Coercion and Type Inference (Summary)

John C. Mitchell[1]
Bell Laboratories
Murray Hill, NJ 07974

10 November 1983

Abstract

A simple semantic model of automatic coercion is proposed. This model is used to explain four rules for inferring polymorphic types and providing automatic coercions between types. With the addition of a fifth rule, the rules become semantically complete but the set of types associated with an expression may be undecidable. An efficient type checking algorithm based on the first four rules is presented. The algorithm is guaranteed to find a type whenever a type can be deduced using the four inference rules. The type checking algorithm may be modified so that calls to type conversion functions are inserted at compile time.

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. This may make it easier to write or modify experimental programs quickly. More importantly, a single untyped program represents many explicitly typed programs. This gives rise to an implicit form of polymorphism. When type information is omitted from programs, there is a type inference problem. If $\mathcal{P}$ is a typed programming language, then *type inference for* $\mathcal{P}$ is the problem of taking any untyped program S and finding all possible ways of inserting type declarations into S that result in legal typed programs of $\mathcal{P}$. Type inference was first considered in [3, 7] and has been developed further in [6, 10, 13].

The main advantage of languages based on type inference is that they provide a form of polymorphism. Since a single untyped function, for example, may have many legal typings, it may be called with actual parameters of many different types. Ideally, type inference allows a programmer to think of the "type" of a function as determined by the meaning of the function. For example, if a function f makes sense whenever its actual parameter is a pair, then an ideal type inference procedure would deduce a type for f that allowed f to be applied to any type of pair. In addition to providing this kind of flexibility, which is often associated with "typeless" languages, type inference detects type errors prior to run time. This facilitates debugging and alleviates the need for costly run-time tests. Unfortunately, ideal type inference cannot be attained using efficient algorithms; it is impossible for any algorithm to deduce all types that correspond to the meaning of any given function f. This point will be explained more fully in Section 4. In practice, we are forced either to use some run-time checking or to forbid some expressions that could pass run-time type checks. The algorithm presented in this paper will forbid some semantically typed expressions, rather than generate run-time tests.

Automatic coercions, such as coercions from integers to real numbers, are common to many programming languages.

---

[1] Author's present address: MIT Lab for Computer Science, Cambridge MA 02139.

However, automatic coercions involve relationships between distinct types and it is not clear, a priori, how coercions effect type inference. One complicating feature of coercions is that a single coercion implies many others: if integers are coercible to reals, then real predicates (boolean-valued functions of real arguments) are coercible to integer predicates. This paper addresses the problem of inferring polymorphic types in a way which supports automatic coercions between various types. Some of the basic properties of coercions are closely related to type containment in more complicated type disciplines. Due to space limitations, some proofs and technical details are deferred to the full paper.

In many cases, with or without coercions, a single untyped function will have infinitely many legal typings. For example, the function Apply(f,x) with body f(x) is legally typed whenever the first formal parameter f is declared to be a single-argument function and the second formal parameter x is declared to be in the domain of f. One way of representing an infinite set of types is by using type expressions with *type variables*. Intuitively, a type expression with variables (also called a *type scheme*) represents the set of types that can be produced by substituting other type expressions for type variables. For example, the type expression

$$((s \to t) \times s) \to t$$

represents the substitution instances

$$((\text{int} \to \text{bool}) \times \text{int}) \to \text{bool},$$

$$((\text{real} \to \text{int}) \times \text{real}) \to \text{int},$$

and many others.[2] In the case of Apply(f,x), the body f(x) is legally typed if f has some functional type $s \to t$, for some types s and t, and if x has type s (the same s). When f has type $s \to t$ and x has type s, the result of Apply(f,x) = f(x) will have type t. Thus we can infer that Apply has type

$$((s \to t) \times s) \to t.$$

Apply takes a pair of arguments, the first of some type $s \to t$ and the second of type s, and returns a result of type t. Since this is true for any s and t, the type expression $((s \to t) \times s) \to t$ describes an infinite set of types of Apply.

The most popular and best known programming language based on type inference is ML [6]. In the specific type system chosen for ML, each expression, function and procedure has a *most general type*, also called a *principal type scheme*. This means that for each clause (i.e. expression, function or procedure) e, there is a single type expression σ (generally with type variables) such that all legal typings of e are substitution instances of σ. Most general type expressions have the convenient property that the most general type of a clause involving clauses d and e can be determined from the most general types of d and e. The efficiency of the ML type checker seems to be a direct consequence of the fact that ML expressions have most general types. We will see that when coercions are added to a subset of ML, type schemes alone do not characterize all possible typings of each term. Nonetheless, there is a concise representation for the set of legal typings of any typable term. This representation uses both type schemes and coercions between types.

