Polymorphic Type Inference for Languages with Overloading and Subtyping

Geoffrey Seward Smith
Ph.D Thesis

91-1230
August 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

POLYMORPHIC TYPE INFERENCE FOR LANGUAGES WITH OVERLOADING AND SUBTYPING

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Geoffrey Seward Smith
August 1991

POLYMORPHIC TYPE INFERENCE

FOR LANGUAGES WITH OVERLOADING AND SUBTYPING

Geoffrey Seward Smith, Ph.D.

Cornell University 1991

Many computer programs have the property that they work correctly on a variety of types of input; such programs are called *polymorphic*. Polymorphic type systems support polymorphism by allowing programs to be given multiple types. In this way, programs are permitted greater flexibility of use, while still receiving the benefits of strong typing.

One especially successful polymorphic type system is the system of Hindley, Milner, and Damas, which is used in the programming language ML. This type system allows programs to be given universally quantified types as a means of expressing polymorphism. It has two especially nice properties. First, every well-typed program has a "best" type, called the *principal type*, that captures all the possible types of the program. Second, principal types can be *inferred*, allowing programs to be written without type declarations. However, two useful kinds of polymorphism cannot be expressed in this type system: overloading and subtyping. *Overloading* is the kind of polymorphism exhibited by a function like addition, whose types cannot be captured by a single universally quantified type formula. *Subtyping* is the property that one type is contained in another, as, for example, $int \subseteq real$.

This dissertation extends the Hindley/Milner/Damas type system to incorporate overloading and subtyping. The key device needed is *constrained universal quantification*, in which quantified variables are allowed only those instantiations that satisfy a set of constraints. We present an inference algorithm and prove that it is sound and complete; hence it infers principal types.

An issue that arises with constrained quantification is the *satisfiability* of constraint sets. We prove that it is undecidable whether a given constraint set is satisfiable; this difficulty leads us to impose restrictions on overloading.

An interesting feature of type inference with subtyping is the necessity of simplifying the inferred types—the unsimplified types are unintelligible. The simplification process involves *shape* unification, graph algorithms such as strongly connected components and transitive reduction, and simplifications based on the monotonicities of type formulas.

Biographical Sketch

Geoffrey Seward Smith was born on December 30, 1960 in Carmel, California. His childhood was spent in Bethesda, Maryland. In 1982, he graduated, *summa cum laude*, from Brandeis University with the degree of Bachelor of Arts in Mathematics and Computer Science. In 1986, he received a Master of Science degree in Computer Science from Cornell University. He was married to Maria Elena Gonzalez in 1982, and in 1988 he and his wife were blessed with a son, Daniel.

To Elena and Daniel

Acknowledgements

My advisor, David Gries, has been a patient, encouraging teacher. His high standards and devotion to the field have been an inspiration to me. His warmth and humanity have made it a pleasure to work with him.

Thanks also to my other committee members, Dexter Kozen and Stephen Chase. I have always marveled at Dexter's depth and breadth of knowledge; he has taught me a great deal throughout my years at Cornell.

Cornell has been a wonderful place to learn and grow; I am indebted to many people here who have taught and inspired me. Dennis Volpano discussed much of the material in this dissertation with me, and his comments and suggestions helped to shape this work. Bard Bloom gave me valuable insight into the shape unification problem. Discussions with Hal Perkins, Earlin Lutz, Steve Jackson, Naixao Zhang, and the other members of the Polya project have added greatly to my understanding of programming languages and type systems.

I have had a very nice group of officemates during my time at Cornell. For friendship, advice, and encouragement, I thank Amy Briggs, Jim Kadin, Daniela Rus, and Judith Underwood.

Finally, I thank Elena and Daniel for their love and support across the years.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the years, the notion of *types* in programming languages has grown steadily in importance and depth. In recent years, the study of types has been enriched by the introduction of ideas from mathematical logic. Yet, many basic questions about types remain unresolved—there is still no consensus about what exactly types *are*, or about what it means for a computer program to be type correct. What follows in this chapter, then, is an attempt to outline what is most important about types and type systems.

1.1 Types

It seems fundamental to human intelligence to *classify* the objects of the world based on their observed similarities and differences. This process of classification is a basic way of dealing with the complexity of the world: by grouping together similar objects, we may *abstract* away from particular objects and study properties of the group as a whole.

In the realm of mathematics, the process of grouping together similar objects has led to the identification of a rich diversity of mathematical structures, for example Euclidean spaces, topological spaces, and categories. The study of a mathematical structure leads to the development of a *theory* of the structure; the theory describes the common properties of the members of the structure and facilitates reasoning about the structure. Reasoning about a structure is further facilitated by the establishment of specialized *notation* tailored to the structure.

Many mathematical structures are also useful computational structures; for example graphs, sets, and sequences are especially useful in programming. In computer science, we call such structures *types*.

A *type*, then, is a set of elements and operations having an associated theory that allows reasoning about manipulations of elements of the type.

The programming process is much easier in a language with a rich collection of types; this is perhaps the most important feature of high-level languages. Hence a major consideration in the design of any such language is how types will be supported. Type support may be divided into two relatively independent aspects: first, to ensure that high-level types are correctly implemented in terms of the low-level data structures available on an actual machine, and second, to ensure that programs use types only in ways that are sensible. The first of these we call *reductionist* type correctness, the second we call *axiomatic* type correctness. We explore these in more detail in the next three sections.

1.2 Reductionist Type Correctness

Since actual machines have but a few types (for example, integers of various sizes, floating-point numbers, arrays, and pointers), a major task of a compiler is to implement the types of a high-level language in terms of low-level data structures. The implementation of types is complicated by the fact that many types (for example, sets) have no best implementation that is appropriate for all situations. Instead, different implementations should be used depending on the kind and frequency of the operations to be performed.

As a rule, programming languages have not attempted to use multiple implementations for built-in types[1]; instead, the usual approach is to provide a language facility that allows the programmer to define a new type and to give its representation in terms of other types. The key issue is how to protect the representation from misuse.

Suppose that we wish to implement a type *dictionary* with operations *insert* and *member?*. We might chose to represent the dictionary as a bit-vector. But this should not mean that type *dictionary* is the same as type *bit-vector*! For, outside the procedures *insert* and *member?*, there should be no access to the underlying representation. Furthermore, just because a value has type *bit-vector* we cannot conclude that the value represents a *dictionary.* The two concerns are called "secrecy" and "authentication" by Morris [Mor73], and they lead him to declare that "types are not sets". This appears to contradict our definition of *type* above, but this is really not the case. Morris is discussing what is needed for one type to represent another; he is not discussing the nature of types.

In general, it can be argued that a failure to distinguish between reductionist and axiomatic type correctness has led to a great deal of confusion about the nature of types.

Cardelli and Wegner follow in the reductionist style when they state,

> A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. [CW85, p. 474]

[1] SETL is an exception.

Similarly, in the language Russell [DD85], the notion of strong typing is defined so that the meaning of a well-typed program is independent of the representations of the types it uses. In a similar way, Mitchell and Plotkin [MP88] suggest that an abstract type that is represented by another type should be given an *existentially* quantified type as a way of hiding the representation.

Aho, Hopcroft, and Ullman [AHU83] assert that two types should be regarded as different if the set of operations they support is different, because the set of operations to be supported is crucial in determining the best implementation of a type. From the point of view of the programmer, however, this view of types seems inappropriate: it is a case of reductionist concerns contaminating the axiomatic view of types. A better way of understanding the situation is in terms of *partial implementations* of the *same* type [GP85,Pri87,GV89].

In the rest of this dissertation, we will not consider reductionist type correctness further. Instead we will think of types as existing axiomatically, and we will not worry about how they are represented. Henceforth, then, we will focus on *axiomatic* type correctness. Before we do so, we consider the notion of *polymorphism*.

1.3 Polymorphism

In mathematics, the study of a proof of a theorem may reveal that the theorem can be generalized. For example, in analyzing a proof of a theorem about the natural numbers, we might realize that the only property of the natural numbers needed is that they form a well-ordered set. This realization would enable us to generalize our theorem to any well-ordered set. Lakatos gives a fascinating account of the role of proof analysis in mathematical discovery in [Lak76].

To exploit this useful principle of generalization, we rewrite our theorems so that they refer to abstract mathematical structures (e.g. well-ordered sets, topological spaces) that have certain properties but whose identity is not fixed. Poincaré describes the process thus:

> ... mathematics is the art of giving the same name to different things ... When the language is well-chosen, we are astonished to learn that all the proofs made for a certain object apply immediately to many new objects; there is nothing to change, not even the words, since the names have become the same. [Poi13, p. 375]

Indeed, a great trend in twentieth-century mathematics has been the rise of *abstract* mathematical structures; rather than studying the properties of particular mathematical structures (for example Euclidean space), mathematicians today are more likely to study properties of abstract mathematical structures (for example metric spaces). Abstract mathematical structures are essentially proof generated; that is, they are defined by abstracting out those properties needed for certain proofs to be carried through.

In computer science, similarly, the study of a correctness proof of an algorithm may reveal that the algorithm works for many types of objects. Suppose for example that exponentiation is defined by

$$x^0 = 1 \text{ and } x^{n+1} = (x^n) * x.$$

Consider the following algorithm, which uses only $O(\log n)$ multiplications:

$$\begin{array}{l} expon(x,n) = \\ \quad \textbf{if } n = 0 \textbf{ then } 1 \\ \qquad\qquad \textbf{else if } even?(n) \textbf{ then } expon(x * x, n \div 2) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else } expon(x, n-1) * x \end{array}$$

In proving the correctness of *expon*, we require that n be a nonnegative integer, but we do not need to fix the type of x. All that is needed is for the type of x to have an associative operation $*$ with multiplicative identity 1. Hence, *expon* should be applicable to many types of inputs.

The principle of generalization goes by the name of *polymorphism* in computer science, except that it is common to encumber the concept of polymorphism with irrelevant implementation considerations: in particular, it is normally demanded that a polymorphic function execute the same machine code, regardless of the types of its inputs. [2]

Following Poincaré, we can say that the basis for polymorphism is the use of the same name for different objects, which allows programs that use these names to apply to many types of inputs. For example, we allow the identifiers *cons* and $*$ to refer to many different functions, depending on the argument types. More precisely, we say that *cons* and $*$ have many types and that they have a meaning corresponding to each of their types:

$$cons : \forall\alpha.\alpha \rightarrow seq(\alpha) \rightarrow seq(\alpha)$$

describes the types of *cons*, [3] and

$$* : int \rightarrow int \rightarrow int$$

$$* : real \rightarrow real \rightarrow real$$

$$* : complex \rightarrow complex \rightarrow complex$$

gives some of the types of $*$.

[2] The discussion of Cardelli and Wegner [CW85] is typical; Prins [Pri87], in contrast, argues against the necessity of uniform implementations for polymorphic functions.

[3] Throughout this dissertation, we use *currying* to avoid the messiness of functions with multiple arguments.

Notice that the types of *cons* are all described by a single universally quantified type formula; we therefore say that *cons* is *parametrically polymorphic.* The types of $*$, in contrast, cannot be uniformly described: they require many type formulas. For this reason, $*$ is said to be *overloaded.*

With respect to the generalization principle, which seems to be the true foundation of polymorphism, there is no essential difference between parametric polymorphism and overloading. Traditionally, however, overloading has been regarded as less genuine and significant than parametric polymorphism; indeed overloading is often belittled as *ad-hoc polymorphism.*[4] This disdain for overloading seems to stem mostly from the desire for uniform implementations—it is easy to give an implementation of *cons* that works for all kinds of sequences, but $*$ requires different machine code for integers than for floating-point numbers. In fact the usual way of treating the overloaded operators in a program is to demand that the local context determine a *particular* overloading to be used at each point; see, for example, [ASU86]. This is even the case in the polymorphic language ML [WB89]. But this treatment of overloading destroys the polymorphism of *expon* above: choosing a particular overloading of $*$ to be used by *expon* arbitrarily restricts *expon* to a single argument type.

Another common assertion is that overloaded identifiers have only finitely many types, in contrast to parametrically overloaded identifiers, which have infinitely many types. This need not be so, however. Consider the operation $\leq$. Two of its types are $real \rightarrow real \rightarrow bool$ and $char \rightarrow char \rightarrow bool$. In addition, $\leq$ may be used to compare *sequences* lexicographically (that is, by dictionary order). What sequences can be compared in this way? Precisely those sequences whose elements can be compared with $\leq$! Hence $\leq$ has the infinite collection of types $seq(char) \rightarrow seq(char) \rightarrow bool$, $seq(seq(char)) \rightarrow seq(seq(char)) \rightarrow bool$, and so forth. Such types can be characterized with the formula

$$\forall \alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool \,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool\,,$$

which represents all types of the form $seq(\tau) \rightarrow seq(\tau) \rightarrow bool$ that satisfy the constraint $\leq : \tau \rightarrow \tau \rightarrow bool$.

Another kind of polymorphism is due to *subtyping,* the property that one type is contained in another. For example, $int \subseteq real$. Subtyping leads to polymorphism in that a value in a type is also in any supertype of that type. So if $\leq$ works on type $real$, $\leq$ also works on type int.

From our axiomatic point of view, subtyping is a matter of set inclusion. Thus we say that *int* is a subtype of *real* because the set of integers is a subset of the set of reals. Another viewpoint is that subtyping is based on *coercion,* and that *int* is a subtype of *real* because an integer value can be coerced to a real value. This latter view is suggested by reductionist considerations (integers and

[4] Again, [CW85] is typical.

reals will probably be represented differently), but it seems better to separate such concerns from issues of axiomatic type correctness.

A striking example of a polymorphic function is Algorithm 5.5 in [AHU74]. The algorithm is defined in terms of an abstract type called a *closed semiring.* Depending on the choice of closed semiring, the algorithm computes the reflexive, transitive closure of a graph, the shortest paths between all pairs of vertices in a weighted graph, or a regular expression equivalent to a nondeterministic finite automaton.

Now that we have some understanding of polymorphism, we turn to the question of axiomatic type correctness.

1.4 Axiomatic Type Correctness

If multiplication is defined only on integers and reals, then it is meaningless to multiply a character by a boolean. A valuable service that a language can provide is to ensure that this sort of mistake cannot happen; this is what axiomatic type correctness enforces.

The basic approach is to associate a type with each *phrase* of a program and to give a system of rules specifying which combinations of phrases are sensible. These rules are called the *type system* of the language. As a simple example, we might have the following rule for $+$: "if e and e' are phrases of type *integer expression*, then $e + e'$ is a phrase of type *integer expression.*"

Type systems are simpler in *functional* languages, because they have only one kind of phrase, namely *expressions. Imperative* languages in contrast, have several kinds of phrases: *expressions*, *commands*, *variables*, and *acceptors.* We will limit ourselves to functional languages in this dissertation; Reynolds explores the typing of imperative languages in [Rey80,Rey81,Rey88] and polymorphism in imperative languages is investigated in [Tof88] and [LW91].

Because in a functional language every phrase is an expression, we will drop the use of *expression* from the types of phrases, saying that $e + e'$ has type *int* and writing $e + e' : int$. We should keep in mind, however, that there is a difference between saying that a program has type *int* and saying that a *value* has type *int*. A program has type *int* if it yields an integer result, but a value has type *int* if it is an element of the set of integers.

Built-in operations will appear as free identifiers within programs; for this reason the type of a program needs to be stated with respect to a *type assumption set* that gives the types of the built-in operations. Hence, the form of a typing judgement is

$$A \vdash e : \tau,$$

which may be read "from type assumption set A it follows that program e has type τ". The rules for proving such judgements make up the type system of the language.

For example, the type $\tau \to \tau'$ is the type of functions that accept an input of type τ and produce an output of type τ'. The rule for typing a function application $e'\,e''$ (i.e. the application of function e' to argument e'') is

$$\frac{A \vdash e' : \tau \to \tau' \quad A \vdash e'' : \tau}{A \vdash e'\,e'' : \tau'}$$

This rule is strikingly like the logical rule *modus ponens*:

$$\frac{\tau \to \tau' \quad \tau}{\tau'}$$

This is a manifestation of the remarkable *propositions as types* principle [GLT89], according to which types may be viewed as logical propositions and a program of a certain type may be viewed as an intuitionistic proof of the proposition corresponding to the type. As a consequence of this principle, it turns out that a type system is a system of *natural deduction* [vD83,Pra65].

Given a set of typing rules, a program is said to be *well typed* if from the rules it can be shown that the program has some type. If the compiler can verify that each program is well typed, the language is said to be *syntactically typed.* Traditionally, syntactically typed languages require programs to be written with type declarations; such declarations make it easy to check whether a program is well typed. In this case the problem is referred to as one of *type checking.* Other languages, such as ML, allow programs to be written without any type information; in this case checking type correctness requires more ingenuity and is referred to as *type inference.*

There are two reasons to prefer type inference to type checking. The obvious reason is that type inference saves the programmer keystrokes. The savings is especially great in the polymorphic type systems considered in this thesis—our type systems involve complex type formulas with constrained quantification, which makes the job of annotating programs with type information quite cumbersome.

A more subtle advantage of type inference is that forcing programmers to write type declarations may force them to overspecialize their programs. Consider, for example, the procedure *swap* in Pascal:

```
procedure swap(var x, y:integer);
var z:integer;
begin z:=x; x:=y; y:=z end
```

There is nothing about *swap* that depends on *integer*; indeed, changing the two declarations of *integer* to *character* produces a correct Pascal program. But there is no declaration a programmer can make that allows *swap* to be used on both *integer* and *character*. So the deeper advantage to type inference is that programs without type declarations implicitly have all types that can be derived for them. We will return to this point in the next section when we discuss *principal types*.

We have so far discussed type correctness solely with respect to a given set of typing rules, and we have said little about what it means for a program to be type correct. This is a tricky area; the absence of non-syntactic errors (for example, division by 0 or nontermination) cannot in general be established at compile time, so saying that a program has type *integer* does not guarantee that the program will succeed and yield an integer output. In practice, there are tradeoffs between the difficulty of type checking and the amount guaranteed by type correctness. A more extreme position is taken in the language Nuprl [C+86], where the language is restricted to less than universal computing power for the sake of stronger type guarantees.[5]

The type systems that we consider will be more modest. We will be satisfied, for example, to give $\div$ type $int \rightarrow int \rightarrow int$, ignoring the fact that $\div$ is a partial function. Furthermore, when considering polymorphism, we will allow a function to accept inputs of all types that 'look' correct, in the sense that the operations needed are defined on the input types. So in the example of *expon*, our type system will not require that $*$ be associative or that 1 be a multiplicative identity. Also, we will not demand that n be nonnegative.

1.5 The Hindley/Milner/Damas Type System

We now outline the polymorphic type system due to Hindley, Milner, and Damas [Hin69,Mil78,DM82,Car87]. This type system is successfully in use in the language ML.

The Hindley/Milner/Damas type system expresses parametric polymorphism by means of universally quantified types. For example, the primitive LISP operation *cons* may be given type

$$\forall\alpha.\alpha \rightarrow seq(\alpha) \rightarrow seq(\alpha)$$

to indicate that for every choice of α, *cons* has type

$$\alpha \rightarrow seq(\alpha) \rightarrow seq(\alpha).$$

A fundamental restriction on this type system is that all quantification must be *outermost*; that is, types must be in prenex form.[6]

[5] In particular, because Nuprl is *strongly normalizing*, a Nuprl function of type $int \rightarrow int$ is guaranteed to be a *total* function from int to int.

[6] In the literature, this is sometimes called the *shallow types* restriction.

