JACM* 12, 1; 1965. pages 23-41.

[Scott 76] Scott, D., Data Types as Lattices. *Siam J. Computing* 5, 3; 1976, pages 522-587.

[Scott 80] Scott, D.S., Relating theories of the lambda calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403-450. Academic Press, 1980.

[Smyth and Plotkin 82] Smyth, M. and Plotkin, G.D., The category-theoretic solution of recursive domain equations. *SIAM J. Computing* 11,1982, pages 761-783.

[Stroustrop 86] Stroustrop, B., *The \$C^{++}\$ Programming Language*. Addison-Wesley, 1986.

[Wand 87] Wand, M., Complete Type Inference for Simple Objects. In *Proc. 2-nd IEEE Symp. on Logic in Computer Science*, 1987, pages 37-44. Corrigendum in {it Proc. 3-rd IEEE Symp. on Logic in Computer Science,} page 132, 1988.

[Wand and O'Keefe 89] Wand, M. and O'Keefe, P., On the complexity of type inference with coercion. In *Proc. ACM Conf. Functional Programming and Computer Architecture*, 1989, pages 293-298.