The simple model of coercion adopted in this paper is based on set containment: if σ is coercible to τ, then we think of the semantic values associated with type σ as contained in the set of values associated with type τ. This approach to coercion encompasses common coercions such as converting integers to reals in arithmetic expressions. The set of integers may be viewed as a subset of the set of real numbers. Except briefly in a later section on inserting calls to conversion functions, we ignore the fact that integers and reals are usually given different representations. We do not consider automatic conversions such as converting reals to integers that cannot be viewed as set containment.

Many of the properties of coercion which are discussed in this paper will apply to other type systems that involve some implicit or explicit notion of type containment. In particular, type containment may come up in type systems which include type quantifiers or other binding operations. For example, the type

$$\forall t.\, t \to t$$

with a universal quantifier is the type of functions which, given any type σ, will behave as functions from σ to σ (cf. [5, 16]). The

[2] The type operator $\to$ means "function space" and $\times$ means "product space." For example, int $\to$ bool is the type of functions from integers to boolean and int $\times$ bool is the type of pairs $\langle a,b \rangle$ where a is an integer and b is a boolean.

type $\forall t.t \to t$ is essentially "contained within" the type $\forall t.(t \to t) \to (t \to t)$. This statement may be made precise in several ways (cf. [14]). Thus we expect that any untyped function which can be given the type $\forall t.\, t \to t$ can also be given the type $\forall t.(t \to t) \to (t \to t)$. In quantified type disciplines (with or without automatic coercion), containments of types is a crucial issue. Automatic coercion motivates a general study of type containment and this study provides insight into type inference for more complicated type systems.

2. Lambda Calculus and its Semantics

Lambda calculus, the basis of ML, is used to demonstrate type inference with coercions. The terms of untyped lambda calculus are defined by the grammar

$$e ::= x \mid e_1\, e_2 \mid \lambda x.\, e.$$

A *lambda model* (D, Fun, Graph) is a set D together with mappings Fun: $D \to [D \to D]$ and Graph: $[D \to D] \to D$ such that Fun $\circ$ Graph is the identity function of the range of Fun and such that all functions on D that are definable by lambda expressions can be "compiled" into elements of D using Graph. See [1, 12] for more information. One useful lambda model is the *term model*. The term model is built from equivalence classes of terms, with application defined by

$$\mathrm{Fun}([d])([e]) = [de],$$

i.e. the result of applying the equivalence class of d to the equivalence class of e is the equivalence class of the application de. See [1, 12] for details.

Given a lambda model (D, Fun, Graph) and environment ρ mapping variables to elements of D, the meaning of a lambda term e is defined inductively by

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

$$[\![e_1\, e_2]\!]\rho = \mathrm{Fun}([\![e_1]\!])([\![e_2]\!])$$

$$[\![\lambda x.e]\!]\rho = \mathrm{Graph}(\text{"the function whose value at } d \in D \text{ is } [\![e]\!]\rho(d/x)\text{"})$$

Again, the reader is referred to [1, 12] for specifics.

A few facts about the reduction rules of lambda calculus are assumed. See [1] for an excellent presentation. The reduction rules are

(α) $\lambda x.e \to \lambda y.(e[y/x])$ if y is not free in e,

(β) $(\lambda x.e)f \to e[f/x]$,

(η) $\lambda x.ex \to e$ if x is not free in e.

If a term e is of the form of the left-hand side of rule (α), (β) or (η), then e is an α-, β- or η-*redex*. We say that e *β-reduces to f in one step* if there is a subterm d of e which is an α-redex or β-redex and f is the result of contracting this redex in e. The term e *β-reduces* to f if there is a sequence of β-reductions leading from e to f. If we allow η-reduction in addition to α- and β-reduction, then we say *β,η-reduces* to f. A term which cannot be reduced is in *normal form*.

3. Type Expressions, Coercion Sets and Type Assignments

Although product types, lists, and other kinds of types are useful in programming languages, these types seem to interact with automatic coercion in a relatively straightforward way. The most interesting types for the our purposes are the functional types. Intuitively, the functional type $\sigma \to \tau$ consists of the set of functions which take arguments of type σ to results of type τ.

We use $\sigma \subseteq \tau$ to denote the fact (or assumption) that values of type σ can be coerced to values of type τ. If we think of the coercion relation between types as an ordering, then $\to$ is monotonic in its second argument:

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

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

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

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. While in some semantics, e.g. [19], the connective $\to$ is monotonic in both arguments, the antimonotonicity seems natural when we think of containment as the ability to coerce. If f is a function of one real argument, and integers are coercible to reals, then f should be applicable to integer values. It seems inappropriate to apply integer functions to real numbers.

Type expressions are built from type variables and constants using the connective $\to$. We adopt the notational conventions that

a, b, ... denote type constants,

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

ρ, σ, τ, ... denote type expressions.