Milner [Mil78] developed a type inference algorithm, called algorithm W, for this type system that allows programs to be written with no type information at all. In Chapter 2, we will study in detail an extension of algorithm W; for now we will be content to show, informally, how W infers a type for the expression $\lambda f.\lambda x.f(f\ x)$.[7]

The expression is of the form $\lambda f.e$ and we don't know what the type of f should be. So we choose a new type variable α, make the assumption $f : \alpha$, and attempt to find a type for $\lambda x.f(f\ x)$. Now the expression is of the form $\lambda x.e'$, so we choose a new type variable β, make the assumption $x : \beta$, and attempt to find a type for $f(f\ x)$. This is an application, so we first find types for f and $(f\ x)$. The type of f is easy; it is just α, by assumption. To find the type of $(f\ x)$, we begin by finding types for f and x; by assumption we have $f : \alpha$ and $x : \beta$. Now we must type the application using the rule for function application,

$$\frac{A \vdash e' : \tau \rightarrow \tau' \quad A \vdash e'' : \tau}{A \vdash e'\,e'' : \tau'}$$

Hence we discover that α must actually be of the form $\beta \rightarrow \gamma$ for some new type variable γ; that is, we refine our assumption about f to $f : \beta \rightarrow \gamma$. (This inference is made using *unification*.) This gives $(f\ x) : \gamma$. Now we must use the rule for function application again to find a type for $f(f\ x)$; we discover that β must be the same as γ, so we refine our assumptions to $f : \gamma \rightarrow \gamma$ and $x : \gamma$, yielding $f(f\ x) : \gamma$. Now we can "discharge"[8] the assumption $x : \gamma$ to get $\lambda x.f(f\ x) : \gamma \rightarrow \gamma$. Next we discharge the assumption $f : \gamma \rightarrow \gamma$ to get $\lambda f.\lambda x.f(f\ x) : (\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)$. Finally, since γ is not free in any active assumptions, we can quantify it, thereby obtaining the typing

$$\{\,\} \vdash \lambda f.\lambda x.f(f\ x) : \forall\gamma.(\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)$$

This turns out to be the best possible typing for this expression; formally, we say that the typing is *principal.* This means that any type whatsoever of $\lambda f.\lambda x.f(f x)$ can be derived from the type $\forall\gamma.(\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)$ by instantiating γ suitably; technically, we say that all the types of $\lambda f.\lambda x.f(f x)$ are *instances* of the principal type.

On the expression **let** $x = e$ **in** e', type inference proceeds by first finding a principal type σ for x, making the assumption $x : \sigma$, and then finding a type for e'.

Observe that, semantically, **let** $x = e$ **in** e' is the same as $(\lambda x.e')e$. They have, however, different typing behavior: in typing **let** $x = e$ **in** e', we may give x a *quantified* type when we type e'; in contrast, in typing $(\lambda x.e')e$, we

[7] The λ-calculus expression $\lambda x.e$ denotes a function with one parameter, x, and with body e. Hence $\lambda x.x$ denotes the identity function.

[8] This terminology comes from natural-deduction theorem proving.

must give x an *unquantified* type when we type e'. This discrepancy, known as the *let anomaly*, is forced by the restriction to outermost quantification: if we find the typing $e' : \tau$ under the assumption $x : \sigma$, where σ is quantified, then when we discharge the assumption $x : \sigma$ we will obtain the typing $\lambda x.e' : \sigma \to \tau$, but the type $\sigma \to \tau$ will not be outermost quantified.

In [DM82], it is shown that the Hindley/Milner/Damas type system has two especially nice properties: first, every well-typed program has a principal type, and second, principal types may be inferred using algorithm W. More precisely, given any program e, algorithm W finds a principal type for e if e is well typed and fails otherwise. Another way of stating this is to say that W is *sound* and *complete* with respect to the typing rules.

One might claim that Pascal, too, has principal types—after all, every Pascal program has a unique type. But, as can be seen from the discussion of *swap* above, this is true for a trivial reason: the type declarations required by Pascal force programs to be overspecialized. If, however, we *erase* all type declarations from *swap*, then the typing rules of Pascal allow *swap* to be given many types, none of which is principal.

For this reason, principal typing results must always be formulated with respect to the *implicitly* typed version of a language; that is, with respect to the version of the language in which all type declarations are erased. With respect to an implicitly typed language, the existence of principal types says, intuitively, that the set of type formulas in the language is expressive enough to capture all the polymorphism implied by the typing rules. It follows that increasing the power of the typing rules of the language demands a corresponding increase in the richness of the set of type formulas; this theme will recur throughout this dissertation.

1.6 Extensions to Hindley/Milner/Damas Polymorphism

One line of research aims at extending Hindley/Milner/Damas polymorphism by removing the restriction to outermost quantified types. This leads to the *polymorphic λ-calculus* of Girard and Reynolds [Gir72,Rey74]. It is at present unknown whether type inference is decidable for this language; for this reason the polymorphic λ-calculus is normally defined with explicit type declarations. [9] Research into the type inference problem for this (and related) languages is described in [Lei83,McC84,KT90]; we will not consider this extension further in this dissertation.

[9] In fact, the instantiation of quantified type variables is usually viewed as a kind of *application*, suggesting that types are somehow values.

Instead, we will study two kinds of polymorphism that cannot be expressed within the Hindley/Milner/Damas type system: *overloading* and *subtyping*. In the Hindley/Milner/Damas type system, an assumption set may contain at most *one* typing assumption for an identifier; this makes it impossible to express the overloadings of $\leq$. For $\leq$ has types $char \rightarrow char \rightarrow bool$ and $real \rightarrow real \rightarrow bool$, but it does not have type

$$\forall \alpha . \alpha \rightarrow \alpha \rightarrow bool.$$

So any single typing $\leq : \sigma$ is either too narrow or too broad. Also, in the Hindley/Milner/Damas type system, there is no way to express subtype inclusions such as $int \subseteq real$.

The goal of this dissertation, then, is to extend the Hindley/Milner/Damas type system to incorporate overloading and subtyping while preserving the existence of principal types and the ability to do type inference. As suggested in the discussion of principal types above, this requires a richer set of type formulas. The key device needed is *constrained (universal) quantification*, in which quantified variables are allowed only those instantiations that satisfy a set of *constraints*.

To deal with overloading, we require *typing* constraints of the form $x : \tau$, where x is an overloaded identifier. This allows us to give function *expon* (described in Section 1.3) the principal type

$$\forall \alpha \textbf{ with } * : \alpha \rightarrow \alpha \rightarrow \alpha,\ 1 : \alpha \,.\, \alpha \rightarrow int \rightarrow \alpha$$

which may be understood as saying that for all choices of α such that $*$ is defined on α and 1 has type α, *expon* has type $\alpha \rightarrow int \rightarrow \alpha$. It is easy to see why such constraints are necessary: since there may be many type assumptions for $*$ and 1, there is probably no single type formula that explicitly gives the types of *expon*; it is therefore necessary to use an *implicit* representation.

The typing

$$expon : \forall \alpha \textbf{ with } * : \alpha \rightarrow \alpha \rightarrow \alpha,\ 1 : \alpha \,.\, \alpha \rightarrow int \rightarrow \alpha$$

may also be understood as a universal Horn clause [Llo84]:

$$expon : \alpha \rightarrow int \rightarrow \alpha \quad \textbf{if} \quad * : \alpha \rightarrow \alpha \rightarrow \alpha,\ 1 : \alpha.$$

The reader may find this viewpoint useful.

Another example is a function *sort* that takes as input a sequence and returns the sequence in sorted order, using $\leq$ to compare the sequence elements. Assuming that $\leq$ is overloaded, *sort* has principal type

$$\forall \alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool \,.\, seq(\alpha) \rightarrow seq(\alpha)\,.$$

To deal with subtyping, we require *inclusion* constraints of the form $\tau \subseteq \tau'$. Consider, for example, the function $\lambda f.\lambda x.f(f\ x)$ discussed earlier. Let us give this function the name *twice*. In the Hindley/Milner/Damas type system, *twice* has principal type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$. But in the presence of subtyping, this type is no longer principal—if $int \subseteq real$, then *twice* has type $(real \to int) \to (real \to int)$, but this type is not deducible from $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$. It turns out that the principal type of *twice* is

$$\forall \alpha, \beta \textbf{ with } \beta \subseteq \alpha \,.\, (\alpha \to \beta) \to (\alpha \to \beta).$$

Another example requiring inclusion constraints is

$$if\ true\ 2\ 1$$

where $if : \forall\alpha.bool \to \alpha \to \alpha \to \alpha$; this has principal type

$$\forall \alpha \textbf{ with } int \subseteq \alpha \,.\, \alpha \to \alpha.$$

An issue that arises with the use of constrained quantification is the *satisfiability* of constraint sets. A type with an unsatisfiable constraint set is *vacuous*; it has no instances. We must take care, therefore, not to call a program well typed unless we can give it a type with a satisfiable constraint set. Unfortunately, we will see that it is undecidable whether a given constraint set is satisfiable; this difficultly will force us to impose restrictions on overloading.

We now briefly outline the remainder of the dissertation. In Chapter 2 we consider overloading. We present the typing rules, give a type inference algorithm, W_o, and prove that it is sound and complete with respect to the typing rules; hence W_o computes principal types. Next we show that checking the satisfiability of constraint sets is undecidable. To deal with this we propose restrictions on overloading that make the satisfiability problem decidable, and yet allow most of the kinds of overloading that occur in practice.

In Chapter 3 we extend the type system of Chapter 2 by adding subtyping. We give another inference algorithm, W_{os}, and prove that it is sound and complete. An interesting feature of type inference with subtyping is the necessity of simplifying the inferred types—the unsimplified types are so complex that they are unintelligible. The simplification process involves a variant of unification called *shape unification*, graph algorithms such as strongly connected components and transitive reduction, and simplifications based on the monotonicities of type formulas.

In Chapter 4 we give a number of examples of type inference, survey related work, and indicate some future research directions.[10]

[10] The reader impatient for more examples may wish to look ahead to Chapter 4 now.

Chapter 2

Overloading

In this chapter, we develop the theory of polymorphism in the presence of overloading. We begin in Section 2.1 by presenting a type system that extends the Hindley/Milner/Damas type system by allowing overloading in type assumption sets and by allowing polymorphic types to have constraint sets as a way of expressing a kind of bounded polymorphism. In Section 2.2 we study properties of the type system and prove a normal-form theorem for derivations in our system. In Section 2.3 we address the type inference problem for our system, presenting algorithm W_o, which generalizes Milner's algorithm W [Mil78,DM82]. In Section 2.4 we establish the soundness and completeness of W_o, showing that our type system has the principal typing property. In Section 2.5 we address a problem deferred from consideration in Sections 2.3 and 2.4, namely the problem of checking the satisfiability of a constraint set. We show that the problem is undecidable in general and, indeed, that the typability problem for our type system is undecidable in general. Section 2.6 attempts to cope with these limitations by proposing restricted forms of overloading for which the satisfiability and typability problems are decidable.

2.1 The Type System

In order to study polymorphism in the presence of overloading in as clean a setting as possible, we take as our language the *core-ML* of Damas and Milner [DM82]. Given a set of *identifiers* (x , y , a , $\leq$, 1 , ...), the set of *expressions* is given by

$$e ::= x \mid \lambda x.e \mid e\,e' \mid \textbf{let } x = e \textbf{ in } e' .$$

As usual, we assume that application $e\,e'$ is left associative and that application binds more tightly than λ or **let**.

The expression **let** $x = e$ **in** e' is evaluated by evaluating e in the current environment, binding the value of e to x, and then evaluating e'. Notice therefore that **let** does not permit recursive definitions. When we wish to make recursive definitions, we will use an explicit fixed-point operator, *fix*.

The basic assertions of any type system are *typings* of the form $e : \sigma$. The typing $e : \sigma$ asserts that expression e has type σ. In the next three subsections, we present our type system, first defining the set of types and type schemes, and then presenting the rules for proving typings.

2.1.1 Types and Type Schemes

Given a set of *type variables* ($\alpha, \beta, \gamma, \delta, \ldots$) and a set of *type constructors* (*int*, *bool*, *char*, *set*, *seq*, ...) of various arities, we define the set of (unquantified) *types* by

$$\tau ::= \alpha \mid \tau \rightarrow \tau' \mid \chi(\tau_1, \ldots, \tau_n)$$

where χ is an n-ary type constructor. If χ is 0-ary, then the parentheses are omitted. As usual, $\rightarrow$ is taken to be right associative. Types will be denoted by τ, π, ρ, ϕ, or ψ.

Next we define the set of quantified types, or *type schemes*, by

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n \textbf{ with } x_1 : \tau_1, \ldots, x_m : \tau_m \,.\, \tau$$

We call $\{\alpha_1, \ldots, \alpha_n\}$ the set of *quantified variables* of σ, $\{x_1 : \tau_1, \ldots, x_m : \tau_m\}$ the set of *constraints* of σ, and τ the *body* of σ. The order of the quantified variables and of the constraints is irrelevant, and we assume that they are written without duplicates. If there are no quantified variables, "$\forall$" is omitted. If there are no constraints, the **with** is omitted. Type schemes will always be denoted by σ.

We will use overbars to denote sequences. Thus $\bar{\alpha}$ is an abbreviation for $\alpha_1, \ldots, \alpha_n$. The length of the sequence will be determined by context.

If $\sigma = \forall \bar{\alpha} \textbf{ with } C \,.\, \tau$, then a variable β occurring in C or τ is said to occur *bound* in σ if β is some α_i and to occur *free* in σ otherwise. We use the notation $fv(\sigma)$ to denote the set of type variables that occur free in σ.

2.1.2 Substitution and α-equivalence

A *substitution* is a set of simultaneous replacements for type variables:

$$[\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n]$$

where the α_i's are distinct. The substitution $[\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n]$ is applied to a type τ by simultaneously replacing all occurrences of each α_i by the corresponding τ_i. Substitutions are applied *on the right*, so the application of substitution S to type τ is denoted by τS. A substitution S may be applied to a typing $x : \tau$ to yield the typing $x : (\tau S)$, and a substitution may be applied to a set of typings by applying it to each typing in the set.

A type variable β is said to occur (free) in the substitution

$$[\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n]$$

if β is one of the α_i's or if β occurs in some τ_i.

Defining the application of a substitution to a type scheme is rather complicated, since we must deal with the usual worries about bound variables and capture [HS86]. In the interest of simplicity, we adopt a definition that does more renaming of bound variables than is strictly necessary. Suppose that we are given a type scheme

$$\sigma = \forall \alpha_1, \ldots, \alpha_n \textbf{ with } C \,.\, \tau$$

and a substitution S, where we assume that the α_i's are distinct. Let $\beta_1, \ldots, \beta_n$ be the alphabetically first distinct type variables of the universe of type variables that occur free in neither S nor σ. (We demand the alphabetically first such variables simply so that σS will be uniquely determined.[1]) Now we can define

$$(\forall \bar{\alpha} \textbf{ with } C \,.\, \tau)S = \forall \bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}]S \,.\, \tau[\bar{\alpha} := \bar{\beta}]S \,.$$

The *composition* of substitutions S and T is denoted by ST. It has the property that for all σ, $(\sigma S)T = \sigma(ST)$.

We sometimes need to modify substitutions. The substitution $S \oplus [\bar{\alpha} := \bar{\tau}]$ is defined by

$$\beta(S \oplus [\bar{\alpha} := \bar{\tau}]) = \begin{cases} \tau_i, & \text{if } \beta \text{ is some } \alpha_i, \\ \beta S, & \text{otherwise.} \end{cases}$$

The names of bound variables in type schemes are unimportant; the notion of α-*equivalence* [HS86] formalizes this idea. We say that

$$(\forall \alpha_1, \ldots, \alpha_n \textbf{ with } C \,.\, \tau) \equiv_\alpha (\forall \beta_1, \ldots, \beta_n \textbf{ with } C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}])$$

(assuming that the α_i's are distinct and the β_i's are distinct) if no β_i is free in $\forall \bar{\alpha}$ **with** $C \,.\, \tau$. If $\sigma \equiv_\alpha \sigma'$, we say that σ' is an α-*variant* of σ.

Lemma 2.1 $\equiv_\alpha$ *is an equivalence relation.*

Proof:

- Letting $\bar{\beta}$ be $\bar{\alpha}$, we see that $\equiv_\alpha$ is reflexive.

[1] Tofte [Tof88] avoids such an arbitrary choice by defining substitution to be a *relation* on type schemes, rather than a function, but this decision leads to a great deal of syntactic awkwardness.

- Suppose $\forall\bar{\alpha}$ **with** $C \,.\, \tau \equiv_\alpha \sigma$. Then $\sigma = \forall\bar{\beta}$ **with** $C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}]$, for some $\bar{\beta}$ not free in $\forall\bar{\alpha}$ **with** $C \,.\, \tau$. Now, $\bar{\alpha}$ are not free in σ, since each α_i has been replaced by bound variable β_i. Hence

$$\sigma \equiv_\alpha \forall\bar{\alpha}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\alpha}] \,.\, \tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\alpha}] .$$

  Since $[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\alpha}] = [\,]$, it follows that $\sigma \equiv_\alpha \forall\bar{\alpha}$ **with** $C \,.\, \tau$. So $\equiv_\alpha$ is symmetric.

- Suppose $(\forall\bar{\alpha}$ **with** $C \,.\, \tau) \equiv_\alpha \sigma$ and $\sigma \equiv_\alpha \sigma'$. Then for some $\bar{\beta}$ not free in $\forall\bar{\alpha}$ **with** $C \,.\, \tau$,

$$\sigma = \forall\bar{\beta}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}]$$

  and for some $\bar{\gamma}$ not free in σ,

$$\sigma' = \forall\bar{\gamma}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}] \,.\, \tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}] .$$

  Because the α_i's are distinct, the β_i's are distinct, and the γ_i's are distinct,

$$[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}] = [\bar{\alpha} := \bar{\gamma}].$$

  Also, $\bar{\gamma}$ are not free in $(\forall\bar{\alpha}$ **with** $C \,.\, \tau)$. So $\forall\bar{\alpha}$ **with** $C \,.\, \tau \equiv_\alpha \sigma'$. Hence $\equiv_\alpha$ is transitive.

QED

Lemma 2.2 *α-equivalence is preserved under substitution. That is, if $\sigma \equiv_\alpha \sigma'$, then $\sigma S \equiv_\alpha \sigma' S$.*

Proof: Let $\sigma = \forall\bar{\alpha}$ **with** $C \,.\, \tau$ and $\sigma' = \forall\bar{\beta}$ **with** $C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}]$. In fact we will show that $\sigma S = \sigma' S$, which of course implies that $\sigma S \equiv_\alpha \sigma' S$. Let $\bar{\gamma}$ be the alphabetically first distinct type variables not occurring free in S or σ. Because σ and σ' have the same free variables, $\bar{\gamma}$ are also the first distinct type variables not occurring free in S or σ'. Hence

$$\begin{array}{ll}
 & \sigma S \\
= & \qquad \ll \text{definition of } \sigma \gg \\
 & (\forall\bar{\alpha}\ \mathbf{with}\ C \,.\, \tau)S \\
= & \qquad \ll \text{definition of substitution} \gg \\
 & \forall\bar{\gamma}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\gamma}]S \,.\, \tau[\bar{\alpha} := \bar{\gamma}]S \\
= & \qquad \ll \text{since } [\bar{\alpha} := \bar{\gamma}] = [\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}] \gg \\
 & \forall\bar{\gamma}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}]S \,.\, \tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}]S \\
= & \qquad \ll \text{definition of substitution} \gg \\
 & (\forall\bar{\beta}\ \mathbf{with}\ C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}])S \\
= & \qquad \ll \text{definition of } \sigma' \gg \\
 & \sigma' S
\end{array}$$

QED

Lemma 2.3 *If* $\bar{\gamma}$ *are distinct type variables that do not occur free in* S *or in* $\forall\bar{\alpha}$ **with** $C\,.\,\tau$ *, then*

$$(\forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau)S \equiv_\alpha \forall\bar{\gamma}\ \textbf{with}\ C[\bar{\alpha} := \bar{\gamma}]S\,.\,\tau[\bar{\alpha} := \bar{\gamma}]S\,.$$

Proof: Let $\bar{\beta}$ be the alphabetically first distinct type variables not occurring free in S or $\forall\bar{\alpha}$ **with** $C\,.\,\tau$. Note that the $\bar{\gamma}$ are not free in

$$\forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}]S\,.$$

> (If γ_i occurs in $C[\bar{\alpha} := \bar{\beta}]S$ or $\tau[\bar{\alpha} := \bar{\beta}]S$, then since γ_i does not occur in S , γ_i occurs in $C[\bar{\alpha} := \bar{\beta}]$ or $\tau[\bar{\alpha} := \bar{\beta}]$. Since γ_i is not free in $\forall\bar{\alpha}$ **with** $C\,.\,\tau$, this means that γ_i is some β_j , which is bound in $\forall\bar{\beta}$ **with** $C[\bar{\alpha} := \bar{\beta}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}]S$.)