An *atomic type* is either a type constant or a type variable. Greek letters α, β, ... from the beginning of the alphabet may be used to denote atomic types. The set of type expressions is defined by the grammar

$$\tau ::= a \mid t \mid \sigma \to \tau.$$

Types will be interpreted as arbitrary sets of elements of lambda models (cf. [2, 8]. A *type environment* η for a model (D, Fun, Graph) is a mapping from atomic types to subsets of D. The meaning of a type expression σ in a type environment η is defined inductively by

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

$$[\![\sigma \to \tau]\!]\eta = \{d \mid \forall\, d_1 \in [\![\sigma]\!]\eta,$$

$$\mathrm{Fun}(d)(d_1) \in [\![\tau]\!]\eta\}.$$

Note that membership in $\sigma \to \tau$ is determined only by the behavior of an element d as a function (i.e. the extension of Fun(d)). Other semantics for $\sigma \to \tau$ are proposed in [9, 14, 19].

A *coercion set* C is a set of coercions $\alpha \subseteq \beta$, where α and β are atomic types. A model (D, Fun, Graph) and type environment η satisfy a coercion set C if

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

Coercions like

$$(a \to b) \subseteq c.$$

have complex implications. Some sets of coercions between structured types may only be satisfiable in special lambda models, and some coercions may drastically change the set of typable terms. For example, if we assume the two coercions

$$(a \to a) \subseteq a \quad \text{and} \quad a \subseteq (a \to a),$$

then *every* term of the untyped lambda calculus will semantically have type a. In contrast, only a subset of the set of terms with normal forms will have types when no coercions are allowed. We make sure that our coercions are not restrictive "domain equations" by only considering logical consequences of coercions between atomic types.

A *type assignment* A is function from a set of variables that may appear in lambda terms to type expressions. Often, type assignments will be functions with finite domains. A type assignment A can be written as a set of statements of the form $x{:}\sigma$. A model, type environment η and environment ρ satisfy type assignment A if

$$\rho(x) \in [\![\sigma]\!]\eta \text{ whenever } A(x) = \sigma.$$

If x is a variable, σ a type expression and A a type assignment, then $A[\sigma/x]$ is a type assignment with $(A[\sigma/x])(y) = A(y)$ for any variable y different from x, and $(A[\sigma/x])(x) = \sigma$.

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

$$C,A \mid e{:}\ \sigma$$

means that if types may be coerced according to C and variables have the types assigned by A, then the expression e has type σ. More formally, a model, type environment η and environment ρ satisfy a typing $e{:}\sigma$ if

$$[\![e]\!]\rho \in [\![\sigma]\!]\eta.$$

A statement $C,A \mid e{:}\ \sigma$ is satisfied in a model (D, Fun, Graph) if every environment and type environment for (D, Fun, Graph) which satisfy C and A also satisfy $e{:}\sigma$. A statement is *valid* if it is satisfied by every model.

4. Rules for Type Inference

There are five simple, semantically complete rules for deducing type statements. The completeness theorem shows that any valid statement $C,A \mid e{:}\sigma$ may be proved. This theorem demonstrates that the typing rules accurately describe the untyped semantics of lambda terms. It is possible to add rules for other applicative programming language constructs, cf. [6, 13]. The first three rules of the system express well-known facts about the functionality of lambda terms [3]. The next rule, (coerce), formalizes the property that if a term e has type σ, and the type σ is coercible to the type τ, then e also has type τ. These four rules are straightforward and define a typed language similar to the usual simple typed lambda calculus. The type checking algorithm TYPE in Section 5 will be based on these four rules.

If we want a semantically complete set of rules, then untyped terms that have the same meaning must be given the same types. One natural way to achieve this is to add a fifth rule (equal) based on equality of untyped terms. Since the equations that are valid in all lambda models are recursively enumerable, the antecedents of this rule are recursively enumerable. However, it is not decidable whether the rule is applicable in any given instance. When we adopt rule (equal), the set of types associated with an expression is, in general, undecidable. The merit of completeness is that it demonstrates that the typing rules give accurate information about the "functionality" or typeless operational behavior of programs. The drawback of semantically complete typing rules is that the types of terms becomes undecidable. The completeness theorem assures us that the first four rules capture the "functionality" of terms, modulo the problem of deciding equivalence of terms. Algorithm TYPE will be based on the decidable proof system without (equal) that is not semantically complete.

The rule (coerce) that models automatic coercions will use two subsidiary rules for deducing consequences of coercion sets. Although coercion sets only contain coercions between atomic types, many other coercions will follow as consequences. Two straightforward rules for deriving coercions are

(arrow) From $\sigma_1 \subseteq \sigma$ and $\tau \subseteq \tau_1$

derive $\sigma \to \tau \subseteq \sigma_1 \to \tau_1$

and

(trans) From $\sigma \subseteq \tau$ and $\tau \subseteq \rho$ derive $\sigma \subseteq \rho$.

The soundness of these rules follows easily from the definition of $[\![\sigma]\!]$. A coercion $\sigma \subseteq \rho$ is *provable from* C, written

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

if $\sigma \subseteq \rho$ can be derived from elements of C using rules (arrow) and (trans).

The following lemma about the structure of proofs of coercions will be used to prove several facts about derivations of typing statements.

Lemma 1: Let σ and τ be type expressions with $\sigma = \sigma_1 \to \sigma_2$ and $\tau = \tau_1 \to \tau_2$. Then $C \vdash \sigma \subseteq \tau$ iff $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$.

Now back to the problem of assigning types to lambda terms. Three well-known rules [3, 4, 8, 9] are

(var) C,A | x:σ whenever A(x) = σ,

(app) From C,A | e_1:$\sigma \to \tau$ and C,A | e_2:σ

derive C,A | e_1 e_2: τ,

(abs) From C, A[σ/x] | e:τ

derive C,A | λ x.e : $\sigma \to \tau$.

The coercion rule for typing lambda terms, based on the rules for deducing inclusions, is

(coerce) From C,A | e:σ and $C \vdash \sigma \subseteq \tau$

derive C,A $\vdash$ e:τ.

It is relatively easy to see that these rules are sound. An interesting property of the four rules is described by the following lemma. Essentially, this generalization of the Subject Reduction Theorem of [3] shows that types as defined by the four typing rules above are closed under β,η-reduction (n.b. not conversion). The lemma is interesting in itself, and will be also be used in the completeness proof and its corollaries.

Lemma 2: (Subject Reduction Lemma) If C,A | e:σ is provable using rules (var), (app), (abs) and (coerce), and e β,η-reduces to f, then there is also a proof of C,A | f:σ using (var), (app), (abs) and (coerce).

The four rules above are not semantically complete. This follows from the fact that every term which is typable using these rules looks just like a term of the simple typed lambda calculus, and every subterm of every term of that calculus has a normal form. However, the expression

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

has a subterm without a normal form but the term is semantically equal to the typable term λ y.y. Therefore, this term has a type semantically, but it cannot be typed according to the rules above.

We obtain a semantically complete set of rules if we add

(equal) From C,A | e :σ and e = f

derive C,A | f:σ.

This rule gives us a complete system and accounts for the undecidability of the consequences of the typing rules. There are

many possible interpretation for the symbol = in rule (equal). Equality can be interpreted as equivalent under β-conversion (the reflexive and transitive closure of β-reduction) or β,η-conversion. If we choose β-conversion, then we obtain a system that is complete for all lambda models. If we choose β,η-conversion instead, then we have a proof system which is complete for all models of η-reduction.[3] Note that the theory of β-conversion, as well as the theory of β,η-conversion, has the property

(*) if e β,η-reduces to f and $e = e_1$, then there exists e_2 such that e_1 β,η-reduces to e_2 and $e_2 = f$

(cf. [8]).

The following lemmas are required if = is not closed under η-reduction. Both are generalizations of lemmas found in [8].

Lemma 3: (Equality Postponement) If C,A | e:σ is provable, then there is a proof of C,A | e:σ in which all uses of rules other than (equal) appear before all uses of (equal). Equivalently, if C,A | e:σ is provable, then there is some f such that e = f and C,A | f:σ is provable without (equal).

Lemma 4: If C,A | e:σ is provable, possibly using (equal), and e β,η-reduces to f, then there is a proof of C,A | f:σ.

A corollary of the subject reduction lemma and the two lemmas above is that if = is β-conversion or β,η-conversion, and e is in the corresponding normal form, then C,A | e:σ is provable using all five rules iff it is provable without rule (equal); see Corollary 1.1.

Theorem 1: The five inference rules are sound and complete for deducing valid statements of the form C,A | e:σ.

The completeness theorem has two interesting corollaries. The first four rules are semantically complete for typing terms in normal form and the two containment rules are complete for deducing consequences of coercion sets.

Corollary 1.1: If e is in β-normal form and C,A | e:σ is valid in all lambda models, then C,A | e:σ may be proved using rules (var), (app), (abs) and (coerce). Similarly, if e is in β,η-normal form and C,A | e:σ is valid in all extensional lambda models, then these four rules are sufficient to prove C,A | e:σ.

Proof: Suppose C,A | e:σ is valid. Then C,A | e:σ is provable from the five rules, with = interpreted appropriately. By the equality postponement lemma, there is some f with f = e and C,A | f:σ provable using only rules (var), (app), (abs) and (coerce). By property (*), there is some term d which both e and f reduce to. But since e is in normal form, we conclude that f reduces to e. So by the subject reduction lemma, it follows that C,A | e:σ is provable without rule (equal).∎

Corollary 1.2: If $\sigma \subseteq \tau$ holds in every model and type environment satisfying a 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 1, it is used critically in the proof of the subject reduction lemma. Since Theorem 1 depends on rule (arrow) only in that (arrow) ensures that types are closed under η-reduction, the rule

(eta) From C,A | λx.ex:σ with x not free in e

derive C,A | e:σ

eliminates the need for rule (arrow). Note that rule (eta) is a sound, derived rule (by the subject reduction lemma), even for nonextensional models.