Also,

$$S[\bar{\beta} := \bar{\gamma}] = [\bar{\beta} := \bar{\gamma}]S,$$

since neither $\bar{\beta}$ nor $\bar{\gamma}$ occur in S . Hence

$$\begin{array}{ll}
 & (\forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau)S \\
= & \quad \ll \text{definition of substitution} \gg \\
 & \forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}]S \\
\equiv_\alpha & \quad \ll \text{definition of } \equiv_\alpha,\ \bar{\gamma} \text{ not free in above} \gg \\
 & \forall\bar{\gamma}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\gamma}]\,.\,\tau[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\gamma}] \\
= & \quad \ll \text{since } S[\bar{\beta} := \bar{\gamma}] = [\bar{\beta} := \bar{\gamma}]S \gg \\
 & \forall\bar{\gamma}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}]S \\
= & \quad \ll \text{since } [\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\gamma}] = [\bar{\alpha} := \bar{\gamma}] \gg \\
 & \forall\bar{\gamma}\ \textbf{with}\ C[\bar{\alpha} := \bar{\gamma}]S\,.\,\tau[\bar{\alpha} := \bar{\gamma}]S\,.
\end{array}$$

QED

2.1.3 The Typing Rules

The purpose of a type system is to prove typings $e : \sigma$. In general, however, the typing $e : \sigma$ depends on typings for the identifiers occurring free in e, so typing judgements are valid only with respect to a *type assumption set.*

A *type assumption set* A is a finite set of assumptions of the form $x : \sigma$. A type assumption set A may contain more than one typing for an identifier x ; in this case we say that x is *overloaded* in A . We next define some terminology regarding assumption sets. A typing of the form $x : \tau$ is called a τ*-assumption.* If there is an assumption about x in A, or if some assumption in A has a (hypoth) $A \vdash x : \sigma$, if $x : \sigma \in A$

($\rightarrow$-intro) $$\frac{A \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \rightarrow \tau'} \quad (x \text{ does not occur in } A)$$

($\rightarrow$-elim) $$\frac{\begin{array}{l} A \vdash e : \tau \rightarrow \tau' \\ A \vdash e' : \tau \end{array}}{A \vdash e\,e' : \tau'}$$

(let) $$\frac{\begin{array}{l} A \vdash e : \sigma \\ A \cup \{x : \sigma\} \vdash e' : \tau \end{array}}{A \vdash \textbf{let } x = e \textbf{ in } e' : \tau} \quad (x \text{ does not occur in } A)$$

($\forall$-intro) $$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall \bar{\alpha} \textbf{ with } C\,.\,\tau} \quad (\bar{\alpha} \text{ not free in } A)$$

($\forall$-elim) $$\frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha} \textbf{ with } C\,.\,\tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

($\equiv_\alpha$) $$\frac{\begin{array}{l} A \vdash e : \sigma \\ \sigma \equiv_\alpha \sigma' \end{array}}{A \vdash e : \sigma'}$$

Figure 2.1: Typing Rules with Overloading

constraint $x : \tau$, then we say that x *occurs* in A. We use the notation $id(A)$ to denote the set of identifiers occurring in A. Finally, we use the notation $fv(A)$ to denote the set of type variables occurring free in A.

The type inference rules given in Figure 2.1 allow us to prove *typing judgements* of the form
$$A \vdash e : \sigma,$$
which asserts that from A it follows that expression e has type σ. If C is a set of assumptions $x : \sigma$, then we use the notation
$$A \vdash C$$
to represent
$$A \vdash x : \sigma, \quad \text{for all } x : \sigma \text{ in } C.$$
This notation is used in rules ($\forall$-intro) and ($\forall$-elim). If $A \vdash e : \sigma$ for some σ, then we say that e is *well typed* with respect to A.

Here is an example of a derivation. Suppose that

$$A = \left\{ \begin{array}{l} \leq\, : char \rightarrow char \rightarrow bool, \\ \leq\, : \forall\alpha \textbf{ with } \leq\, : \alpha \rightarrow \alpha \rightarrow bool\,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool\,, \\ c : seq(char) \end{array} \right\}$$

We will derive the typing

$$A \vdash \textbf{let } f = \lambda x.(x \leq x) \textbf{ in } f(c) : bool\,.$$

We have

$$A \cup \{\leq\, : \beta \rightarrow \beta \rightarrow bool, x : \beta\} \vdash \leq\, : \beta \rightarrow \beta \rightarrow bool \tag{2.1}$$

by (hypoth),

$$A \cup \{\leq\, : \beta \rightarrow \beta \rightarrow bool, x : \beta\} \vdash x : \beta \tag{2.2}$$

by (hypoth),

$$A \cup \{\leq\, : \beta \rightarrow \beta \rightarrow bool, x : \beta\} \vdash \leq (x) : \beta \rightarrow bool \tag{2.3}$$

by ($\rightarrow$ -elim) on (2.1) and (2.2),

$$A \cup \{\leq\, : \beta \rightarrow \beta \rightarrow bool, x : \beta\} \vdash x \leq x : bool \tag{2.4}$$

by ($\rightarrow$ -elim) on (2.3) and (2.2),

$$A \cup \{\leq\, : \beta \rightarrow \beta \rightarrow bool\} \vdash \lambda x.(x \leq x) : \beta \rightarrow bool \tag{2.5}$$

by ($\rightarrow$ -intro) on (2.4),

$$A \vdash \leq\, : (\beta \rightarrow \beta \rightarrow bool)[\beta := char] \tag{2.6}$$

by (hypoth), and

$$A \vdash \lambda x.(x \leq x) : \forall\beta \textbf{ with } \leq\, : \beta \rightarrow \beta \rightarrow bool\,.\, \beta \rightarrow bool \tag{2.7}$$

by ($\forall$-intro) on (2.5) and (2.6). Now let

$$B = A \cup \{f : \forall\beta \textbf{ with } \leq\, : \beta \rightarrow \beta \rightarrow bool\,.\, \beta \rightarrow bool\}.$$

Then

$$B \vdash f : \forall\beta \textbf{ with } \leq\, : \beta \rightarrow \beta \rightarrow bool\,.\, \beta \rightarrow bool \tag{2.8}$$

by (hypoth),

$$B \vdash \leq\, : \forall\alpha \textbf{ with } \leq\, : \alpha \rightarrow \alpha \rightarrow bool\,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool \tag{2.9}$$

by (hypoth),

$$B \vdash \leq\, : char \rightarrow char \rightarrow bool \tag{2.10}$$

by (hypoth),

$$B \vdash \leq\ : seq(char) \rightarrow seq(char) \rightarrow bool \tag{2.11}$$

by ($\forall$ -elim) on (2.9) and (2.10) with $\alpha := char$,

$$B \vdash f : seq(char) \rightarrow bool \tag{2.12}$$

by ($\forall$ -elim) on (2.8) and (2.11) with $\beta := seq(char)$,

$$B \vdash c : seq(char) \tag{2.13}$$

by (hypoth),

$$B \vdash f(c) : bool \tag{2.14}$$

by ($\rightarrow$ -elim) on (2.12) and (2.13), and

$$A \vdash \textbf{let}\ \ f = \lambda x.(x \leq x)\ \ \textbf{in}\ \ f(c)\ : bool \tag{2.15}$$

by (let) on (2.7) and (2.14).

Rules (hypoth), ($\rightarrow$ -intro), ($\rightarrow$ -elim), and (let) are the same as in [DM82], except for the restrictions on ($\rightarrow$ -intro) and (let), which we discuss at the end of this section. Rules ($\forall$ -intro) and ($\forall$ -elim) are different, since they must deal with constraint sets. Notice in particular that rule (let) cannot be used to create an overloading for an identifier. Hence, the only overloadings in the language are those given by the initial assumption set.

Rule ($\equiv_\alpha$) is not included in [DM82], but it should be, for it is needed to correct a defect of natural deduction. The problem is that without ($\equiv_\alpha$), the names chosen for bound variables can make a difference, contrary to what we would expect. For example, without rule ($\equiv_\alpha$), if

$$A = \{f : \forall\beta.\alpha \rightarrow \beta,\ \ x : \alpha\},$$

then

$$A \vdash f(x) : \forall\beta.\beta$$

but

$$A \not\vdash f(x) : \forall\alpha.\alpha\,,$$

because α is free in A. More seriously, with rule ($\equiv_\alpha$) we can prove that if $A \vdash e : \sigma$, then $AS \vdash e : \sigma S$, but without ($\equiv_\alpha$) this fails. For if

$$A = \{f : \forall\beta.\alpha \rightarrow \beta \rightarrow \gamma,\ \ x : \alpha\},\ \ S = [\gamma := \beta],$$

then

$$A \vdash f(x) : \forall\beta.\beta \rightarrow \gamma$$

but $AS = \{f : \forall\delta.\alpha \rightarrow \delta \rightarrow \beta, x : \alpha\}$, $(\forall\beta.\beta \rightarrow \gamma)S = \forall\alpha.\alpha \rightarrow \beta$, and

$$AS \not\vdash f(x) : \forall\alpha.\alpha \rightarrow \beta.$$

The second hypothesis of the ($\forall$-intro) rule says that a constraint set can be moved into a type scheme only if the constraint set is satisfiable—the typing $A \vdash e : \forall\bar{\alpha}$ **with** $C\,.\,\tau$ cannot be derived unless we know that there is some way of instantiating the $\bar{\alpha}$ so that C is satisfied. This restriction is crucial in preventing many nonsensical expressions from being well typed. For example, if

$$A = \{+ : int \rightarrow int \rightarrow int,\ \ + : real \rightarrow real \rightarrow real,\ \ true : bool\},$$

then without the satisfiability condition we would have

$$A \vdash (true + true) : \mathbf{with}\ + : bool \rightarrow bool \rightarrow bool\,.\,bool$$

even though $+$ doesn't work on *bool*!

Overloading has also been investigated by Kaes [Kae88] and by Wadler and Blott [WB89]. Their systems do not require the satisfiability of constraint sets in type schemes, with the result that certain nonsensical expressions, such as $true + true$, are well typed in their systems. [2]

The restrictions on ($\rightarrow$-intro) and (let) forbid the rebinding of an identifier using λ or **let**. Hence to get the usual block structure in expressions, we would need to define the notion of $\equiv_\alpha$ on expressions to allow the renaming of bound identifiers. For simplicity only, we will not bother to do this.

The usual rules [DM82] for ($\rightarrow$-intro) and (let) are

$$\frac{A_x \cup \{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \rightarrow \tau'}$$

and

$$\frac{\begin{array}{l} A \vdash e : \sigma \\ A_x \cup \{x : \sigma\} \vdash e' : \tau \end{array}}{A \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau}$$

where A_x denotes A with any assumptions about x deleted. It turns out that under these rules, a number of obviously desirable properties fail. For example, one would imagine that if $A \cup B \vdash e : \sigma$ and $A \vdash B$, then $A \vdash e : \sigma$. But if

$$A = \{y : \forall\alpha\ \mathbf{with}\ x : \alpha\,.\,\alpha,\ \ x : int\},$$

$$B = \{y : int\},$$

then with respect to the usual ($\rightarrow$-intro) rule, $A \cup B \vdash \lambda x.y : bool \rightarrow int$ and $A \vdash B$, but $A \not\vdash \lambda x.y : bool \rightarrow int$.

[2] In Kaes' system, $true + true$ isn't well typed, but examples like it are.

2.1.4 The Instance Relation and Principal Types

Given a typing $A \vdash e : \sigma$, other types for e may be obtained by extending the derivation with the ($\forall$-elim) rule. The set of types thus derivable is captured by the *instance relation*, $\geq_A$.

Definition 2.1 ($\forall\bar{\alpha}$ **with** $C . \tau$) $\geq_A \tau'$ *if there is a substitution* $[\bar{\alpha} := \bar{\pi}]$ *such that*

- $A \vdash C[\bar{\alpha} := \bar{\pi}]$ *and*
- $\tau[\bar{\alpha} := \bar{\pi}] = \tau'$.

Furthermore we say that $\sigma \geq_A \sigma'$ *if for all* τ, $\sigma' \geq_A \tau$ *implies* $\sigma \geq_A \tau$. *In this case we say that* σ' *is an* instance *of* σ *with respect to* A.

This definition has the property that if $\sigma \geq_A \tau$, then for any e, $A \vdash e : \sigma$ implies $A \vdash e : \tau$, because the derivation of $A \vdash e : \sigma$ can be extended using ($\forall$-elim).

Like the $\geq_A$ relation of Wadler and Blott [WB89], our $\geq_A$ relation generalizes the $>$ relation of Damas and Milner [DM82]. Unlike Wadler and Blott, however, we follow the approach of Mitchell and Harper [MH88] in defining $\sigma \geq_A \sigma'$.[3] It is easy to see that $\geq_A$ is reflexive and transitive and that $\sigma \geq_A \tau$ is well defined.

In a polymorphic type system, an expression e may have many typings, and it is therefore desirable to have some "best" typing, which in some way subsumes all other typings. The notion of a *principal typing* makes this precise.

Definition 2.2 *The typing* $A \vdash e : \sigma$ *is said to be* principal *if for all typings* $A \vdash e : \sigma'$, $\sigma \geq_A \sigma'$. *In this case* σ *is said to be a principal type for* e *with respect to* A.

Principal types in our system cannot be understood by themselves; they depend upon the assumption set A. If we say that the principal type with respect to A of the exponentiation function *expon* is

$$\forall\alpha \textbf{ with } * : \alpha \to \alpha \to \alpha,\ 1 : \alpha . \alpha \to int \to \alpha\ ,$$

we must look at A to discover what overloadings of $*$ and 1 are available. Only then do we know how *expon* can be used.

[3] The Damas/Milner approach to defining $\sigma > \sigma'$ is syntactic; roughly speaking, it says that $\sigma > \sigma'$ if a derivation of $A \vdash e : \sigma$ can be extended to a derivation of $A \vdash e : \sigma'$ using ($\forall$-elim) followed by ($\forall$-intro). The Mitchell/Harper approach, in contrast, is semantic. On $>$ the two approaches are equivalent, but on $\geq_A$ the syntactic approach (adopted by Wadler and Blott) is weaker.

Because one must look to the assumption set A to determine what overloadings are available, overloaded symbols do not have very satisfying principal types. For example, suppose that A contains the following assumptions:

$$\begin{aligned}
&\leq : char \rightarrow char \rightarrow bool, \\
&\leq : real \rightarrow real \rightarrow bool, \\
&\leq : \forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\,seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool\ .
\end{aligned}$$

Then one principal type for $\leq$ with respect to A is

$$\forall\alpha \textbf{ with } \leq : \alpha\,.\,\alpha\ ,$$

which obviously doesn't convey anything! A somewhat better principal type is

$$\forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\,\alpha \rightarrow \alpha \rightarrow bool\ ,$$

but this isn't very informative either.[4] In practice, of course, one is seldom interested in the principal types of overloaded symbols; it is the principal types of compound programs, like *expon*, that are of interest, and such programs have principal types that seem reasonable.

In Damas and Milner's system principal types are unique up to $\equiv_\alpha$; in our system, because of the constraint sets in type schemes, there can be more interesting non-unique principal types. Above, we gave two different principal types for $\leq$. Similarly, if

$$A = \{f : int,\ f : bool,\ g : bool,\ g : char\},$$

then if $\forall\alpha \textbf{ with } f : \alpha,\ g : \alpha\,.\,\alpha$ is a principal type for some expression e with respect to A, then so is $bool$, because the only way to satisfy the constraint set $\{f : \alpha,\ g : \alpha\}$ is to let α be $bool$.

The reader may be uneasy about the existence of multiple principal types, especially since, under our definition of $\geq_A$, they are not all interderivable. In the inference algorithm, however, it is only necessary to generate τ types from a principal type, not σ types. Further, *any* principal type of an expression can be used to derive *all* the τ types of the expression. So multiple principal types are not a problem.

---

[4] These two principal types for $\leq$ help to explain why we use the semantic approach to defining $\geq_A$. Under the syntactic definition,

$$\forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\,\alpha \rightarrow \alpha \rightarrow bool \quad \not\geq_A \quad \forall\alpha \textbf{ with } \leq : \alpha\,.\,\alpha\ ,$$

so it would appear that

$$\forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\,\alpha \rightarrow \alpha \rightarrow bool$$

is not principal. But Wadler and Blott want this to be a principal type of $\leq$, so they have arranged their type system so that $\leq$ does *not* have type $\forall\alpha \textbf{ with } \leq : \alpha\,.\,\alpha$. Besides being unreasonable, this clutters the type system.

2.2 Properties of the Type System

In this section we establish a number of useful properties of the type system. We begin by showing that, roughly speaking, typing derivations are preserved under substitution. This property will be used many times in what follows.

Lemma 2.4 *If $A \vdash e : \sigma$ then $AS \vdash e : \sigma S$. Furthermore, given any derivation of $A \vdash e : \sigma$, a derivation of $AS \vdash e : \sigma S$ can be constructed that uses the same sequence of steps except that it inserts a ($\equiv_\alpha$) step after each ($\forall$-intro) step.*

Proof: By induction on the length of the derivation of $A \vdash e : \sigma$. Consider the last step of the derivation:

(hypoth) The derivation is simply $A \vdash x : \sigma$ where $x : \sigma \in A$. So $x : \sigma S \in AS$, and $AS \vdash x : \sigma S$.

($\rightarrow$-intro) The derivation ends with

$$\frac{A \cup \{x : \tau\} \vdash e' : \tau'}{A \vdash \lambda x.e' : \tau \rightarrow \tau'}$$

where x does not occur in A. By induction, $(A \cup \{x : \tau\})S \vdash e' : \tau' S$. Since $(A \cup \{x : \tau\})S = AS \cup \{x : \tau S\}$, and since x does not occur in AS, ($\rightarrow$-intro) can be used to give

$$AS \vdash \lambda x.e' : \tau S \rightarrow \tau' S$$

and $\tau S \rightarrow \tau' S = (\tau \rightarrow \tau')S$.

($\rightarrow$-elim) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e' : \tau \rightarrow \tau' \\ A \vdash e'' : \tau \end{array}}{A \vdash e' e'' : \tau'}$$

By induction, $AS \vdash e' : (\tau \rightarrow \tau')S$ and $AS \vdash e'' : \tau S$. Since $(\tau \rightarrow \tau')S = \tau S \rightarrow \tau' S$, ($\rightarrow$-elim) can be used to give $AS \vdash e'e'' : \tau' S$.

(let) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e' : \sigma' \\ A \cup \{x : \sigma'\} \vdash e'' : \tau \end{array}}{A \vdash \textbf{let } x = e' \textbf{ in } e'' : \tau}$$

where x does not occur in A. By induction, $AS \vdash e' : \sigma' S$ and $(A \cup \{x : \sigma'\})S \vdash e'' : \tau S$. Since $(A \cup \{x : \sigma'\})S = AS \cup \{x : \sigma' S\}$ and since x does not occur in AS, it follows from (let) that $AS \vdash \textbf{let } x = e' \textbf{ in } e'' : \tau S$.

($\forall$ -intro) The derivation ends with

$$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau}$$

where $\bar{\alpha}$ are not free in A. Let $\bar{\gamma}$ be distinct type variables occurring free in neither S, nor $\forall \bar{\alpha}$ **with** $C \,.\, \tau$, nor A. By induction,

$$(A \cup C)[\bar{\alpha} := \bar{\gamma}]S \vdash e : \tau[\bar{\alpha} := \bar{\gamma}]S$$

and

$$AS \vdash C[\bar{\alpha} := \bar{\pi}]S.$$

Now $(A \cup C)[\bar{\alpha} := \bar{\gamma}]S = AS \cup C[\bar{\alpha} := \bar{\gamma}]S$, since the $\bar{\alpha}$ are not free in A. Also, $[\bar{\alpha} := \bar{\pi}]S$ and $[\bar{\alpha} := \bar{\gamma}]S[\bar{\gamma} := \bar{\pi}S]$ agree on C.

> (Both substitutions take $\alpha_i \mapsto \pi_i S$. On any other variable δ in C, both substitutions take $\delta \mapsto \delta S$, as no γ_j can occur in δS.)