It is interesting to note that Corollary 1.2 also holds for the much more specialized "ideal" model of types [11] for lambda models which are complete partial orders (cf. [14]). In the ideal model, the only sets that are considered to be types are sets with specific order properties.

5. A Type Checking Algorithm

An important feature of the four typing rules (var), (app), (abs) and (coerce) is that whenever we can prove a typing statement C,A | e:σ, we know that all subterms of e are typable. An

[3] In fact, Theorem 1 holds whenever = is interpreted as equivalence in any lambda theory that extends the theory of β-conversion (cf. [1]), provided that property (*) holds. More precisely, let T be the equational theory of some lambda model and define e = f iff e = f ∈ T. If = satisfies property (*), then the rules (var), (app), (abs), (coerce) and (equal) are sound and complete for deducing typing statements that hold in all models of T.

intuitively appealing principal is that an expression should be typable only if all its subterms are typable. This principal, along with the fact that typing using (equal) is algorithmically intractable, suggests that we should base a type checking algorithm on the proof system without rule (equal). In the remaining sections of the paper, we consider typing only as defined by the first four typing rules, not semantic typing characterized by the system with (equal).

While a statement C,A | e:σ tells the type of e subject to some assumptions about coercions and types of variables, it does not tell us the types of subterms of e. As a consequence, it is difficult to tell by inspection whether a statement is provable from rules (var), (app), (abs) and (coerce). Furthermore, a provable statement C,A | e:σ does not give any information about which types might have been given to bound variables in the proof of the statement. We will be able to analyze the type checking algorithm TYPE more easily, and associate types with binding occurrences of bound variables, if we use more detailed formulas that include the types of all subterms.

An *explicitly typed term* $\bar{e}$ is a term e together with a mapping from subterms of e to type expressions. We will be informal about the mapping and write types as subscripts whenever we need to mention them explicitly, i.e. $\bar{e}_\sigma$ denotes an explicitly typed term $\bar{e}$ whose type is σ.

Normal well-typings are defined as follows.

$C,A \mid \bar{e}_\sigma$ is a *normal well-typing* if

(i) $\bar{e}$ is a x_σ for a variable x and either

(a) $A(x) = \sigma$ or

(b) for some τ, we have $A(x) = \tau$ and $C \vdash \tau \subseteq \sigma$

(ii) $\bar{e}$ is an application $\bar{d}_{\rho \to \tau}\ \bar{f}_\upsilon$ such that

(a) both $C,A \mid \bar{d}_{\rho \to \tau}$ and $C,A \mid \bar{f}_\upsilon$

are normal well-typings

(b) $\upsilon = \rho$

(iii) $\bar{e}$ is an abstraction $\lambda x_\rho.\bar{d}_\tau$,

the type σ is $\rho \to \tau$, and

$C, A[\rho/x] \mid \bar{d}_\tau$ is a normal well-typing

We define *well-typings* using an inductive definition as above, but substitute the clause

(ii b') $C \vdash \upsilon \subseteq \rho$

for $\upsilon = \rho$ in case (ii) above. Note that every normal well-typing is a well-typing. We will see from Lemma 5 that there is an efficient algorithm for checking whether $C \vdash \tau \subseteq \rho$. It follows that we can easily decide whether a candidate $C,A \mid \bar{e}_\sigma$ is in fact a well-typing.

Lemma 5: 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 1.

We have the following lemma relating well-typings to provable statements. The lemma may also be interpreted as a normal form lemma for proofs from (var), (app), (abs) and (coerce). Lemma 5 implies that it is only necessary to use rule (coerce) immediately following a use of rule (var).

Lemma 6: A statement C, A | e:σ is provable (without the equality rule (equal)) iff there is an explicit typing $\bar{e}_\sigma$ of e such that $C,A \mid \bar{e}_\sigma$ is a normal well-typing.

Given a term e, Algorithm TYPE will deduce a well-typing C, A | $\bar{e}_\sigma$ if any well-typing for e exists. The coercion set C will contain coercions between type variables. It is tempting, at first glance, to try to generalize the algorithms of [7] or [13] to find typings relative to some given, fixed set of coercions between type constants. For example, given a coercion set C and a closed term e, we might try to compute an explicitly typed term $\bar{e}_\sigma$ such that for every well-typing C, A | $\bar{e}_\tau$, the term $\bar{e}_\tau$ would be a substitution instance of $\bar{e}_\sigma$. Without coercion sets, this is possible [4, 7]. However, this cannot be done for all coercion sets. An example will be given in the final paper.

Substitution and Coercion Set Algorithms

Algorithm TYPE will use three subsidiary algorithms. One computes coercions sets and the other two produce substitutions. A *substitution* is a function from type variables to type

expressions. 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 by S(t). If $\bar{e}$ is an explicitly typed term, then $S\bar{e}$ is the result of applying S to every type in $\bar{e}$. A substitution S applied to a type assignment A is the assignment SA with (SA)(x) = S(A(x)). We define an equivalence relation on types before defining the effect of a substitution on a coercion set. After these definitions, we consider the substitution algorithms.