Hence

$$AS \cup C[\bar{\alpha} := \bar{\gamma}]S \vdash e : \tau[\bar{\alpha} := \bar{\gamma}]S$$

and

$$AS \vdash C[\bar{\alpha} := \bar{\gamma}]S[\bar{\gamma} := \bar{\pi}S].$$

So, since $\bar{\gamma}$ are not free in AS, by ($\forall$-intro)

$$AS \vdash e : \forall \bar{\gamma} \textbf{ with } C[\bar{\alpha} := \bar{\gamma}]S \,.\, \tau[\bar{\alpha} := \bar{\gamma}]S \,.$$

By Lemma 2.3,

$$\forall \bar{\gamma} \textbf{ with } C[\bar{\alpha} := \bar{\gamma}]S \,.\, \tau[\bar{\alpha} := \bar{\gamma}]S \equiv_\alpha (\forall \bar{\alpha} \textbf{ with } C \,.\, \tau)S,$$

so by inserting an extra ($\equiv_\alpha$) step we obtain

$$AS \vdash e : (\forall \bar{\alpha} \textbf{ with } C \,.\, \tau)S.$$

($\forall$ -elim) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

By induction,

$$AS \vdash e : (\forall \bar{\alpha} \textbf{ with } C \,.\, \tau)S$$

and

$$AS \vdash C[\bar{\alpha} := \bar{\pi}]S.$$

Let $\bar{\beta}$ be the alphabetically first distinct type variables occurring free in neither S nor $\forall\bar{\alpha}$ **with** $C\,.\,\tau$. Then

$$(\forall\bar{\alpha} \textbf{ with } C\,.\,\tau\,)S = \forall\bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}]S\,.\,\tau[\bar{\alpha} := \bar{\beta}]S\,.$$

By the same argument used in the ($\forall$-intro) case above,

$$C[\bar{\alpha} := \bar{\pi}]S = C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}S].$$

So by ($\forall$-elim),

$$AS \vdash e : \tau[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}S],$$

and as above,

$$\tau[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}S] = \tau[\bar{\alpha} := \bar{\pi}]S.$$

($\equiv_\alpha$) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \sigma' \\ \sigma' \equiv_\alpha \sigma \end{array}}{A \vdash e : \sigma}$$

By induction, $AS \vdash e : \sigma' S$, and by Lemma 2.2, $\sigma' S \equiv_\alpha \sigma S$. So by ($\equiv_\alpha$), $AS \vdash e : \sigma S$.

QED

We now begin the development of the main result of this section, a normal-form theorem for derivations in our type system. Let ($\forall$-elim$'$) be the following weakened ($\forall$-elim) rule:

$$(\forall\text{-elim}') \qquad \frac{\begin{array}{l} (x : \forall\bar{\alpha} \textbf{ with } C\,.\,\tau\,) \in A \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash x : \tau[\bar{\alpha} := \bar{\pi}].}$$

Write $A \vdash' e : \sigma$ if $A \vdash e : \sigma$ is derivable in the system obtained by deleting the ($\forall$-elim) rule and replacing it with the ($\forall$-elim$'$) rule. Obviously, $A \vdash' e : \sigma$ implies $A \vdash e : \sigma$. We will see that the converse holds as well, so $\vdash'$ derivations may be viewed as a *normal form* for $\vdash$ derivations. A similar normal-form result for Damas and Milner's type system appears in [CDDK86] and is discussed in [Tof88].

The usefulness of the restricted rules is that if $A \vdash' e : \tau$, then the form of e uniquely determines the last rule used in the derivation: if e is x the derivation ends with ($\forall$-elim$'$), if e is $\lambda x.e'$ the derivation ends with ($\rightarrow$-intro), if e is $e'\,e''$ the derivation ends with ($\rightarrow$-elim), and if e is **let** $x = e'$ **in** e'' the derivation ends with (let). In contrast, a derivation of $A \vdash e : \tau$ is much less predictable: at any point in the derivation there may be a use of ($\forall$-intro) followed by a use of ($\forall$-elim).

Observe that since a ($\forall$-elim$'$) step is equivalent to a (hypoth) step followed by a ($\forall$-elim) step, it follows immediately from Lemma 2.4 that $\vdash'$ derivations are also preserved under substitution:

Corollary 2.5 *If $A \vdash' e : \sigma$ then $AS \vdash' e : \sigma S$. Furthermore, given any derivation of $A \vdash' e : \sigma$, a derivation of $AS \vdash' e : \sigma S$ can be constructed that uses the same sequence of steps except that it inserts a ($\equiv_\alpha$) step after each ($\forall$-intro) step.*

Before proving the normal-form theorem, we show a number of auxiliary lemmas. These auxiliary lemmas, like Lemma 2.4, are proved by induction on the length of the derivation. Because the proofs are tedious, we relegate them to Appendix A.

Lemma 2.6 *If x does not occur in A, $x \neq y$, and $A \cup \{x : \sigma\} \vdash' y : \tau$, then $A \vdash' y : \tau$.*

Proof: See Appendix A. **QED**

Lemma 2.7 *If $A \vdash' e : \sigma$ then $A \cup B \vdash' e : \sigma$, provided that no identifier occurring in B is λ-bound or let-bound in e.*

Proof: See Appendix A. **QED**

Lemma 2.8 *Let B be a set of τ-assumptions. If $A \cup B \vdash' e : \sigma$ and $A \vdash' B$, then $A \vdash' e : \sigma$.*

Proof: See Appendix A. **QED**

With the above lemmas in hand, we can prove the normal-form theorem for derivations:

Theorem 2.9 *$A \vdash e : \sigma$ if and only if $A \vdash' e : \sigma$.*

Proof: It is immediate that $A \vdash' e : \sigma$ implies $A \vdash e : \sigma$, since a ($\forall$-elim$'$) step can be replaced by a (hypoth) step followed by a ($\forall$-elim) step. We prove that $A \vdash e : \sigma$ implies $A \vdash' e : \sigma$ by induction on the length of the derivation of $A \vdash e : \sigma$. All cases follow immediately from the induction hypothesis except for ($\forall$-elim).

(**$\forall$-elim**) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha} \textbf{ with } C \, . \, \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

By induction we have

$$A \vdash' C[\bar{\alpha} := \bar{\pi}].$$

Now, the derivation of $A \vdash e : \forall\bar{\alpha}$ **with** $C \,.\, \tau$ ends with (hypoth), ($\forall$-intro), or ($\equiv_\alpha$).

(hypoth) We can simply use ($\forall$-elim$'$) to get $A \vdash' e : \tau[\bar{\alpha} := \bar{\pi}]$.

($\forall$-intro) The derivation ends with

$$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\rho}] \end{array}}{A \vdash e : \forall\bar{\alpha} \textbf{ with } C \,.\, \tau}$$

where $\bar{\alpha}$ are not free in A. By induction, $A \cup C \vdash' e : \tau$. By Corollary 2.5,

$$(A \cup C)[\bar{\alpha} := \bar{\pi}] \vdash' e : \tau[\bar{\alpha} := \bar{\pi}].$$

Since $\bar{\alpha}$ are not free in A,

$$(A \cup C)[\bar{\alpha} := \bar{\pi}] = A \cup C[\bar{\alpha} := \bar{\pi}].$$

Since $A \vdash' C[\bar{\alpha} := \bar{\pi}]$, by Lemma 2.8 we have

$$A \vdash' e : \tau[\bar{\alpha} := \bar{\pi}].$$

($\equiv_\alpha$) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \sigma' \\ \sigma' \equiv_\alpha \forall\bar{\alpha} \textbf{ with } C \,.\, \tau \end{array}}{A \vdash e : \forall\bar{\alpha} \textbf{ with } C \,.\, \tau}$$

where for some $\bar{\beta}$ not free in $\forall\bar{\alpha}$ **with** $C \,.\, \tau$,

$$\sigma' = \forall\bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}].$$

Note that

$$C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}] = C[\bar{\alpha} := \bar{\pi}]$$

and

$$\tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}] = \tau[\bar{\alpha} := \bar{\pi}],$$

so the ($\equiv_\alpha$) step can be dropped and we can instead use the derivation

$$\frac{\begin{array}{l} A \vdash e : \forall\bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}] \\ A \vdash C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}]}$$

This is a shorter derivation, so by induction,

$$A \vdash' e : \tau[\bar{\alpha} := \bar{\pi}].$$

QED

Theorem 2.9 leads to a number of immediate corollaries:

Corollary 2.10 *If x does not occur in A, $x \neq y$, and $A \cup \{x : \sigma\} \vdash y : \tau$, then $A \vdash y : \tau$.*

Corollary 2.11 *If $A \vdash e : \sigma$ then $A \cup B \vdash e : \sigma$, provided that no identifier occurring in B is λ-bound or let-bound in e.*

Corollary 2.12 *Let B be a set of τ-assumptions. If $A \cup B \vdash e : \sigma$ and $A \vdash B$, then $A \vdash e : \sigma$.*

Henceforth we will be fairly casual about the distinction between $\vdash$ and $\vdash'$.

Lemma 2.13 *If $A \vdash e : \sigma$ then no identifier occurring in A is λ-bound or let-bound in e.*

Proof: See Appendix A. **QED**

This lemma gives us the following variant of Corollary 2.11. The variant will be used in proving algorithm W_o sound.

Corollary 2.14 *If $A \vdash e : \sigma$ then $A \cup B \vdash e : \sigma$, provided that all identifiers occurring in B also occur in A.*

We conclude this section by introducing a property that is needed to ensure that a type assumption set is, in a certain sense, well-behaved. The property is motivated by the following observation: the satisfiability condition of the ($\forall$-intro) rule prevents the derivation of 'vacuous' typings $A \vdash e : \forall\bar{\alpha}$ **with** $C . \tau$ where C is unsatisfiable, but such vacuous typings could be present in A itself.

Definition 2.3 *A has* satisfiable constraints *if whenever $(x : \forall\bar{\alpha}$ **with** $C . \tau) \epsilon A$, there exists a substitution $[\bar{\alpha} := \bar{\pi}]$ such that $A \vdash C[\bar{\alpha} := \bar{\pi}]$.*

Note that it is no real restriction to limit ourselves to assumption sets with satisfiable constraints, since any assumption with an unsatisfiable constraint set is basically useless. The essential property of assumption sets with satisfiable constraints is that any well-typed program has *unquantified* types; no well-typed program has only quantified types.

Lemma 2.15 *If A has satisfiable constraints and $A \vdash e : \sigma$, then there exists a type τ such that $A \vdash e : \tau$.*

Proof: If σ is a τ-type, there is nothing to show. Otherwise, we can strip off any final ($\equiv_\alpha$) steps in the derivation $A \vdash e : \sigma$ to obtain a derivation $A \vdash e : \sigma'$ ending with either (hypoth) or ($\forall$-intro).

In the (hypoth) case, the derivation is simply $A \vdash x : \sigma'$ where $x : \sigma' \epsilon A$. Say σ' is of the form $\forall\bar{\alpha}$ **with** $C . \tau$; since A has satisfiable constraints it follows that there is a substitution $[\bar{\alpha} := \bar{\pi}]$ such that $A \vdash C[\bar{\alpha} := \bar{\pi}]$. Hence the derivation may be extended using ($\forall$-elim), giving $A \vdash x : \tau[\bar{\alpha} := \bar{\pi}]$.

In the ($\forall$-intro) case, the derivation must end with

$$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall\bar{\alpha}\ \textbf{with}\ C . \tau}$$

where $\bar{\alpha}$ are not free in A. Again the derivation may be extended using ($\forall$-elim), giving $A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]$. **QED**

Lemma 2.16 *If A has satisfiable constraints and there exists a substitution $[\bar{\alpha} := \bar{\pi}]$ such that $A \vdash C[\bar{\alpha} := \bar{\pi}]$, then $A \cup \{x : \forall\bar{\alpha}$ **with** $C . \tau\}$ has satisfiable constraints.*

Proof: By Corollary 2.11, we have $A \cup \{x : \forall\bar{\alpha}$ **with** $C . \tau\} \vdash C[\bar{\alpha} := \bar{\pi}]$. And if $y : \forall\bar{\beta}$ **with** $C' . \tau' \ \epsilon\ A$, then since A has satisfiable constraints there exists $[\bar{\beta} := \bar{\rho}]$ such that $A \vdash C'[\bar{\beta} := \bar{\rho}]$. Using Corollary 2.11 again, we get that $A \cup \{x : \forall\bar{\alpha}$ **with** $C . \tau\} \vdash C'[\bar{\beta} := \bar{\rho}]$. **QED**

In proving the completeness of algorithm W_o below, we must restrict our attention to assumption sets with satisfiable constraints.

2.3 Algorithm W_o

A fairly straightforward generalization of Milner's algorithm W [Mil78,DM82] can be used to infer principal types for expressions in our language. Algorithm W_o is given in Figure 2.2.

$W_o(A, e)$ returns a triple (S, B, τ), such that

$$AS \cup B \vdash e : \tau .$$

Informally, τ is the type of e, B is a set of τ-assumptions that describe all the uses of overloaded identifiers in e, and S is a substitution that contains refinements to the typing assumptions in A.[5]

[5] We informally discussed the refinement of A by Milner's algorithm W in Chapter 1.

$W_o(A, e)$ is defined by cases:

1. e is x

   if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
   return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else if $(x : \forall\bar{\alpha}$ **with** $C \,.\, \tau) \in A$,
   return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else *fail.*

2. e is $\lambda x.e'$

   if x occurs in A, *fail.*
   let $(S_1, B_1, \tau_1) = W_o(A \cup \{x : \alpha\}, e')$ where α is new;
   return $(S_1, B_1, \alpha S_1 \rightarrow \tau_1)$.

3. e is $e'e''$

   let $(S_1, B_1, \tau_1) = W_o(A, e')$;
   let $(S_2, B_2, \tau_2) = W_o(AS_1, e'')$;
   let $S_3 = \mathit{unify}(\tau_1 S_2, \tau_2 \rightarrow \alpha)$ where α is new;
   return $(S_1 S_2 S_3, B_1 S_2 S_3 \cup B_2 S_3, \alpha S_3)$.

4. e is **let** $x = e'$ **in** e''

   if x occurs in A, *fail.*
   let $(S_1, B_1, \tau_1) = W_o(A, e')$;
   let $(B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$;
   let $(S_2, B_2, \tau_2) = W_o(AS_1 \cup \{x : \sigma_1\}, e'')$;
   return $(S_1 S_2, B_1' S_2 \cup B_2, \tau_2)$.

Figure 2.2: Algorithm W_o

Case 1 of algorithm W_o makes use of the *least common generalization* (*lcg*) of an overloaded identifier.[6] The *lcg* of an overloaded identifier x captures any common structure among the overloadings of x. For example, given the overloadings

$$\begin{aligned}
&\leq : char \rightarrow char \rightarrow bool, \\
&\leq : real \rightarrow real \rightarrow bool, \\
&\leq : \forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool\,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool, \\
&\leq : \forall\alpha,\beta \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool,\ \leq : \beta \rightarrow \beta \rightarrow bool\,. \\
&\qquad (\alpha \times \beta) \rightarrow (\alpha \times \beta) \rightarrow bool,
\end{aligned}$$

the *lcg* of $\leq$ is $\forall\alpha.\alpha \rightarrow \alpha \rightarrow bool$, which tells us that in all of its instances $\leq$ takes two arguments of the same type and returns a $bool$.

Formally, we say that τ is a *common generalization* of $\rho_1, \ldots, \rho_n$ if there exist substitutions $S_1, \ldots, S_n$ such that $\tau S_i = \rho_i$ for all i, and we say that τ is a *least common generalization* of $\rho_1, \ldots, \rho_n$ if

1. τ is a common generalization of $\rho_1, \ldots, \rho_n$, and
2. if τ' is any common generalization of $\rho_1, \ldots, \rho_n$, there exists a substitution S such that $\tau' S = \tau$.

Finally, we say that $\forall\bar{\alpha}.\tau$ is an *lcg* of an identifier x that is overloaded in A if τ is a least common generalization of the bodies of those type schemes σ with $x : \sigma \in A$, and $\bar{\alpha}$ are the variables of τ. Least common generalizations are discussed in [Rey70], which gives an algorithm for computing them, and in [McC84] under the name *least general predecessors.*

The "new" type variables called for in algorithm W_o are produced by a new type variable generator, which produces an unbounded sequence of distinct type variables. Before $W_o(A, e)$ is called, the new type variable generator is initialized so that it does not generate any type variables occurring free in A.

Function *unify,* used in case 3 of W_o, returns the *most general unifier* [Rob65, Kni89] of its arguments. That is, $unify(\tau, \tau')$ returns a substitution U such that $\tau U = \tau' U$, failing if no such substitution exists; furthermore U has the property that for any V such that $\tau V = \tau' V$, V can be factored into UV' for some V'.

Function *close,* used in case 4 of W_o, is defined in Figure 2.3. The idea behind *close* is to take a derivation

$$A \cup B \vdash e : \tau$$

where B is a set of assumptions about overloaded identifiers and, roughly speaking, to apply ($\forall$-intro) as much as possible, yielding an assumption set B' and a type scheme σ such that

$$A \cup B' \vdash e : \sigma.$$

[6] In fact the soundness and completeness of W_o do not rely on the use of the *least* common generalizations of the overloaded identifiers; any common generalization will do. In particular, the type $\forall\alpha.\alpha$ can always be used, but this leads to less informative principal types.

$close(A, B, \tau)$:

let $\bar{\alpha}$ be the type variables free in B or τ but not in A;
let B'' be the set of constraints in B in which some α_i occurs;
if A has no free type variables,
  then if $satisfiable(B, A)$
    then $B' = \{\}$
    else *fail*
  else $B' = B$;
return $(B', \forall\bar{\alpha}$ **with** $B''.\tau)$.

Figure 2.3: Function *close*

If A has no free type variables, B' will be empty.

Function $satisfiable(B, A)$ used in *close* checks whether B is satisfiable with respect to A, that is, whether there exists a substitution S such that $A \vdash BS$. We defer discussion of how *satisfiable* might be computed until Section 2.5.

Actually, there is a considerable amount of freedom in specifying the exact behavior of *close*. It is possible to give fancier versions of *close* that try to do more simplification of the generated type σ. We have, therefore, extracted the properties of *close* needed to prove the soundness and completeness of W_o into two lemmas, Lemma 2.17 and Lemma 2.19; this allows us to replace the simple *close* of Figure 2.3 by any other *close* that satisfies the lemmas.[7]

If A has no free type variables, a principal typing for e can be obtained as follows: $W_o(A, e)$ yields (S, B, τ) such that $AS \cup B \vdash e : \tau$. Since A has no free type variables, this means that $A \cup B \vdash e : \tau$. If $(B', \sigma) = close(A, B, \tau)$ succeeds, then we will have $A \cup B' \vdash e : \sigma$ and $B' = \{\}$, so $A \vdash e : \sigma$, and this typing will be principal.

[7] In particular, under the reasonable assumption that there are no free variables in the initial assumption set, we may weaken the condition "A has no free type variables" on the third line of *close*, replacing it with the condition "no type variable in B is free in A". The weaker condition has the advantage of allowing certain type errors to be detected sooner.

$$
\begin{array}{l}
\mathit{fix}\ \lambda leq.\lambda x.\lambda y. \\
\quad \mathit{if}\ (\mathit{null?}\ x) \\
\quad\quad \mathit{true} \\
\quad\quad \mathit{if}\ (\mathit{null?}\ y) \\
\quad\quad\quad \mathit{false} \\
\quad\quad\quad \mathit{if}\ (=\ (\mathit{car}\ x)\ (\mathit{car}\ y)) \\
\quad\quad\quad\quad (\mathit{leq}\ (\mathit{cdr}\ x)\ (\mathit{cdr}\ y)) \\
\quad\quad\quad\quad (\leq\ (\mathit{car}\ x)\ (\mathit{car}\ y))
\end{array}
$$

Figure 2.4: Example *lexicographic*

Here is an example of a type inference using W_o. Suppose that

$$
A = \left\{
\begin{array}{l}
\mathit{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha, \\
\mathit{if} : \forall\alpha.\mathit{bool} \to \alpha \to \alpha \to \alpha, \\
= : \forall\alpha.\alpha \to \alpha \to \mathit{bool}, \\
\mathit{true} : \mathit{bool}, \\
\mathit{false} : \mathit{bool}, \\
\mathit{car} : \forall\alpha.\mathit{seq}(\alpha) \to \alpha, \\
\mathit{cdr} : \forall\alpha.\mathit{seq}(\alpha) \to \mathit{seq}(\alpha), \\
\mathit{null?} : \forall\alpha.\mathit{seq}(\alpha) \to \mathit{bool}, \\
\leq : \mathit{char} \to \mathit{char} \to \mathit{bool}, \\
\leq : \forall\alpha\ \mathbf{with}\ \leq : \alpha \to \alpha \to \mathit{bool}\,.\, \mathit{seq}(\alpha) \to \mathit{seq}(\alpha) \to \mathit{bool}
\end{array}
\right\}
$$

Identifier *fix* denotes a least fixed-point operator, which allows us to express recursion without the use of an explicit **letrec** construct.[8] Let *lexicographic* be the expression given in Figure 2.4. Function *lexicographic* takes two sequences x and y and tests whether x lexicographically precedes y, using $\leq$ to compare the elements of the sequences.

The call $W_o(A, \mathit{lexicographic})$ returns

$$
\left(
\begin{array}{l}
\left[
\begin{array}{l}
\gamma := \mathit{seq}(\lambda_1),\ \eta := \mathit{bool},\ \theta := \mathit{bool} \to \mathit{bool} \to \mathit{bool},\ \varepsilon := \mathit{bool},\ \iota := \\
\mathit{bool} \to \mathit{bool},\ \delta := \mathit{seq}(\lambda_1),\ \mu := \mathit{bool},\ \nu := \mathit{bool} \to \mathit{bool} \to \mathit{bool},\ \kappa := \\
\mathit{bool},\ \xi := \mathit{bool} \to \mathit{bool},\ \zeta := \lambda_1,\ \sigma := \lambda_1,\ \rho := \lambda_1,\ \tau := \lambda_1 \to \mathit{bool}, \\
\lambda := \lambda_1,\ \phi := \lambda_1,\ \upsilon := \lambda_1,\ \chi := \mathit{bool},\ \psi := \mathit{bool} \to \mathit{bool} \to \mathit{bool}, \\
\pi := \lambda_1,\ \alpha_1 := \mathit{seq}(\lambda_1),\ \beta := \mathit{seq}(\lambda_1) \to \mathit{seq}(\lambda_1) \to \mathit{bool},\ \omega := \lambda_1, \\
\delta_1 := \mathit{seq}(\lambda_1),\ \beta_1 := \mathit{seq}(\lambda_1) \to \mathit{bool},\ \varepsilon := \mathit{bool},\ \zeta_1 := \mathit{bool} \to \mathit{bool}, \\
\gamma_1 := \lambda_1,\ \iota_1 := \lambda_1,\ \theta_1 := \lambda_1,\ \kappa_1 := \lambda_1 \to \mathit{bool},\ \eta_1 := \lambda_1,\ \mu_1 := \lambda_1, \\
\nu_1 := \mathit{bool},\ o := \mathit{bool},\ \xi_1 := \mathit{bool},\ o_1 := \mathit{bool},\ \pi_1 := \mathit{bool},\ \alpha := \\
\mathit{seq}(\lambda_1) \to \mathit{seq}(\lambda_1) \to \mathit{bool},\ \rho_1 := \mathit{seq}(\lambda_1) \to \mathit{seq}(\lambda_1) \to \mathit{bool}
\end{array}
\right], \\
\{\leq : \lambda_1 \to \lambda_1 \to \mathit{bool}\}, \\
\mathit{seq}(\lambda_1) \to \mathit{seq}(\lambda_1) \to \mathit{bool}
\end{array}
\right)
$$

[8] More precisely, "**letrec** $x = e$ **in** e'" is equivalent to "**let** $x = \mathit{fix}\ \lambda x.e$ **in** e'". See [Rey85] for further discussion of *fix*.

Then the call $close(A,\ \{\leq\ :\ \lambda_1\ \rightarrow\ \lambda_1\ \rightarrow\ bool\},\ seq(\lambda_1)\ \rightarrow\ seq(\lambda_1)\ \rightarrow\ bool)$ returns

$$(\{\},\ \forall\lambda_1\ \mathbf{with}\ \leq : \lambda_1 \rightarrow \lambda_1 \rightarrow bool\,.\, seq(\lambda_1) \rightarrow seq(\lambda_1) \rightarrow bool\,),$$

giving the principal typing

$$A \vdash lexicographic : \forall\lambda_1\ \mathbf{with}\ \leq : \lambda_1 \rightarrow \lambda_1 \rightarrow bool\,.\, seq(\lambda_1) \rightarrow seq(\lambda_1) \rightarrow bool\,.$$

2.4 Properties of W_o

2.4.1 Soundness of W_o

First we have a lemma giving the properties needed of *close* in proving that W_o is sound.

Lemma 2.17 *If $(B', \sigma) = close(A, B, \tau)$ succeeds, then for any e, if $A \cup B \vdash e : \tau$ then $A \cup B' \vdash e : \sigma$. Also, every identifier occurring in B' or in the constraints of σ occurs in B.*

Proof: Suppose that $(B', \sigma) = close(A, B, \tau)$ succeeds and that $A \cup B \vdash e : \tau$. There are two cases to consider:

1. <u>A has no free type variables.</u>

   In this case $satisfiable(B, A)$ is true, so there exists a substitution S such that $A \vdash BS$. The only variables occurring in B are the $\bar{\alpha}$, so we may write $S = [\bar{\alpha} := \bar{\pi}]$ without loss of generality. Since $B'' \subseteq B$, we have $A \vdash B''[\bar{\alpha} := \bar{\pi}]$ and $A \vdash (B - B'')[\bar{\alpha} := \bar{\pi}]$. Now by the definition of B'', the $\bar{\alpha}$ do not occur in $B - B''$, so in fact we have $A \vdash B - B''$. Then, by Corollary 2.11, we have $A \cup B'' \vdash B - B''$. Since $A \cup B'' \cup (B - B'') \vdash e : \tau$, we may apply Corollary 2.12 to yield $A \cup B'' \vdash e : \tau$. The $\bar{\alpha}$ are by definition not free in A, so ($\forall$-intro) may be applied:

$$\frac{\begin{array}{l} A \cup B'' \vdash e : \tau \\ A \vdash B''[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall\bar{\alpha}\ \mathbf{with}\ B''\,.\,\tau}$$

   showing that $A \cup B' \vdash e : \sigma$.

2. <u>A has free type variables.</u>

   Let $\bar{\beta}$ be new type variables not occurring in A, B, or τ. By Lemma 2.4, we have $(A \cup B)[\bar{\alpha} := \bar{\beta}] \vdash e : \tau[\bar{\alpha} := \bar{\beta}]$. Then by Corollary 2.14, $(A \cup$

$B)[\bar\alpha := \bar\beta] \cup B \vdash e : \tau[\bar\alpha := \bar\beta]$. Now $(A \cup B)[\bar\alpha := \bar\beta] \cup B = A \cup B \cup B''[\bar\alpha := \bar\beta]$, by the definitions of $\bar\alpha$ and B''. Hence ($\forall$-intro) may be applied:

$$\frac{\begin{array}{l} A \cup B \cup B''[\bar\alpha := \bar\beta] \vdash e : \tau[\bar\alpha := \bar\beta] \\ A \cup B \vdash B''[\bar\alpha := \bar\beta][\bar\beta := \bar\alpha] \end{array}}{A \cup B \vdash e : \forall\bar\beta \textbf{ with } B''[\bar\alpha := \bar\beta] \,.\, \tau[\bar\alpha := \bar\beta]}$$

since the $\bar\beta$ are not free in $A \cup B$. Finally, since

$$\forall\bar\beta \textbf{ with } B''[\bar\alpha := \bar\beta] \,.\, \tau[\bar\alpha := \bar\beta] \equiv_\alpha \forall\bar\alpha \textbf{ with } B'' \,.\, \tau,$$

we may use ($\equiv_\alpha$) to get $A \cup B \vdash e : \forall\bar\alpha \textbf{ with } B'' \,.\, \tau$.

As for the second part of the lemma, it follows immediately from the fact that B' and B'' are subsets of B. QED

Theorem 2.18 *If $(S, B, \tau) = W_o(A, e)$ succeeds, then $AS \cup B \vdash e : \tau$. Furthermore, every identifier occurring in B is overloaded in A or occurs in some constraint of some assumption in A.*

Proof: By induction on the structure of e.

1. e is x

   Then $W_o(A, e)$ is defined as:

   > if x is overloaded in A with *lcg* $\forall\bar\alpha.\tau$,
   > return $([\,], \{x : \tau[\bar\alpha := \bar\beta]\}, \tau[\bar\alpha := \bar\beta])$ where $\bar\beta$ are new
   > else if $(x : \forall\bar\alpha \textbf{ with } C \,.\, \tau) \in A$,
   > return $([\,], C[\bar\alpha := \bar\beta], \tau[\bar\alpha := \bar\beta])$ where $\bar\beta$ are new
   > else *fail.*

   If x is overloaded in A, then $(S, B, \tau) = ([\,], \{x : \tau[\bar\alpha := \bar\beta]\}, \tau[\bar\alpha := \bar\beta])$, so $AS \cup B = A \cup \{x : \tau[\bar\alpha := \bar\beta]\}$. So by (hypoth), $AS \cup B \vdash x : \tau[\bar\alpha := \bar\beta]$. Also, every identifier occurring in B is overloaded in A.

   If x is not overloaded in A, then $(S, B, \tau) = ([\,], C[\bar\alpha := \bar\beta], \tau[\bar\alpha := \bar\beta])$, so $AS \cup B = A \cup C[\bar\alpha := \bar\beta]$. By (hypoth), $AS \cup B \vdash x : \forall\bar\alpha \textbf{ with } C \,.\, \tau$ and $AS \cup B \vdash C[\bar\alpha := \bar\beta]$. So by a (possibly trivial) use of ($\forall$-elim), $AS \cup B \vdash x : \tau[\bar\alpha := \bar\beta]$. Also, every identifier occurring in B occurs in a constraint of an assumption in A.

2. e is $\lambda x.e'$

   Then $W_o(A, e)$ is defined as:

   > if x occurs in A, *fail.*
   > let $(S_1, B_1, \tau_1) = W_o(A \cup \{x : \alpha\}, e')$ where α is new;
   > return $(S_1, B_1, \alpha S_1 \to \tau_1)$.

By induction, $(A \cup \{x : \alpha\})S_1 \cup B_1 \vdash e' : \tau_1$. Also, every identifier in B_1 is overloaded in $A \cup \{x : \alpha\}$ or occurs in some constraint of some assumption in $A \cup \{x : \alpha\}$. Since x does not occur in A, and since the type α has no constraints, this means that every identifier in B_1 is overloaded in A or occurs in some constraint of some assumption in A. In particular, x does not occur in B_1 and, therefore, x does not occur in $AS_1 \cup B_1$. So, since $(A \cup \{x : \alpha\})S_1 \cup B_1 = AS_1 \cup B_1 \cup \{x : \alpha S_1\}$, we can use ($\rightarrow$-intro) to get $AS_1 \cup B_1 \vdash \lambda x.e' : \alpha S_1 \rightarrow \tau_1$.

3. <u>e is $e'e''$</u>

Then $W_o(A, e)$ is defined as:

let $(S_1, B_1, \tau_1) = W_o(A, e')$;
let $(S_2, B_2, \tau_2) = W_o(AS_1, e'')$;
let $S_3 = \mathit{unify}(\tau_1 S_2, \tau_2 \rightarrow \alpha)$ where α is new;
return $(S_1S_2S_3, B_1S_2S_3 \cup B_2S_3, \alpha S_3)$.

By induction, $AS_1 \cup B_1 \vdash e' : \tau_1$ and $AS_1S_2 \cup B_2 \vdash e'' : \tau_2$, and every identifier occurring in B_1 or in B_2 is overloaded in A or occurs in some constraint of some assumption in A. Hence every identifier occurring in $B_1S_2S_3 \cup B_2S_3$ is overloaded in A or occurs in some constraint of some assumption in A. By Lemma 2.4, $AS_1S_2S_3 \cup B_1S_2S_3 \vdash e' : \tau_1S_2S_3$ and $AS_1S_2S_3 \cup B_2S_3 \vdash e'' : \tau_2S_3$. So by Corollary 2.14,

$$AS_1S_2S_3 \cup B_1S_2S_3 \cup B_2S_3 \vdash e' : \tau_1S_2S_3$$

and

$$AS_1S_2S_3 \cup B_1S_2S_3 \cup B_2S_3 \vdash e'' : \tau_2S_3.$$

By the definition of *unify*, $\tau_1S_2S_3 = \tau_2S_3 \rightarrow \alpha S_3$. Hence by ($\rightarrow$-elim), we have

$$AS_1S_2S_3 \cup B_1S_2S_3 \cup B_2S_3 \vdash e'e'' : \alpha S_3.$$

4. <u>e is **let** $x = e'$ **in** e''</u>

Then $W_o(A, e)$ is defined as:

if x occurs in A, *fail.*
let $(S_1, B_1, \tau_1) = W_o(A, e')$;
let $(B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$;
let $(S_2, B_2, \tau_2) = W_o(AS_1 \cup \{x : \sigma_1\}, e'')$;
return $(S_1S_2, B_1'S_2 \cup B_2, \tau_2)$.

By induction, $AS_1 \cup B_1 \vdash e' : \tau_1$, and every identifier occurring in B_1 is overloaded in A or occurs in some constraint of some assumption in A. So by Lemma 2.17, $AS_1 \cup B_1' \vdash e' : \sigma_1$ and every identifier occurring in

B_1' is overloaded in A or occurs in some constraint of some assumption in A. In particular, since x does not occur in A, x does not occur in B_1'. Also by induction, $(AS_1 \cup \{x : \sigma_1\})S_2 \cup B_2 \vdash e'' : \tau_2$, which is the same as $AS_1S_2 \cup B_2 \cup \{x : \sigma_1 S_2\} \vdash e'' : \tau_2$, and every identifier occurring in B_2 is overloaded in $AS_1 \cup \{x : \sigma_1\}$ or occurs in some constraint of some assumption in $AS_1 \cup \{x : \sigma_1\}$. Now x does not occur in A, and by Lemma 2.17 every identifier occurring in the constraints of σ_1 occurs in B_1. Hence every identifier occurring in B_2 is overloaded in A or occurs in some constraint of some assumption in A. In particular, then, x does not occur in B_2. Now Lemma 2.4 and Corollary 2.14 may be applied to give

$$AS_1S_2 \cup B_1'S_2 \cup B_2 \vdash e' : \sigma_1 S_2$$

and

$$AS_1S_2 \cup B_1'S_2 \cup B_2 \cup \{x : \sigma_1 S_2\} \vdash e'' : \tau_2.$$

So, since x does not occur in $AS_1S_2 \cup B_1'S_2 \cup B_2$, (let) may be used to get

$$AS_1S_2 \cup B_1'S_2 \cup B_2 \vdash \textbf{let } x = e' \textbf{ in } e'' : \tau_2.$$

QED

2.4.2 Completeness of W_o

Ultimately, we will prove the following principal typing result.

Theorem *Let A be an assumption set with satisfiable constraints and no free type variables. If e is well typed with respect to A, then $(S, B, \tau) = W_o(A, e)$ succeeds, $(B', \sigma) = close(A, B, \tau)$ succeeds, and the typing $A \vdash e : \sigma$ is principal.*

Reaching this result will take some time, however.

We start with a lemma that gives the properties of *close* needed to establish the completeness of W_o.

Lemma 2.19 *If $AS \vdash BS$, then $(B', \sigma) = close(A, B, \tau)$ succeeds and σ is of the form $\forall \bar{\alpha}$* **with** *$B''.\tau$. In addition,*

- *if A has no free type variables then $B' = \{\}$,*
- *$fv(\sigma) \subseteq fv(A)$, and*
- *$B' \cup B'' \subseteq B$.*

Proof: If A has free type variables, then $close(A, B, \tau)$ succeeds. If A has no free type variables, then the assumption $AS \vdash BS$ becomes $A \vdash BS$. Hence *satisfiable*(B, A) is *true*, and $close(A, B, \tau)$ succeeds.

The remaining parts of the lemma are immediate from the definition of $close(A, B, \tau)$. **QED**

Lemma 2.20 *If* $(S, B, \tau) = W_o(A, e)$ *succeeds, then every type variable occurring in* S, B, *or* τ *occurs free in* A *or else is generated by the new type variable generator.*

Proof: A straightforward induction on the structure of e, using properties of *unify* and Lemma 2.19. **QED**

Definition 2.4 *Let* A *and* A' *be assumption sets. We say that* A *is* stronger than A', *written* $A \succeq A'$, *if*

1. $A' \vdash x : \tau$ *implies* $A \vdash x : \tau$, *and*
2. $id(A) \subseteq id(A')$.

Roughly speaking, $A \succeq A'$ means that A can do anything that A' can. One would expect, then, that the following lemma would be true.

Lemma *If* $A' \vdash e : \tau$, A' *has satisfiable constraints, and* $A \succeq A'$, *then* $A \vdash e : \tau$.

Given this lemma, we can prove the following completeness theorem for W_o:

Theorem *If* $AS \vdash e : \tau$ *and* AS *has satisfiable constraints, then* $(S_0, B_0, \tau_0) = W_o(A, e)$ *succeeds and there exists a substitution* T *such that*

1. $S = S_0 T$, *except on new type variables of* $W_o(A, e)$,
2. $AS \vdash B_0 T$, *and*
3. $\tau = \tau_0 T$.

Unfortunately the lemma appears to defy a straightforward inductive proof. The essential difficulty is in the ($\forall$-intro) case. Consider the case where the derivation $A' \vdash e : \tau$ ends with (let):

$$\frac{\begin{array}{l} A' \vdash e' : \sigma \\ A' \cup \{x : \sigma\} \vdash e'' : \tau \end{array}}{A' \vdash \textbf{let } x = e' \textbf{ in } e'' : \tau}$$

We would need to use induction somehow to show that $A \vdash e' : \sigma'$ for some σ' such that

$$A \cup \{x : \sigma'\} \succeq A' \cup \{x : \sigma\},$$

allowing us to use induction on the second hypothesis. Suppose that the derivation of $A' \vdash e' : \sigma$ is by ($\forall$-intro):

$$\frac{\begin{array}{l} A' \cup C \vdash e' : \tau' \\ A' \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A' \vdash e' : \forall\bar{\alpha} \textbf{ with } C \,.\, \tau'}$$

Now we would like to use induction to show that $A \cup C \vdash e' : \tau'$. But we have a serious problem: it need not be the case that $A \cup C \succeq A' \cup C$! And in fact, it need not be the case that $A \cup C \vdash e' : \tau'$. For example, let $A = \{x : int, y : int\}$ and $A' = \{x : \forall\alpha \textbf{ with } y : \alpha \,.\, \alpha, y : int\}$. Then $A \succeq A'$. But $A \cup \{y : \beta\} \not\succeq A' \cup \{y : \beta\}$, and also $A \cup \{y : \beta\} \not\vdash x : \beta$ although $A' \cup \{y : \beta\} \vdash x : \beta$.

The point is that the derivation of $A \vdash e : \tau$ may have to have a very different structure from the derivation of $A' \vdash e : \tau$, so it does not seem possible to 'locally' modify the derivation of $A' \vdash e : \tau$ to arrive at a derivation of $A \vdash e : \tau$. How, then, can we find a derivation of $A \vdash e : \tau$? The answer is to use W_o to find it. But we haven't yet shown the completeness of W_o, and indeed the whole reason for introducing the lemma above was to prove the completeness theorem! So it appears necessary to combine the lemma and theorem above into a single theorem that yields both as corollaries and that allows both to be proved simultaneously. We shall now do this.

Theorem 2.21 *Suppose $A' \vdash e : \tau$, A' has satisfiable constraints, and $AS \succeq A'$. Then $(S_0, B_0, \tau_0) = W_o(A, e)$ succeeds and there exists a substitution T such that*

1. *$S = S_0 T$, except on new type variables of $W_o(A, e)$,*
2. *$AS \vdash B_0 T$, and*
3. *$\tau = \tau_0 T$.*

Proof: By induction on the structure of e. By Theorem 2.9, $A' \vdash' e : \tau$.

- e is x

  By the definition of $AS \succeq A'$, we have $AS \vdash' x : \tau$. The last (non-trivial) step of this derivation is either (hypoth) or ($\forall$-elim$'$); we may write the derivation so that the last step is a (possibly trivial) use of ($\forall$-elim$'$). If $(x : \forall\bar{\alpha} \textbf{ with } C \,.\, \rho) \in A$, then $(x : \forall\bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}]S \,.\, \rho[\bar{\alpha} := \bar{\beta}]S) \in AS$, where $\bar{\beta}$ are the first distinct type variables not free in $\forall\bar{\alpha} \textbf{ with } C \,.\, \rho$ or in S. Hence the derivation $AS \vdash' x : \tau$ ends with

$$\frac{\begin{array}{l} (x : \forall\bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}]S \,.\, \rho[\bar{\alpha} := \bar{\beta}]S) \in AS \\ AS \vdash' C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \end{array}}{AS \vdash' x : \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]}$$

  where $\rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] = \tau$.