Intuitively, two type expressions *match* if they have the same form. More precisely,

(i) if σ is atomic, then σ matches τ iff τ is atomic

(ii) if $\sigma = \sigma_1 \to \sigma_2$, then σ matches τ iff

$\tau = \tau_1 \to \tau_2$ and σ_1 matches τ_1 and σ_2 matches τ_2.

It is easy to verify that matching is an equivalence relation on types. Furthermore, for any type σ, there is a type ρ, the *most general type that matches* σ, with the property that if σ matches τ, then τ is a substitution instance of ρ. The most general type that matches σ may be produced from σ just by renaming occurrences of type variables so that no type variable appears twice in the expression.

Recall that a coercion set C may only contain coercions between type constants and type variables, not type expressions. For example, the coercion set $\{t \to t \subseteq t\}$ is not well-formed. In order to use substitution as a mechanism for generating new well-typings, we have to define the action of a substitution S on a coercion set C. Given any coercion $\sigma \subseteq \tau$ between matching type expressions σ and τ, there is a unique minimal coercion set that implies $\sigma \subseteq \tau$.

> **Lemma 7:** Let σ and τ be matching type expressions. There is a coercion set C with $C \vdash \sigma \subseteq \tau$ and such that if $C_1 \vdash \sigma \subseteq \tau$, then $C_1 \vdash C$.

The lemma is proved by observing that a coercion set C implies

$$\sigma_1 \to \sigma_2 \subseteq \tau_1 \to \tau_2$$

only if C implies $\tau_1 \subseteq \sigma_1$ and C implies $\sigma_2 \subseteq \tau_2$. The minimal coercion set that implies $\sigma \subseteq \tau$ will be denoted COERCE(σ, τ). A linear time algorithm for COERCE(σ, τ) will appear in the appendix of the final paper. A substitution S *respects* coercion set C if, for every $\sigma \subseteq \tau$ in C, we have $S\sigma$ matches $S\tau$. If S respects C, then we define the action of S on C by

$$SC = \bigcup_{\sigma \subseteq \tau \in C} \text{COERCE}(\sigma, \tau).$$

Suppose A_1 and A_2 are type assignments with the same finite domain. Robinson's unification algorithm, UNIFY(A_1, A_2), can be used to find a most general substitution S that gives $SA_1 = SA_2$. If no such substitution exists, then UNIFY will *fail*.

> **Lemma 8:** There is an algorithm UNIFY(A_1, A_2) which computes a substitution S with $SA_1 = SA_2$ if any such substitution exists. Furthermore, if R is any substitution with $RA_1 = RA_2$, then there is a substitution T with $R = T \circ S$.

The algorithm UNIFY can be programmed to run in time $O(n\,\alpha(n))$, where n is the sum of the lengths of the type expressions and the function α is the inverse of Ackermann's function (cf. [15]).

We will also use a modified version of UNIFY to compute most general matchings. Two assignments A_1 and A_2 match if they have the same domain and, for every x in their domain, $A_1(x)$ matches $A_2(x)$. The algorithm TYPE uses an algorithm MATCH to compute matching substitutions that respect certain coercion sets. The algorithm MATCH(C, A_1, A_2) computes a most general substitution S that causes A_1 to match A_2 and that respects C. The algorithm, essentially a modification of unification (cf. [15, 18]), *fails* if there is no substitution S such that SA_1 matches SA_2 and S respects C. More formally,

> **Lemma 9:** Let C be a coercion set and A_1 and A_2 be type assignments. If S = MATCH(C, A_1, A_2), then S respects C and SA_1 matches SA_2. Furthermore, if R is a substitution that respects C such that RA_1 matches RA_2, then S = MATCH(C, A_1, A_2) succeeds and R = T∘S for some substitution T.

The algorithm MATCH and the proof of the lemma will appear in the final paper. Given any finite set of type variables $\mathfrak{V}$, we may assume that the range of S = MATCH(C, A_1, A_2) does not use any elements of $\mathfrak{V}$, i.e. we may assume that for all type variables t and u,

$t \in \mathfrak{V} \Rightarrow$ t does not appear in Su.

This assumption is used in the proof of Theorem 3.

If A is a type assignment, A \ x denotes the assignment which is identical to A, but not defined on x. If A_1 and A_2 are type

assignments, then $A_1 \setminus A_2$ is the assignment A_3 with $A_3(x) = \sigma$ iff $A_1(x) = \sigma$ and x is not in the domain of A_2.