  We need to show that $(S_0, B_0, \tau_0) = W_o(A, x)$ succeeds and that there exists T such that

1. $S = S_0T$, except on new type variables of $W_o(A, x)$,
2. $AS \vdash B_0T$, and
3. $\tau = \tau_0 T$.

Now, $W_o(A, x)$ is defined by

> if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
> return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
> else if $(x : \forall\bar{\alpha}$ **with** $C\,.\,\tau) \in A$,
> return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
> else *fail*.

If x is overloaded in A with *lcg* $\forall\bar{\gamma}.\rho_0$, then $(S_0, B_0, \tau_0) = W_o(A, x)$ succeeds with $S_0 = [\,]$, $B_0 = \{x : \rho_0[\bar{\gamma} := \bar{\delta}]\}$, and $\tau_0 = \rho_0[\bar{\gamma} := \bar{\delta}]$, where $\bar{\delta}$ are new. Since $\forall\bar{\gamma}.\rho_0$ is the *lcg* of x, $\bar{\gamma}$ are the only variables in ρ_0 and there exist $\bar{\phi}$ such that $\rho_0[\bar{\gamma} := \bar{\phi}] = \rho$. Let T be $S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]]$. Then

$$
\begin{array}{ll}
 & \tau_0 T \\
= & \quad \ll \text{definition} \gg \\
 & \rho_0[\bar{\gamma} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\delta} \text{ occur in } \rho_0[\bar{\gamma} := \bar{\delta}] \gg \\
 & \rho_0[\bar{\gamma} := \bar{\delta}][\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
 & \rho_0[\bar{\gamma} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
 & \rho_0[\bar{\gamma} := \bar{\phi}][\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\
= & \quad \ll \text{by above} \gg \\
 & \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\
= & \quad \ll \text{by above} \gg \\
 & \tau.
\end{array}
$$

So

1. $S_0T = S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] = S$, except on $\bar{\delta}$. That is, $S_0T = S$, except on the new type variables of $W_o(A, x)$.
2. Since $B_0T = \{x : \tau_0T\} = \{x : \tau\}$, it follows that $AS \vdash B_0T$.
3. $\tau_0T = \tau$.

If x is not overloaded in A, then $(S_0, B_0, \tau_0) = W_o(A, x)$ succeeds with $S_0 = [\,]$, $B_0 = C[\bar{\alpha} := \bar{\delta}]$, and $\tau_0 = \rho[\bar{\alpha} := \bar{\delta}]$, where $\bar{\delta}$ are new. Observe that $[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}])$ and $[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ agree on C and on ρ.

> (The only variables in C or ρ are the $\bar{\alpha}$ and variables ε not among $\bar{\beta}$ or $\bar{\delta}$. Both substitutions map $\alpha_i \mapsto \pi_i$ and $\varepsilon \mapsto \varepsilon S$.)

Let T be $S \oplus [\bar{\delta} := \bar{\pi}]$. Then

1. $S_0T = S \oplus [\bar{\delta} := \bar{\pi}] = S$, except on $\bar{\delta}$. That is, $S_0T = S$, except on the new type variables of $W_o(A, x)$.
2. Also,
$$\begin{array}{cl} & B_0T \\ = & \quad \ll \text{definition} \gg \\ & C[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) \\ = & \quad \ll \text{by above observation} \gg \\ & C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]. \end{array}$$
Hence $AS \vdash B_0T$.
3. Finally,
$$\begin{array}{cl} & \tau_0T \\ = & \quad \ll \text{definition} \gg \\ & \rho[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) \\ = & \quad \ll \text{by above observation} \gg \\ & \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\ = & \\ & \tau. \end{array}$$

- <u>e is $\lambda x.e'$</u>

The last (non-trivial) step of the derivation of $A' \vdash' e : \tau$ must be ($\rightarrow$-intro):
$$\frac{A' \cup \{x : \tau'\} \vdash' e' : \tau''}{A' \vdash' \lambda x.e' : \tau' \rightarrow \tau''}$$
where x does not occur in A'.

We must show that $(S_0, B_0, \tau_0) = W_o(A, \lambda x.e')$ succeeds and that there exists T such that

1. $S = S_0T$, except on new type variables of $W_o(A, \lambda x.e')$,
2. $AS \vdash B_0T$, and
3. $\tau' \rightarrow \tau'' = \tau_0T$.

Now, $W_o(A, \lambda x.e')$ is defined by

> if x occurs in A, *fail.*
> let $(S_1, B_1, \tau_1) = W_o(A \cup \{x : \alpha\}, e')$ where α is new;
> return $(S_1, B_1, \alpha S_1 \rightarrow \tau_1)$.

Since x does not occur in A' and (by $AS \succeq A'$) $id(AS) \subseteq id(A')$, it follows that x does not occur in AS. This implies that x does not occur in A, so the first line does not fail.

Now we wish to use induction to show that the recursive call succeeds. The new type variable α is not free in A, so

$$AS \cup \{x : \tau'\} = (A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']).$$

By Lemma 2.16, $A' \cup \{x : \tau'\}$ has satisfiable constraints. Next we argue that $AS \cup \{x : \tau'\} \succeq A' \cup \{x : \tau'\}$. Clearly, $id(AS \cup \{x : \tau'\}) \subseteq id(A' \cup \{x : \tau'\})$. Now suppose that $A' \cup \{x : \tau'\} \vdash y : \rho$. If $y \neq x$, then by Corollary 2.10, $A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Corollary 2.11, $AS \cup \{x : \tau'\} \vdash y : \rho$. On the other hand, if $y = x$, then the derivation $A' \cup \{x : \tau'\} \vdash' y : \rho$ must be by (hypoth)—hence $\rho = \tau'$. So by (hypoth), $AS \cup \{x : \tau'\} \vdash y : \rho$. We have shown that

- $A' \cup \{x : \tau'\} \vdash e' : \tau''$,
- $A' \cup \{x : \tau'\}$ has satisfiable constraints, and
- $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \succeq A' \cup \{x : \tau'\}$.

So by induction, $(S_1, B_1, \tau_1) = W_o(A \cup \{x : \alpha\}, e')$ succeeds and there exists T_1 such that

1. $S \oplus [\alpha := \tau'] = S_1 T_1$, except on new variables of $W_o(A \cup \{x : \alpha\}, e')$,
2. $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \vdash B_1 T_1$, and
3. $\tau'' = \tau_1 T_1$.

So $(S_0, B_0, \tau_0) = W_o(A, \lambda x.e')$ succeeds with $S_0 = S_1$, $B_0 = B_1$, and $\tau_0 = \alpha S_1 \rightarrow \tau_1$.

Let T be T_1. Then

1. Observe that

$$\begin{array}{ll} & S_0 T \\ = & \quad \ll \text{definition} \gg \\ & S_1 T_1 \\ = & \quad \ll \text{by part 1 of inductive hypothesis above} \gg \\ & S \oplus [\alpha := \tau'],\ \text{except on new variables of } W_o(A \cup \{x : \alpha\}, e') \\ = & \quad \ll \text{definition of } \oplus \gg \\ & S,\ \text{except on } \alpha. \end{array}$$

   Hence $S_0 T = S$, except on the new type variables of $W_o(A \cup \{x : \alpha\}, e')$ and on α. That is, $S_0 T = S$, except on the new type variables of $W_o(A, \lambda x.e')$.

2. $B_0T = B_1T_1$ and we have $AS \cup \{x : \tau'\} \vdash B_1T_1$. By Theorem 2.18, every identifier occurring in B_1 is overloaded in $A \cup \{x : \alpha\}$ or occurs in some constraint of some assumption in $A \cup \{x : \alpha\}$. Since x does not occur in A and α has no constraints, this means that x does not occur in B_1. Hence Corollary 2.10 may be applied to each member of B_1T_1, yielding
$$AS \vdash B_1T_1.$$

3. Finally,

$$\begin{array}{ll} & \tau_0 T \\ = & \quad \ll \text{definition} \gg \\ & (\alpha S_1 \rightarrow \tau_1)T_1 \\ = & \quad \ll \text{by part 3 of the use of induction above} \gg \\ & \alpha S_1 T_1 \rightarrow \tau'' \\ = & \quad \ll \alpha \text{ is not a new variable of } W_o(A \cup \{x : \alpha\}, e') \gg \\ & \alpha(S \oplus [\alpha := \tau']) \rightarrow \tau'' \\ = & \quad \ll \text{definition of } \oplus \gg \\ & \tau' \rightarrow \tau''. \end{array}$$

- $\underline{e \text{ is } e'e''}$

The last (non-trivial) step of the derivation of $A' \vdash' e : \tau$ must be ($\rightarrow$-elim):

$$\frac{\begin{array}{c} A' \vdash' e' : \tau' \rightarrow \tau \\ A' \vdash' e'' : \tau' \end{array}}{A' \vdash' e' e'' : \tau}$$

We need to show that $(S_0, B_0, \tau_0) = W_o(A, e'e'')$ succeeds and that there exists T such that

1. $S = S_0T$, except on new type variables of $W_o(A, e'e'')$,
2. $AS \vdash B_0T$, and
3. $\tau = \tau_0 T$.

Now, $W_o(A, e'e'')$ is defined by

$$\begin{array}{l} \text{let } (S_1, B_1, \tau_1) = W_o(A, e'); \\ \text{let } (S_2, B_2, \tau_2) = W_o(AS_1, e''); \\ \text{let } S_3 = \mathit{unify}(\tau_1 S_2, \tau_2 \rightarrow \alpha) \text{ where } \alpha \text{ is new;} \\ \text{return } (S_1S_2S_3, B_1S_2S_3 \cup B_2S_3, \alpha S_3). \end{array}$$

By induction, $(S_1, B_1, \tau_1) = W_o(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1T_1$, except on new type variables of $W_o(A, e')$,

2. $AS \vdash B_1T_1$, and
3. $\tau' \to \tau = \tau_1 T_1$.

Now $AS = A(S_1T_1) = (AS_1)T_1$, as the new type variables of $W_o(A, e')$ do not occur free in A. So by induction, $(S_2, B_2, \tau_2) = W_o(AS_1, e'')$ succeeds and there exists T_2 such that

1. $T_1 = S_2T_2$, except on new type variables of $W_o(AS_1, e'')$,
2. $(AS_1)T_1 \vdash B_2T_2$, and
3. $\tau' = \tau_2 T_2$.

By Lemma 2.20, the new variable α does not occur in A, S_1, B_1, τ_1, S_2, B_2, or τ_2. So consider $T_2 \oplus [\alpha := \tau]$:

$$
\begin{array}{ll}
 & (\tau_1 S_2)(T_2 \oplus [\alpha := \tau]) \\
= & \qquad \ll \alpha \text{ does not occur in } \tau_1 S_2 \gg \\
 & \tau_1 S_2 T_2 \\
= & \qquad \ll \text{no new type variable of } W_o(AS_1, e'') \text{ occurs in } \tau_1 \gg \\
 & \tau_1 T_1 \\
= & \qquad \ll \text{by part 3 of first use of induction above} \gg \\
 & \tau' \to \tau
\end{array}
$$

In addition,

$$
\begin{array}{ll}
 & (\tau_2 \to \alpha)(T_2 \oplus [\alpha := \tau]) \\
= & \qquad \ll \alpha \text{ does not occur in } \tau_2 \gg \\
 & \tau_2 T_2 \to \tau \\
= & \qquad \ll \text{by part 3 of second use of induction above} \gg \\
 & \tau' \to \tau
\end{array}
$$

Hence by Robinson's unification theorem [Rob65], $S_3 = \mathit{unify}(\tau_1 S_2, \tau_2 \to \alpha)$ succeeds and there exists T_3 such that

$$T_2 \oplus [\alpha := \tau] = S_3 T_3.$$

So $(S_0, B_0, \tau_0) = W_o(A, e'e'')$ succeeds with $S_0 = S_1S_2S_3$, $B_0 = B_1S_2S_3 \cup B_2S_3$, and $\tau_0 = \alpha S_3$.

Let T be T_3. Then

1. Observe that

$$
\begin{array}{ll}
& S_0T \\
= & \quad \ll \text{definition} \gg \\
& S_1S_2S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
& S_1S_2(T_2 \oplus [\alpha := \tau]) \\
= & \quad \ll \alpha \text{ does not occur in } S_1S_2 \gg \\
& S_1S_2T_2, \text{ except on } \alpha \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
& \quad \ll \text{new variables of } W_o(AS_1, e'') \text{ don't occur in } S_1 \gg \\
& S_1T_1, \text{ except on the new type variables of } W_o(AS_1, e'') \\
= & \quad \ll \text{by part 1 of the first use of induction above} \gg \\
& S, \text{ except on the new type variables of } W_o(A, e').
\end{array}
$$

Hence we see that $S_0T = S$ except on α, on the new type variables of $W_o(AS_1, e'')$, and on the new type variables of $W_o(A, e')$. That is, $S_0T = S$ except on the new type variables of $W_o(A, e'e'')$.

2. Next,

$$
\begin{array}{ll}
& B_0T \\
= & \quad \ll \text{definition} \gg \\
& B_1S_2S_3T_3 \cup B_2S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
& B_1S_2(T_2 \oplus [\alpha := \tau]) \cup B_2(T_2 \oplus [\alpha := \tau]) \\
= & \quad \ll \alpha \text{ does not occur in } B_1S_2 \text{ or in } B_2 \gg \\
& B_1S_2T_2 \cup B_2T_2 \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
& \quad \ll \text{new variables of } W_o(AS_1, e'') \text{ don't occur in } B_1 \gg \\
& B_1T_1 \cup B_2T_2
\end{array}
$$

By part 2 of the first and second uses of induction above, $AS \vdash B_1T_1 \cup B_2T_2$, and so $AS \vdash B_0T$.

3. Finally,

$$
\begin{array}{ll}
& \tau_0T \\
= & \quad \ll \text{definition} \gg \\
& \alpha S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
& \alpha(T_2 \oplus [\alpha := \tau]) \\
= & \quad \ll \text{definition of } \oplus \gg \\
& \tau
\end{array}
$$

- <u>e is **let** $x = e'$ **in** e''</u>

The last (non-trivial) step of the derivation of $A' \vdash' e : \tau$ must be (let):

$$\frac{\begin{array}{l} A' \vdash' e' : \sigma \\ A' \cup \{x : \sigma\} \vdash' e'' : \tau \end{array}}{A' \vdash' \textbf{let } x = e' \textbf{ in } e'' : \tau}$$

where x does not occur in A'.

We need to show that $(S_0, B_0, \tau_0) = W_o(A, \textbf{let } x = e' \textbf{ in } e'')$ succeeds and that there exists T such that

1. $S = S_0 T$, except on new type variables of $W_o(A, \textbf{let } x = e' \textbf{ in } e'')$,
2. $AS \vdash B_0 T$, and
3. $\tau = \tau_0 T$.

Now, $W_o(A, \textbf{let } x = e' \textbf{ in } e'')$ is defined by

> if x occurs in A, *fail.*
> let $(S_1, B_1, \tau_1) = W_o(A, e')$;
> let $(B_1', \sigma_1) = close(AS_1, B_1, \tau_1)$;
> let $(S_2, B_2, \tau_2) = W_o(AS_1 \cup \{x : \sigma_1\}, e'')$;
> return $(S_1 S_2, B_1' S_2 \cup B_2, \tau_2)$.

Since $AS \succeq A'$, x does not occur in AS or in A. So the first line does not fail.

Since A' has satisfiable constraints, by Lemma 2.15 there exists τ' such that $A' \vdash e' : \tau'$. Hence by induction, $(S_1, B_1, \tau_1) = W_o(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1 T_1$, except on new type variables of $W_o(A, e')$,
2. $AS \vdash B_1 T_1$, and
3. $\tau' = \tau_1 T_1$.

By 1 and 2 above, we have $(AS_1)T_1 \vdash B_1 T_1$, so by Lemma 2.19 it follows that $(B_1', \sigma_1) = close(AS_1, B_1, \tau_1)$ succeeds and σ_1 is of the form $\forall \bar{\alpha}$ **with** $B_1'' . \tau_1$. Also $fv(\sigma_1) \subseteq fv(AS_1)$ and $B_1' \cup B_1'' \subseteq B_1$.

Now, in order to apply the induction hypothesis to the second recursive call we need to show that

$$AS \cup \{x : \sigma_1 T_1\} \succeq A' \cup \{x : \sigma\}.$$

There are two parts to this:

1. $A' \cup \{x : \sigma\} \vdash y : \rho$ implies $AS \cup \{x : \sigma_1 T_1\} \vdash y : \rho$, and

2. $id(AS \cup \{x : \sigma_1 T_1\}) \subseteq id(A' \cup \{x : \sigma\})$.

The second part is quite simple; we show it first. By Lemma 2.17 and Theorem 2.18, $id(\sigma_1) \subseteq id(B_1) \subseteq id(A)$. Hence

$$\begin{array}{ll} & id(AS \cup \{x : \sigma_1 T_1\}) \\ = & \quad \ll \text{by above} \gg \\ & id(AS) \cup \{x\} \\ \subseteq & \quad \ll AS \succeq A' \gg \\ & id(A') \cup \{x\} \\ \subseteq & \\ & id(A' \cup \{x : \sigma\}), \end{array}$$

showing 2.

Now we show 1. Suppose that $A' \cup \{x : \sigma\} \vdash y : \rho$. If $y \neq x$, then by Corollary 2.10, $A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Corollary 2.11, $AS \cup \{x : \sigma_1 T_1\} \vdash y : \rho$.

If, on the other hand, $y = x$, then our argument will begin by establishing that $A' \vdash e' : \rho$. By Theorem 2.9 we have $A' \cup \{x : \sigma\} \vdash' x : \rho$ and the derivation ends with (hypoth) or ($\forall$-elim$'$). If the derivation ends with (hypoth), then $\sigma = \rho$ and so $A' \vdash e' : \rho$. If the derivation ends with ($\forall$-elim$'$), then σ is of the form $\forall\bar{\beta}$ **with** $C \,.\, \rho'$ and the derivation is of the form

$$\frac{\begin{array}{l} x : \forall\bar{\beta} \textbf{ with } C \,.\, \rho' \;\epsilon\; A' \cup \{x : \sigma\} \\ A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}] \end{array}}{A' \cup \{x : \sigma\} \vdash' x : \rho'[\bar{\beta} := \bar{\pi}]}$$

where $\rho = \rho'[\bar{\beta} := \bar{\pi}]$. It is evident that x does not occur in C (if x occurs in C, then the derivation of $A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}]$ will be infinitely high), so by Corollary 2.10 applied to each member of $C[\bar{\beta} := \bar{\pi}]$, it follows that $A' \vdash C[\bar{\beta} := \bar{\pi}]$. Now the derivation $A' \vdash e' : \forall\bar{\beta}$ **with** $C \,.\, \rho'$ may be extended using ($\forall$-elim):

$$\frac{\begin{array}{l} A' \vdash e' : \forall\bar{\beta} \textbf{ with } C \,.\, \rho' \\ A' \vdash C[\bar{\beta} := \bar{\pi}] \end{array}}{A' \vdash e' : \rho'[\bar{\beta} := \bar{\pi}],}$$

giving that $A' \vdash e' : \rho$.

So by induction there exists a substitution T_2 such that

1. $S = S_1 T_2$, except on new type variables of $W_o(A, e')$,
2. $AS \vdash B_1 T_2$, and
3. $\rho = \tau_1 T_2$.

The new type variables of $W_o(A, e')$ are not free in A, so $AS_1T_1 = AS = AS_1T_2$. Hence T_1 and T_2 are equal when restricted to the free variables of AS_1. Since, by Lemma 2.19, $fv(\sigma_1) \subseteq fv(AS_1)$, it follows that $\sigma_1 T_1 = \sigma_1 T_2$. Now, σ_1 is of the form $\forall \bar{\alpha}$ **with** $B_1'' . \tau_1$, so $\sigma_1 T_1$ $(= \sigma_1 T_2)$ is of the form

$$\forall \bar{\gamma} \textbf{ with } B_1''[\bar{\alpha} := \bar{\gamma}]T_2 . \tau_1[\bar{\alpha} := \bar{\gamma}]T_2,$$

where $\bar{\gamma}$ are the first distinct type variables not free in T_2 or σ_1.