Instances of Well-Typings

Well-typings are preserved by substitutions that respect coercion sets. Let C, A | $\bar{e}_\sigma$ be a well-typing. A well-typing C', A' | $\bar{e}_\tau$ an *instance* of C, A | $\bar{e}_\sigma$ if there exists a substitution S which respects C such that

$$C' \vdash SC, \quad A'|_{FV(e)} = SA|_{FV(e)}, \text{ and } \bar{e}_\tau = S\bar{e}_\sigma.$$

Here FV(e) denotes the set of variables that appear free in e. Note that SC, SA | $S\bar{e}_\sigma$ is an instance of C, A | $\bar{e}_\sigma$. We have

Lemma 10: Let C,A | $\bar{e}_\sigma$ be a well-typing. Every instance of C,A | $\bar{e}_\sigma$ is a well-typing.

The proof is by induction on the structure of e.

Algorithm TYPE

The algorithm is written below in an applicative style. Given an untyped expression e', the algorithm either returns a well-typing C, A | $\bar{e}$ or *fails*. The "overbars" on explicitly typed terms are omitted for typographical reasons.

```
TYPE(e') =
cases
  e' is a variable x:
     let s and t be new type variables
     return { s⊆t }, {x:s} | x_t
  e' is an application de:
     let C1, A1 | d_σ = TYPE(d)
     C2, A2 | e_τ = TYPE(e)
     A3 = A1 ∪ (A2 \ A1)
     A4 = A2 ∪ (A1 \ A2)
     R  =  MATCH(C1∪C2, A3[σ/z], A4[τ→t/z])
           where z and t are new variables
     S = UNIFY(RA3, RA4) ∘ R
     C = COERCE(Sτ, Left(Sσ))
     return
     C ∪ SC1 ∪ SC2, SA3 | Sd_σ e_τ
  e' is an abstraction λx.e
     let C, A, e_t = TYPE(e)
     return C, A \ x | λx_A(x).e_τ
end cases
```

The function *Left*(σ) used in the application case returns σ_1 if σ is of the form $\sigma_1 \to \sigma_2$ and is undefined otherwise. It is easy to see that the value of *Left* is defined in the call above. The algorithm may *fail* in the application case if either the call to MATCH or to UNIFY *fails*. If TYPE(e) does not *fail*, then it produces a well-typing.

Theorem 2: Let C, A | $\bar{e}'_\rho$ = TYPE(e'). Then every instance of C, A | $\bar{e}'_\rho$ is a well-typing.

Conversely, if there is a well-typing for e, then TYPE(e) will produce a well-typing.

Theorem 3: Suppose C', A' | $\bar{e}'_\rho$ is a well-typing. Then TYPE(e') = C, A | $\bar{e}'_\sigma$ succeeds. Furthermore, C', A' | $\bar{e}'_\rho$ is an instance of C, A, $\bar{e}'_\sigma$.

Theorem 2 is proved by induction on the structure of terms.

Proof of Theorem 3: The theorem is proved by induction on the structure of terms. Suppose e' is a variable x and C', A' | x_σ is a well-typing. The algorithm returns{ s⊆t }, {x:s} | x_t. Let τ = A'(x) and let T be the substitution [τ,σ/s,t]. Then certainly $A'|_{FV(e')} = (T\{x{:}s\})|_{FV(e')}$ and $Tx_t = x_\sigma$. Furthermore, since C', A' | x_σ is a well-typing, we must have C' $\vdash$ T{ s⊆t }. Thus C', A' | x_σ is an instance of TYPE(x).

Suppose e' is an application de, and C', A' | $\bar{d}_{\mu\to\nu}\bar{e}_k$ is a well-typing. By the inductive hypothesis,

C', A' | $\bar{d}_{\mu\to\nu}$ is an instance of C_1, A_1 | $\bar{d}_\sigma$ and
C', A' | $\bar{e}_k$ is an instance of C_2, A_2 | $\bar{e}_\tau$,

where C_1, C_2, etc. are as defined in the application case of Algorithm TYPE. Since A_1 assigns types to all free variables in d, we have $A_1|_{FV(d)} = A_3|_{FV(d)}$. Similarly, $A_2|_{FV(e)} = A_4|_{FV(e)}$. Therefore,

C', A' | $\bar{d}_{\mu\to\nu}$ is an instance of C_1, A_3 | $\bar{d}_\sigma$ and
C', A' | $\bar{e}_k$ is an instance of C_2, A_4 | $\bar{e}_\tau$.

We assume that UNIFY and MATCH always use new variables, so no type variables in C_2, A_4 | $\bar{e}_\tau$ appear in C_1, A_3 | $\bar{d}_\sigma$. Therefore, both instances above are by a single substitution T

that preserves $C_1 \cup C_2$. By the properties of MATCH, we know that $T = T_R \circ R$ for some T_R, where R is as in Algorithm TYPE. Furthermore, by the properties of UNIFY, we can see that there must be some substitution T_S such that $T = T_S \circ S$, where again S is as in Algorithm TYPE.