Consider the substitution $[\bar{\gamma} := \bar{\alpha}T_2]$. We have

$$B_1''[\bar{\alpha} := \bar{\gamma}]T_2[\bar{\gamma} := \bar{\alpha}T_2] = B_1''T_2$$

and

$$\tau_1[\bar{\alpha} := \bar{\gamma}]T_2[\bar{\gamma} := \bar{\alpha}T_2] = \tau_1 T_2.$$

By 2 above, we have $AS \vdash B_1 T_2$. Since by Lemma 2.19 we have $B_1'' \subseteq B_1$, it follows that $AS \vdash B_1''T_2$. By Corollary 2.11, then, we have $AS \cup \{x : \sigma_1 T_1\} \vdash B_1''T_2$. Hence we have the following derivation:

$$AS \cup \{x : \sigma_1 T_1\} \vdash x : \forall \bar{\gamma} \textbf{ with } B_1''[\bar{\alpha} := \bar{\gamma}]T_2 . \tau_1[\bar{\alpha} := \bar{\gamma}]T_2$$

by (hypoth) since $\sigma_1 T_1 = \forall \bar{\gamma}$ **with** $B_1''[\bar{\alpha} := \bar{\gamma}]T_2 . \tau_1[\bar{\alpha} := \bar{\gamma}]T_2$,

$$AS \cup \{x : \sigma_1 T_1\} \vdash B_1''[\bar{\alpha} := \bar{\gamma}]T_2[\bar{\gamma} := \bar{\alpha}T_2]$$

by above since $B_1''[\bar{\alpha} := \bar{\gamma}]T_2[\bar{\gamma} := \bar{\alpha}T_2] = B_1''T_2$, and hence

$$AS \cup \{x : \sigma_1 T_1\} \vdash x : \tau_1 T_2,$$

by ($\forall$-elim) since $\tau_1[\bar{\alpha} := \bar{\gamma}]T_2[\bar{\gamma} := \bar{\alpha}T_2] = \tau_1 T_2$. Since by 3 above, $\tau_1 T_2 = \rho$, we have at last that

$$AS \cup \{x : \sigma_1 T_1\} \vdash x : \rho.$$

This completes that proof that $AS \cup \{x : \sigma_1 T_1\} \succeq A' \cup \{x : \sigma\}$, which is the same as

$$(AS_1 \cup \{x : \sigma_1\})T_1 \succeq A' \cup \{x : \sigma\}.$$

Because $A' \vdash e' : \sigma$ and A' has satisfiable constraints, it is easy to see that the constraints of σ are satisfiable with respect to A'. Hence by Lemma 2.16, $A' \cup \{x : \sigma\}$ has satisfiable constraints.

This allows us to apply the induction hypothesis again, which shows that $(S_2, B_2, \tau_2) = W_o(AS_1 \cup \{x : \sigma_1\}, e'')$ succeeds and that there exists T_3 such that

1. $T_1 = S_2 T_3$, except on the new type variables of $W_o(AS_1 \cup \{x : \sigma_1\}, e'')$,

2. $AS \cup \{x : \sigma_1 T_1\} \vdash B_2 T_3$, and
3. $\tau = \tau_2 T_3$.

So $(S_0, B_0, \tau_0) = W_o(A, \textbf{let } x = e' \textbf{ in } e'')$ succeeds with $S_0 = S_1 S_2$, $B_0 = B_1' S_2 \cup B_2$, and $\tau_0 = \tau_2$.

Let T be T_3. Then

1. Observe that

$$\begin{array}{ll} & S_0 T \\ = & \quad \ll \text{definition} \gg \\ & S_1 S_2 T_3 \\ = & \quad \ll \text{part 1 of third use of induction; the new variables} \gg \\ & \quad \ll \text{of } W_o(AS_1 \cup \{x : \sigma_1\}, e'') \text{ don't occur in } S_1 \gg \\ & S_1 T_1, \text{ except on new type variables of } W_o(AS_1 \cup \{x : \sigma_1\}, e'') \\ = & \quad \ll \text{by part 1 of first use of induction above} \gg \\ & S, \text{ except on new type variables of } W_o(A, e'). \end{array}$$

   So $S_0 T = S$, except on new variables of $W_o(A, \textbf{let } x = e' \textbf{ in } e'')$.

2. Next,

$$\begin{array}{ll} & B_0 T \\ = & \quad \ll \text{definition} \gg \\ & B_1' S_2 T_3 \cup B_2 T_3 \\ \subseteq & \quad \ll \text{Lemma 2.19} \gg \\ & B_1 S_2 T_3 \cup B_2 T_3 \\ = & \quad \ll \text{part 1 of third use of induction; the new variables} \gg \\ & \quad \ll \text{of } W_o(AS_1 \cup \{x : \sigma_1\}, e'') \text{ don't occur in } B_1 \gg \\ & B_1 T_1 \cup B_2 T_3 \end{array}$$

   Now by part 2 of the first use of induction above, $AS \vdash B_1 T_1$. And by part 2 of the third use of induction above, $AS \cup \{x : \sigma_1 T_1\} \vdash B_2 T_3$. By Theorem 2.18, every identifier occurring in B_1 is overloaded in A or occurs in some constraint of some assumption in A. Since x does not occur in A, it follows that x does not occur in B_1 either. Now by Lemma 2.19, the constraints of σ_1 are a subset of B_1. Hence x does not occur in the constraints of σ_1. By Theorem 2.18, every identifier occurring in B_2 is overloaded in $AS_1 \cup \{x : \sigma_1\}$ or occurs in some constraint of some assumption in $AS_1 \cup \{x : \sigma_1\}$. Since x does not occur in AS_1 or in the constraints of σ_1, it follows that x does not occur in B_2. Hence Corollary 2.10 may be applied to each member of $B_2 T_3$, yielding $AS \vdash B_2 T_3$. So $AS \vdash B_0 T$.

3. Finally,

$$\begin{array}{ll} & \tau_0 T \\ = & \qquad \ll \text{definition} \gg \\ & \tau_2 T_3 \\ = & \qquad \ll \text{by part 3 of the third use of induction above} \gg \\ & \tau. \end{array}$$

QED

At long last, we get the completeness of W_o as a simple corollary:

Corollary 2.22 *If* $AS \vdash e : \tau$ *and* AS *has satisfiable constraints, then* $(S_0, B_0, \tau_0) = W_o(A, e)$ *succeeds and there exists a substitution* T *such that*

1. $S = S_0 T$ *, except on new type variables of* $W_o(A, e)$ *,*
2. $AS \vdash B_0 T$ *, and*
3. $\tau = \tau_0 T$ *.*

Proof: Follows immediately from Theorem 2.21 by letting A' be AS, and noting that $\succeq$ is reflexive. **QED**

Also, we can now prove the stubborn $\succeq$-lemma:

Corollary 2.23 *If* $A' \vdash e : \tau$ *,* A' *has satisfiable constraints, and* $A \succeq A'$ *, then* $A \vdash e : \tau$ *.*

Proof: Letting S be $[\,]$, we see from Theorem 2.21 that $(S_0, B_0, \tau_0) = W_o(A, e)$ succeeds and there exists a substitution T such that

1. $[\,] = S_0 T$, except on new type variables of $W_o(A, e)$,
2. $A \vdash B_0 T$, and
3. $\tau = \tau_0 T$.

Now by Theorem 2.18, $AS_0 \cup B_0 \vdash e : \tau_0$, so by Lemma 2.4, $AS_0 T \cup B_0 T \vdash e : \tau_0 T$. Using 1 and 3 above, this gives us $A \cup B_0 T \vdash e : \tau$. Since by 2 above $A \vdash B_0 T$, we may use Corollary 2.12 to show that $A \vdash e : \tau$. **QED**

Finally, we obtain as a corollary the following principal typing result:

Corollary 2.24 *Let* A *be an assumption set with satisfiable constraints and no free type variables. If* e *is well typed with respect to* A*, then* $(S, B, \tau) = W_o(A, e)$ *succeeds,* $(B', \sigma) = close(A, B, \tau)$ *succeeds, and the typing* $A \vdash e : \sigma$ *is principal.*

Proof: Since e is well typed with respect to A and since A has satisfiable constraints, by Lemma 2.15 there exists π such that $A \vdash e : \pi$. Hence we see from Corollary 2.22 that $(S, B, \tau) = W_o(A, e)$ succeeds and there exists T such that

1. $[\,] = ST$, except on the new type variables of $W_o(A,e)$,

2. $A \vdash BT$, and

3. $\pi = \tau T$.

Since $AT \vdash BT$, by Lemma 2.19 $(B', \sigma) = close(A, B, \tau)$ succeeds, σ is of the form $\forall \bar{\alpha}$ **with** $B''.\tau$, and

- $B' = \{\}$,
- σ has no free type variables, and
- $B'' \subseteq B$.

By Theorem 2.18 and Lemma 2.17, we have $A \vdash e : \sigma$.

Now let σ' be an arbitrary type scheme such that $A \vdash e : \sigma'$. We need to show that $\sigma \geq_A \sigma'$. Suppose that $\sigma' \geq_A \rho$. Then by the definition of $\geq_A$ it follows that ($\forall$-elim) may be used to derive $A \vdash e : \rho$. So by Corollary 2.22 again, there exists T' such that

1. $[\,] = ST'$, except on the new type variables of $W_o(A,e)$,

2. $A \vdash BT'$, and

3. $\rho = \tau T'$.

Since $B'' \subseteq B$, we have $A \vdash B''T'$. Notice also that all of the variables occurring in B'' or in τ are among the $\bar{\alpha}$, so by restricting T' to the $\bar{\alpha}$ we obtain a substitution $[\bar{\alpha} := \bar{\phi}]$ such that $A \vdash B''[\bar{\alpha} := \bar{\phi}]$ and $\rho = \tau[\bar{\alpha} := \bar{\phi}]$. This shows that $\forall \bar{\alpha}$ **with** $B''.\tau \geq_A \rho$, and hence that $\forall \bar{\alpha}$ **with** $B''.\tau \geq_A \sigma'$. So $A \vdash e : \forall \bar{\alpha}$ **with** $B''.\tau$ is principal. **QED**

2.5 Limitative Results

Recall that in our definition of $close(A, B, \tau)$ in Section 2.3, we made use of a function $satisfiable(B, A)$ without indicating how it might be computed. We address that omission in this section.

Given a set B of τ-assumptions and an assumption set A, $satisfiable(B, A)$ returns *true* if B is satisfiable with respect to A (that is, if there exists a substitution S such that $A \vdash BS$) and returns *false* otherwise. Unfortunately, the satisfiability problem turns out to be undecidable.

Theorem 2.25 *Given an assumption set A and a set B of τ-assumptions, it is undecidable whether B is satisfiable with respect to A.*

Proof: It is easy to reduce the Post Correspondence Problem (PCP) [HU79] to the satisfiability problem. We present the reduction by means of an example. Suppose we are given the PCP instance

$$\begin{array}{ll} x_1 = 10 & y_1 = 101 \\ x_2 = 011 & y_2 = 11 \\ x_3 = 110 & y_3 = \varepsilon, \end{array}$$

where ε denotes the empty string. We will use type constants 0, 1, and \$ and a right-associative binary type constructor $\rightarrow$. Define

$$A_{pcp} = \left\{ \begin{array}{l} p : (1 \rightarrow 0) \rightarrow (1 \rightarrow 0 \rightarrow 1), \\ p : (0 \rightarrow 1 \rightarrow 1) \rightarrow (1 \rightarrow 1), \\ p : (1 \rightarrow 1 \rightarrow 0 \rightarrow \$) \rightarrow \$, \\ p : \forall \alpha, \beta \textbf{ with } p : \alpha \rightarrow \beta \,.\, (1 \rightarrow 0 \rightarrow \alpha) \rightarrow (1 \rightarrow 0 \rightarrow 1 \rightarrow \beta), \\ p : \forall \alpha, \beta \textbf{ with } p : \alpha \rightarrow \beta \,.\, (0 \rightarrow 1 \rightarrow 1 \rightarrow \alpha) \rightarrow (1 \rightarrow 1 \rightarrow \beta), \\ p : \forall \alpha, \beta \textbf{ with } p : \alpha \rightarrow \beta \,.\, (1 \rightarrow 1 \rightarrow 0 \rightarrow \alpha) \rightarrow \beta \end{array} \right\}$$

Now consider whether $\{p : \alpha \rightarrow \alpha\}$ is satisfiable with respect to A_{pcp}.

This will be true if and only if there is some τ such that $A_{pcp} \vdash p : \tau \rightarrow \tau$. But A_{pcp} gives p precisely all types of the form $\pi \rightarrow \pi'$, where π is obtained by concatenating various x_i's and π' is obtained by concatenating the corresponding y_i's.[9] Hence $\{p : \alpha \rightarrow \alpha\}$ is satisfiable with respect to A_{pcp} precisely when the given PCP instance has a solution. **QED**

It is conceivable that there is some way to do type inference without using function *satisfiable*(B, A), but such hopes are quickly dashed by the following corollary.

Corollary 2.26 *Given an assumption set A and an expression e, it is undecidable whether e is well typed with respect to A.*

Proof: Again we use a reduction from PCP. Define A_{pcp} as above, and consider whether the expression $(\lambda f.\lambda x.f(f\ x))\,p$ is well typed with respect to A_{pcp}.

Now, $\lambda f.\lambda x.f(f\ x)$ has principal type $\forall \alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. Hence $(\lambda f.\lambda x.f(f\ x))\,p$ is well typed with respect to A_{pcp} if and only if p has some type of the form $\tau \rightarrow \tau$, which in turn holds if and only if the given PCP instance has a solution.[10] **QED**

Another easy corollary is that the instance relation $\geq_A$ is undecidable.

---

[9] Possibly π and π' have a \$ tacked on the end.

[10] We remark in passing that the expression $(\lambda f.\lambda x.f(f\,x))p$ cannot be replaced by the simpler expression $\lambda x.p(p\,x)$ in this proof. The problem with the simpler expression is that the two occurrences of p are not forced to have the same type.

Corollary 2.27 *Given A, σ, and τ, it is undecidable whether $\sigma \geq_A \tau$.*

Proof: Define A_{pcp} as above; then

$$\forall\alpha \textbf{ with } p : \alpha \to \alpha \,.\, bool \ \geq_{A_{pcp}} \ bool$$

if and only if the given PCP instance has a solution. **QED**

It might appear that the key to all of these undecidability results is the necessity of 'guessing': for example, to determine whether $\{p : \alpha \to \alpha\}$ is satisfiable we must guess how α should be instantiated. This suggests that if we arrange to eliminate such guessing, perhaps undecidability will be avoided.

To this end, we can define the class of *coupled* type schemes: a type scheme $\forall\bar{\alpha} \textbf{ with } C \,.\, \tau$ is said to be *coupled* if every type variable occurring in C occurs also in τ. For example, the type $\forall\alpha \textbf{ with } p : \alpha \to \alpha \,.\, bool$ used in the proof of Corollary 2.27 is uncoupled. We furthermore say that an assumption set A is *coupled* if it contains only coupled assumptions.

We also require the notion of a *nonoverlapping* assumption set. We say that type schemes σ and σ' *overlap* if (after renaming to eliminate any common bound type variables) their bodies are unifiable. Then we say that an assumption set A is *nonoverlapping* if A contains no two assumptions $x : \sigma$ and $x : \sigma'$ with σ and σ' overlapping.

Finally we say that a type τ is *ground* if it contains no occurrences of type variables, and an assumption $x : \tau$ is ground if τ is ground.

If A is coupled and nonoverlapping and B contains only ground assumptions, then no guessing is involved in verifying the satisfiability of B with respect to A—since B contains only ground assumptions, B is satisfiable if and only if $A \vdash B$, and we can verify $A \vdash B$ by "working backwards" in a completely deterministic manner. For example, if

$$A = \{c : char, c : \forall\alpha \textbf{ with } c : \alpha \,.\, seq(\alpha)\,\}$$

then we can reason

$$\begin{array}{c} A \vdash \{c : seq(seq(char))\} \\ \Updownarrow \\ A \vdash \{c : seq(char)\} \\ \Updownarrow \\ A \vdash \{c : char\}. \end{array}$$

More formally, we can verify that $A \vdash B$ as follows:

> **while** $B \neq \{\}$ **do**
> choose $x : \tau$ in B;
> find an assumption $x : \forall \bar{\alpha}$ **with** $C . \rho$ in A
> such that for some S, $\rho S = \tau$;
> if none, then answer 'no';
> $B := (B - \{x : \tau\}) \cup CS$;
> **od**;
> answer 'yes'.

Because of the restriction to nonoverlapping and coupled assumptions, there is always a unique assumption to apply next, and B always contains only ground assumptions. Notice, however, that if $A = \{c : \forall \alpha$ **with** $c : seq(\alpha) . \alpha\}$, then an infinite branch will be generated when we attempt to verify that $A \vdash \{c : char\}$, so our procedure is not guaranteed to terminate. One might imagine that such infinite branches can somehow be detected and avoided, but, surprisingly, this is not so.

Theorem 2.28 *Given a coupled, nonoverlapping assumption set A and a ground assumption $x : \tau$, it is undecidable whether $A \vdash x : \tau$.*

Proof: We reduce the problem of determining whether a deterministic Turing Machine halts on empty input [HU79] to this problem. Suppose we are given a Turing Machine $M = (Q, \{0,1\}, \{0,1,B\}, \delta, q_0, B, \{q_H\})$, where

- Q is the set of states,
- $\{0,1\}$ is the input alphabet,
- $\{0,1,B\}$ is the tape alphabet,
- $\delta : Q \times \{0,1,B\} \rightarrow Q \times \{0,1,B\} \times \{L,R\}$,
- q_0 is the start state,
- B is the blank symbol,
- q_H is the unique halting state.

The idea is to encode instantaneous descriptions (IDs) of M as types and to define a set of type assumptions A_{halt} so that $A_{halt} \vdash h : \rho$ if and only if ρ encodes an ID from which M eventually halts. We need just one identifier h, a binary type constructor $\rightarrow$, and a set of type constants $Q \cup \{0,1,B,\varepsilon\}$.

We encode IDs as follows: $01Bq10$ is encoded as

$$(B \rightarrow 1 \rightarrow 0 \rightarrow \varepsilon) \rightarrow q \rightarrow (1 \rightarrow 0 \rightarrow \varepsilon),$$

and in general, for α_1 and α_2 in $\{0,1,B\}^*$, $\alpha_1 q \alpha_2$ is encoded as

$$\overline{\alpha_1^{\text{reverse}}} \to q \to \overline{\alpha_2},$$

where if $\alpha = X_1X_2\ldots X_n$, $n \geq 0$, then $\overline{\alpha} = X_1 \to X_2 \to \cdots \to X_n \to \varepsilon$.

Next, we observe that M eventually halts from ID if and only if either

- ID is a halting ID, or
- M eventually halts from the successor of ID.

We include assumptions in A_{halt} to deal with these two possibilities. First, to deal with halting IDs, we include

$$h : \forall \alpha, \beta . \alpha \to q_H \to \beta.$$

Next we want to express "$h : ID$ if $h :$ successor(ID)". Suppose that $\delta(q,X) = (q', Y, R)$. We model this by including

$$h : \alpha \to q \to (X \to \beta) \textbf{ if } h : (Y \to \alpha) \to q' \to \beta,$$

which in our official syntax is

$$h : \forall \alpha, \beta \textbf{ with } h : (Y \to \alpha) \to q' \to \beta \,.\, \alpha \to q \to (X \to \beta).$$

In addition, if $X = B$ then we also include

$$h : \alpha \to q \to \varepsilon \textbf{ if } h : (Y \to \alpha) \to q' \to \varepsilon.$$

Similarly, if $\delta(q,X) = (q', Y, L)$, then for each Z in the tape alphabet we include

$$h : (Z \to \alpha) \to q \to (X \to \beta) \textbf{ if } h : \alpha \to q' \to (Z \to Y \to \beta)$$

and, if $X = B$,

$$h : (Z \to \alpha) \to q \to \varepsilon \textbf{ if } h : \alpha \to q' \to (Z \to Y \to \varepsilon).$$

It follows, then, that $A_{halt} \vdash h : \varepsilon \to q_0 \to \varepsilon$ if and only if M halts on ε. Furthermore, A_{halt} is coupled and nonoverlapping. **QED**

In Section 2.6 we present a set of restrictions that *do* make the satisfiability problem decidable. First, however, we digress to mention an application of Theorem 2.28 to logic and logic programming.