We have now shown that

$$C' \vdash T_S(SC_1 \cup SC_2),$$

$$A' |_{FV(e')} = T_S S A_1 = T_S S A_2.$$

Since C', A' | $\bar{e}'$ $_\rho$ is a well-typing we must have

$$C' \vdash T_R \mathrm{COERCE}(S\tau, \mathit{Left}(S\sigma)).$$

It follows that C', A' | $\bar{e}'$ $_\rho$ is an instance of C, A | $\bar{e}$ $_\sigma$.

Finally, suppose e' is an abstraction $\lambda x.e$ and C', A' | $\lambda x_\mu.\bar{e}$ $_\nu$ is a well-typing. Then, by definition of well-typing,

$$C', A'[\mu/x] \mid \bar{e}_\nu$$

is a well-typing. By the inductive hypothesis, this well-typing is an instance of

$$\mathrm{TYPE}(e) = C, A \mid \bar{e}_\tau.$$

This means that there must be a substitution S such that

$$C' \vdash SC, \quad A[\mu/x] |_{FV(e)} = SA |_{FV(e)} \text{ and } \bar{e}_\nu = S\bar{e}_\tau.$$

So $A' |_{FV(e')} = S(A \setminus x) |_{FV(e')}$ and $\lambda x_\mu.\bar{e}_\nu = S(\lambda x_{A(x)}.\bar{e}_\tau)$.

This concludes the proof of Theorem 3. ∎

A simple modification to Algorithm TYPE inserts calls to conversion functions. The modification does not affect the running time of the algorithm. A complete description is deferred to the final paper.

Conclusion

A relatively simple set of inference rules is sufficient to deduce all semantically valid types of any expression. However, semantic completeness is achieved at the cost of making the set of types of a term undecidable. When the inference rule that gives semantically equivalent expressions the same set of types is removed, the consequences of the inference system become decidable.

A type inference algorithm for the decidable set of inference rules is presented. Algorithm TYPE allows automatic coercions to be added to programming languages like ML. It may also be used to insert calls to conversion functions at compile time. One appealing property of algorithm TYPE is that a term has a type only if all subterms have types. If we think of the typable expressions as the expressions that are easy for human readers to understand, then algorithm TYPE will force a programmer to write code with the property that each subsection of a program is easy to understand. No untypable subsections will pass the type check.

The simple language of type schemes without binding operators does not allow some useful expressions to be typed. More complicated type disciplines, e.g. [5, 10, 14, 18], let more complicated expressions be given types, without sacrificing the principle that an expression is typable only if every subexpression is typable. Containment of types is a crucial issue in type inference for these type disciplines. It is hoped that the study of type containment presented here will be of use in developing more flexible type inference procedures.

References

1. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1981.

2. Barendregt, H., Coppo, M. and Dezani-Ciancaglini, M. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symbolic Logic* (198?). to appear.

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

4. Damas, L. and Milner, R. Principal Type Schemes for Functional Programs. 9-th ACM Symposium on Principles of Programming Languages, 1982, pp. 207-212.

5. Fortune, S., Leivant, D. and O'Donnel, M. The Expressiveness of Simple and Second Order Type Structures. *JACM 30*, 1 (1983). pp 151-185

6. Gordon, M.J., R. Milner and C.P. Wadsworth. *Lecture Notes in Computer Science. Vol. 78: Edinburgh LCF.* Springer-Verlag, 1979.

7. Hindley, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. AMS 146* (1969). pp 29-60.

8. Hindley, R. The Completeness Theorem for Typing Lambda Terms. *Theor. Comp. Sci. 22* (1983). pp 1-17.

9. Hindley, R. Curry's Type Rules Are Complete with Respect to the F-Semantics Too. *Theor. Comp. Sci. 22* (1983). pp 127-133.

10. Leivant, D. Polymorphic Type Inference. Proc. 10-th ACM Symp. on Principles of Programming Languages, 1983, pp. 88-98.

11. MacQueen, D. Private Communication. 1983.

12. Meyer, A.R. What Is A Model of the Lambda Calculus ?. *Information and Control 52*, 1 (1982). pp 87-122.

13. Milner, R. A Theory of Type Polymorphism in Programming. *JCSS 17* (1978). pp 348-375.

14. MacQueen, D. and Sethi, R. A Semantic Model of Types for Applicative Languages. ACM Symp. on LISP and Functional Programming, 1982, pp. 243-252..

15. Paterson, M.S. and Wegman, M.N. Linear Unification. *JCSS 16* (1978). pp 158-167.

16. Reynolds, J.C. Towards a Theory of Type Structure. Paris Colloq. on Programming, 1974, pp. 408-425.

17. Reynolds, J.C. *The Craft of Programming.* Prentice Hall, 1981.

18. Robinson, J.A. A Machine Oriented Logic Based on the Resolution Principle. *JACM 12*, 1 (1965). pp 23-41.

19. Scott, D. Data Types as Lattices. *Siam J. Comput. 5*, 3 (1976). pp 522-587.