2.5.1 A Logical Digression

As we remarked earlier, our type assumptions can be rephrased as Horn clauses [Llo84]. For each identifier x, we introduce a unary predicate symbol T_x with the intention that $T_x(\tau)$ means $x : \tau$. Then the type assumption

$$x : \forall\alpha \textbf{ with } x : \alpha,\ y : \alpha \rightarrow \alpha \,.\, set(\alpha)$$

is represented by the Horn clause

$$T_x(set(\alpha)) \leftarrow T_x(\alpha) \wedge T_y(\alpha \rightarrow \alpha).$$

The essential property of the translation is this: if A^* denotes the translation of A, then

$$A \vdash x : \tau$$

if and only if

$$A^* \cup \{\textbf{false} \leftarrow T_x(\tau)\} \text{ is unsatisfiable.}$$

Furthermore, our definitions of *coupled* and *nonoverlapping* can be applied in the obvious way to Horn clauses. So we have the following corollary to Theorem 2.28.

Corollary 2.29 *It is undecidable whether a set of coupled, nonoverlapping Horn clauses is satisfiable.*

This corollary has consequences for logic programming as well: the set of all logic programs that never require backtracking in their execution includes the class of all sets of coupled, nonoverlapping Horn clauses. Hence the possibility of nontermination cannot be avoided in logic programming, even in situations where no backtracking can arise.

2.6 Restricted Forms of Overloading

In this section, we impose restrictions on the forms of overloading allowed; our goal is to make the satisfiability problem decidable, which in turn will make the typability problem decidable.

We saw in Section 2.5 that our assumption sets are expressive enough to encode the Post Correspondence Problem and the Halting Problem. But the examples of overloading that we care about in practice do not require nearly so much power. Consider the assumptions for $\leq$, our typical interesting example of overloading:

$$\begin{aligned}
&\leq : char \rightarrow char \rightarrow bool,\\
&\leq : real \rightarrow real \rightarrow bool,\\
&\leq : \forall\alpha \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool \,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool\,,\\
&\leq : \forall\alpha,\beta \textbf{ with } \leq : \alpha \rightarrow \alpha \rightarrow bool,\ \leq : \beta \rightarrow \beta \rightarrow bool\,.\\
&\qquad (\alpha \times \beta) \rightarrow (\alpha \times \beta) \rightarrow bool
\end{aligned}$$

The effect of these assumptions may be stated thus: the types of $\leq$ are of the form $\tau \rightarrow \tau \rightarrow bool$; by the first rule τ may be $char$, by the second rule τ may be $real$, by the third rule τ may be $seq(\tau')$ if τ may be τ', and by the fourth rule τ may be $\tau' \times \tau''$ if τ may be τ' and τ''. That is, $\leq$ has exactly those types of the form $\tau \rightarrow \tau \rightarrow bool$ where τ is any type built from the constructors $real$, $char$, seq, and $\times$. In this situation, we can do better than our usual implicit representation of the types for $\leq$; in fact we could use the explicit representation

$$\forall \alpha_{\{char, real, seq, \times\}}.\alpha \rightarrow \alpha \rightarrow bool,$$

associating the constructor set $\{char, real, seq, \times\}$ with α to express the fact that α may only be instantiated with types built from those constructors. Furthermore, we will see that this explicit representation of the types of $\leq$ is very useful in solving the satisfiability problem. We now make these ideas precise.

Definition 2.5 *We say that x is* overloaded by constructors *in A if the lcg of x in A is of the form $\forall \alpha.\tau$ and if for every assumption $x : \forall \bar{\beta}$* **with** *$C . \rho$ in A,*

- *$\rho = \tau[\alpha := \chi(\bar{\beta})]$, for some type constructor χ, and*
- *$C = \{x : \tau[\alpha := \beta_i] \mid \beta_i \epsilon \bar{\beta}\}$.*

Furthermore, we say that the type constructor set *of x in A is*

$$\{\chi \mid A \text{ contains an assumption } x : \forall \bar{\beta} \textbf{ with } C . \tau[\alpha := \chi(\bar{\beta})]\}$$

For example, under the assumptions given above, $\leq$ is overloaded by constructors and has type constructor set $\{char, real, seq, \times\}$.

Lemma 2.30 *If x is overloaded by constructors in A with lcg $\forall \alpha.\tau$ and type constructor set X, then $A \vdash x : \rho$ if and only if ρ is of the form $\tau[\alpha := \pi]$ and π is a type built from the constructors in X.*

Proof: The 'if' direction is a straightforward induction on the structure of π.

The 'only if' direction is a straightforward induction on the length of the derivation of $A \vdash' x : \rho$. **QED**

Definition 2.6 *We say that A is an* overloading by constructors *if*

- *every identifier overloaded in A is overloaded by constructors and*
- *for every constraint $x : \rho$ in an assumption in A, x is overloaded in A and ρ is an instance of the lcg of x.*

We now argue that if A is an overloading by constructors, then the satisfiability problem can be solved efficiently. For simplicity, however, we restrict *satisfiable*(B, A) to constraint sets B generated by W_o; this is justified by the fact that *close* calls *satisfiable* only on constraint sets generated by W_o. Now, if A is an overloading by constructors, then a simple inductive argument shows that if $(S, B, \tau) = W_o(A, e)$, then B contains only constraints of the form $x : \rho$ where x is overloaded in A and ρ is an instance of the *lcg* of x. Hence we restrict our attention to constraint sets of this form.

Let B be a constraint set containing only constraints of the form $x : \rho$ where x is overloaded in A and ρ is an instance of the *lcg* of x. Let $x : \rho$ be a constraint in B. By assumption, $\rho = \tau_x[\alpha := \pi]$ where $\forall\alpha.\tau_x$ is the *lcg* of x in A, so by Lemma 2.30, $A \vdash x : \rho S$ if and only if πS is a type constructed from X_x, the type constructor set of x. Examine π; if it contains any type constructor not in X_x, then B is unsatisfiable. Otherwise, S satisfies $x : \rho$ if and only if S instantiates all variables in π to types constructed from the constructors in X_x. In this way, the constraint $x : \rho$ in B associates the constructor set X_x with each variable occurring in ρ. Now, for each variable β in B, take the intersection of all the constructor sets thus associated with β; this gives explicitly the set of possible instantiations of β that satisfy B—in particular, there is some satisfying instantiation of β if and only if the intersection contains a 0-ary type constructor. Hence, B is satisfiable if and only if every variable in B has an associated constructor set containing a 0-ary type constructor. Clearly, this procedure can be done in polynomial time.

There is a natural example of overloading for which the restriction to overloading by constructors is too severe. The example, due to Dennis Volpano, is matrix multiplication: for matrices to be multiplied it is necessary that the elements of the matrices can be multiplied and also added, for we need to form inner products. Hence we would like to use the assumption

$$\begin{aligned} * : \forall\alpha \ \textbf{with} \ + : \alpha \rightarrow \alpha \rightarrow \alpha,\ * : \alpha \rightarrow \alpha \rightarrow \alpha\,. \\ \mathit{matrix}(\alpha) \rightarrow \mathit{matrix}(\alpha) \rightarrow \mathit{matrix}(\alpha), \end{aligned}$$

but this is not an overloading by constructors.

We conjecture, however, that the satisfiability problem remains decidable when the second part of the definition of 'overloaded by constructors' is relaxed to

$$C \supseteq \{x : \tau[\alpha := \beta_i] \mid \beta_i \epsilon \bar{\beta}\},$$

which permits the above matrix multiplication assumption. Further research into the satisfiability problem is described in [VS91].

2.7 Conclusion

We have shown that overloading may be combined with Hindley-Milner-Damas polymorphism in a natural way that preserves the principal typing property and type inference. However, certain restrictions on overloading need to be imposed to avoid undecidability.

Chapter 3

Subtyping

In this chapter we develop the theory of polymorphism in the presence of both overloading and subtyping. Our new type system is an extension of the type system of Chapter 2, so this chapter will parallel the development of Chapter 2, preserving as many of the results as possible.

3.1 The Type System

One approach to subtyping is to use explicit type conversions, thereby eliminating the need to change the type system at all. For example, we can indicate that *int* is a subtype of *real* simply by defining a conversion function $c : int \rightarrow real$. The trouble with this approach is that it limits polymorphism. Consider, for example, the function *twice*, defined as $\lambda f.\lambda x.f(f\,x)$. Intuitively, the application of *twice* to a function $g : real \rightarrow int$ should be well typed. But if subtyping is achieved solely through explicit type conversion, this is not so; indeed, for the application $twice\,g$ to be well typed, we must change the definition of *twice* to $\lambda f.\lambda x.f(c(f\,x))$. But now *twice* cannot be applied to a function of type $char \rightarrow char$. In general, the use of explicit type conversions forces programs to be overspecialized.

Instead, our approach to subtyping is to augment the type system with a new kind of assertion. In addition to *typings* $e : \sigma$, we have *inclusions* $\tau \subseteq \tau'$. The inclusion $\tau \subseteq \tau'$ asserts that τ is a subtype of τ'. Inclusions are incorporated into the type system in four ways:

1. Inclusions may occur in constraint sets of type schemes.

2. Inclusions may occur in assumption sets.

3. There are new rules for proving judgements of the form $A \vdash \tau \subseteq \tau'$.

4. Finally, the new typing rule

$$(\subseteq) \qquad \frac{\begin{array}{c} A \vdash e : \tau \\ A \vdash \tau \subseteq \tau' \end{array}}{A \vdash e : \tau'}$$

links the inclusion sublogic with the typing sublogic.

3.1.1 Types and Type Schemes

Types are just as in Chapter 2; type schemes are now defined by

$$\sigma ::= \forall \alpha_1, \ldots, \alpha_n \textbf{ with } C_1, \ldots, C_m \,.\, \tau \,,$$

where each constraint C_i is either a typing $x : \rho$ or an inclusion $\rho \subseteq \rho'$.

We say that a type is *atomic* if it is a type constant (that is, a 0-ary type constructor) or a type variable.

3.1.2 Substitution and α-equivalence

A substitution S may be applied to an inclusion $\tau \subseteq \tau'$, yielding the inclusion $(\tau S) \subseteq (\tau' S)$.

All the lemmas about α-equivalence given in Section 2.1.2 carry over to our richer set of type schemes, without modification.

3.1.3 The Typing Rules

A type assumption set A may now contain inclusions $\tau \subseteq \tau'$ as well as typings $x : \sigma$. The notion of τ-assumption is extended to include inclusions $\tau \subseteq \tau'$ as well as typings of the form $x : \tau$.

All the typing rules given in Figure 2.1 are retained in the new system. In addition, there are new rules given in Figure 3.1. The first five rules allow inclusion judgements to be proved. Rule (hypoth) allows inclusion assumptions to be used; rules (reflex) and (trans) assert that $\subseteq$ is reflexive and transitive. Next, there are rules expressing the well-known monotonicities of the various type constructors [Rey80,Car84,Rey85]. For example, $\rightarrow$ is antimonotonic in its first argument and monotonic in its second argument. The name $((-) \rightarrow (+))$ compactly represents this information. Additional rules, such as $(\,set(+)\,)$, will be introduced as needed. The last rule, $(\subseteq)$, says that an expression of type τ has any supertype of τ as well.

As an example, we derive the principal typing

$$\{\} \vdash \lambda f.\lambda x.f(f\ x) : \forall \alpha, \beta \textbf{ with } \beta \subseteq \alpha \,.\, (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\,.$$

We have

$$\{\beta \subseteq \alpha,\ f : \alpha \rightarrow \beta,\ x : \alpha\} \vdash f : \alpha \rightarrow \beta \tag{3.1}$$

(hypoth)	$A \vdash \tau \subseteq \tau'$, if $(\tau \subseteq \tau') \in A$
(reflex)	$A \vdash \tau \subseteq \tau$
(trans)	$\dfrac{A \vdash \tau \subseteq \tau' \quad A \vdash \tau' \subseteq \tau''}{A \vdash \tau \subseteq \tau''}$
$((-) \rightarrow (+))$	$\dfrac{A \vdash \tau' \subseteq \tau \quad A \vdash \rho \subseteq \rho'}{A \vdash (\tau \rightarrow \rho) \subseteq (\tau' \rightarrow \rho')}$
(*seq*(+))	$\dfrac{A \vdash \tau \subseteq \tau'}{A \vdash seq(\tau) \subseteq seq(\tau')}$
$(\subseteq)$	$\dfrac{A \vdash e : \tau \quad A \vdash \tau \subseteq \tau'}{A \vdash e : \tau'}$

Figure 3.1: Additional Rules for Subtyping

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash x : \alpha \tag{3.2}$$

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash (f\ x) : \beta \tag{3.3}$$

by ($\to$ -elim) on (3.1) and (3.2),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash \beta \subseteq \alpha \tag{3.4}$$

by (hypoth),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash (f\ x) : \alpha \tag{3.5}$$

by ($\subseteq$) on (3.3) and (3.4),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta,\ x : \alpha\} \vdash f(f\ x) : \beta \tag{3.6}$$

by ($\to$ -elim) on (3.1) and (3.5),

$$\{\beta \subseteq \alpha,\ f : \alpha \to \beta\} \vdash \lambda x.f(f\ x) : \alpha \to \beta \tag{3.7}$$

by ($\to$ -intro) on (3.6),

$$\{\beta \subseteq \alpha\} \vdash \lambda f.\lambda x.f(f\ x) : (\alpha \to \beta) \to (\alpha \to \beta) \tag{3.8}$$

by ($\to$ -intro) on (3.7),

$$\{\} \vdash (\beta \subseteq \alpha)[\beta := \alpha] \tag{3.9}$$

by (reflex), and finally

$$\{\} \vdash \lambda f.\lambda x.f(f\ x) : \forall \alpha, \beta \textbf{ with } \beta \subseteq \alpha \,.\, (\alpha \to \beta) \to (\alpha \to \beta) \tag{3.10}$$

by ($\forall$ -intro) on (3.8) and (3.9).

Recall that, without subtyping, the principal type of $\lambda f.\lambda x.f(f\ x)$ is simply $\forall \alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$. This illustrates that adding subtyping to a type system has more pervasive effects than does adding overloading—the principal type of a program that uses no overloaded identifiers is unaffected by the presence of overloading, but the principal type of *every* program is potentially changed by the presence of subtyping.

3.1.4 The Instance Relation

The instance relation $\geq_A$ needs to be refined to account for subtyping. What we want, essentially, is that $\sigma \geq_A \tau$ if and only if $A \cup \{x : \sigma\} \vdash x : \tau$, where x is any identifier not occurring in A.

Definition 3.1 ($\forall \bar{\alpha}$ **with** $C \,.\, \tau$) $\geq_A \tau'$ *if there is a substitution* $[\bar{\alpha} := \bar{\pi}]$ *such that*

- $A \vdash C[\bar{\alpha} := \bar{\pi}]$ *and*
- $A \vdash \tau[\bar{\alpha} := \bar{\pi}] \subseteq \tau'$.

3.2 Properties of the Type System

In this section we show that the properties of the type system of Chapter 2 are not disturbed by the addition of subtyping.

We begin by showing a number of useful properties of the inclusion sublogic; fortunately, inclusions have simpler behavior than do typings, so the proofs are easier.

Lemma 3.1 *If $A \vdash \tau \subseteq \tau'$, then $AS \vdash \tau S \subseteq \tau' S$. Furthermore, the derivation of $AS \vdash \tau S \subseteq \tau' S$ uses the same sequence of steps as the derivation of $A \vdash \tau \subseteq \tau'$.*

Proof: By induction on the length of the derivation of $A \vdash \tau \subseteq \tau'$. Consider the last step of the derivation:

(hypoth) The derivation is simply $A \vdash \tau \subseteq \tau'$ where $(\tau \subseteq \tau') \in A$. So $(\tau S \subseteq \tau' S) \in AS$, and $AS \vdash \tau S \subseteq \tau' S$ by (hypoth).

(reflex) By (reflex), $AS \vdash \tau S \subseteq \tau S$.

(trans) The derivation ends with

$$\frac{\begin{array}{l} A \vdash \tau \subseteq \tau' \\ A \vdash \tau' \subseteq \tau'' \end{array}}{A \vdash \tau \subseteq \tau''}$$

By induction, $AS \vdash \tau S \subseteq \tau' S$ and $AS \vdash \tau' S \subseteq \tau'' S$, so by (trans), $AS \vdash \tau S \subseteq \tau'' S$.

$((-) \rightarrow (+))$ The derivation ends with

$$\frac{\begin{array}{l} A \vdash \tau' \subseteq \tau \\ A \vdash \rho \subseteq \rho' \end{array}}{A \vdash (\tau \rightarrow \rho) \subseteq (\tau' \rightarrow \rho')}$$

By induction, $AS \vdash \tau' S \subseteq \tau S$ and $AS \vdash \rho S \subseteq \rho' S$, so by $((-) \rightarrow (+))$, $AS \vdash (\tau S \rightarrow \rho S) \subseteq (\tau' S \rightarrow \rho' S)$.

Other type constructors are handled in the same way. **QED**

Lemma 3.2 *If $A \cup \{x : \sigma\} \vdash \tau \subseteq \tau'$, then $A \vdash \tau \subseteq \tau'$.*

Proof: By a straightforward induction on the length of the derivation of $A \cup \{x : \sigma\} \vdash \tau \subseteq \tau'$. **QED**

Lemma 3.3 *If $A \vdash \tau \subseteq \tau'$, then $A \cup B \vdash \tau \subseteq \tau'$.*

Proof: By induction on the length of the derivation of $A \vdash \tau \subseteq \tau'$. **QED**

Lemma 3.4 *If* $A \cup B \vdash \tau \subseteq \tau'$ *and* $A \vdash B$, *then* $A \vdash \tau \subseteq \tau'$.

Proof: Yet another straightforward induction, this time on the length of the derivation of $A \cup B \vdash \tau \subseteq \tau'$. **QED**

Now we show that the properties proved in Section 2.2 still hold in our new system with subtyping.

Here is the counterpart to Lemma 2.4:

Lemma 3.5 *If* $A \vdash e : \sigma$ *then* $AS \vdash e : \sigma S$. *Furthermore, given any derivation of* $A \vdash e : \sigma$, *a derivation of* $AS \vdash e : \sigma S$ *can be constructed that uses the same sequence of steps except that it inserts a* ($\equiv_\alpha$) *step after each* ($\forall$-intro) *step.*

Proof: The proof of Lemma 2.4 suffices, except that we must consider the case when the derivation of $A \vdash e : \sigma$ ends with ($\subseteq$).

($\subseteq$) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \tau \\ A \vdash \tau \subseteq \tau' \end{array}}{A \vdash e : \tau'}$$

By induction, $AS \vdash e : \tau S$, and by Lemma 3.1, $AS \vdash \tau S \subseteq \tau' S$. So by ($\subseteq$), $AS \vdash e : \tau' S$.

QED

Again, our major goal is to prove a normal-form theorem for derivations. Define rule ($\forall$-elim$'$) as in Chapter 2 (see page 26), and say $A \vdash' e : \sigma$ and $A \vdash' \tau \subseteq \tau'$ if these can be derived in the system obtained by deleting ($\forall$-elim) and replacing it with ($\forall$-elim$'$). Because $\vdash'$ does not restrict the inclusion sublogic in any way, the following lemma is immediate.

Lemma 3.6 $A \vdash \tau \subseteq \tau'$ *if and only if* $A \vdash' \tau \subseteq \tau'$.

Here is the counterpart to Corollary 2.5:

Corollary 3.7 *If* $A \vdash' e : \sigma$ *then* $AS \vdash' e : \sigma S$. *Furthermore, given any derivation of* $A \vdash' e : \sigma$, *a derivation of* $AS \vdash' e : \sigma S$ *can be constructed that uses the same sequence of steps except that it inserts a* ($\equiv_\alpha$) *step after each* ($\forall$-intro) *step.*

The counterparts to Lemmas 2.6, 2.7, and 2.8 are given next; their straightforward proofs are given in Appendix A.

Lemma 3.8 *If* x *does not occur in* A, $x \neq y$, *and* $A \cup \{x : \sigma\} \vdash' y : \tau$, *then* $A \vdash' y : \tau$.

Lemma 3.9 *If* $A \vdash' e : \sigma$ *then* $A \cup B \vdash' e : \sigma$, *provided that no identifier occurring in* B *is* λ*-bound or let-bound in* e.

Lemma 3.10 *Let* B *be a set of* τ*-assumptions. If* $A \cup B \vdash' e : \sigma$ *and* $A \vdash' B$, *then* $A \vdash' e : \sigma$.

With these lemmas, we can now show the counterpart to Theorem 2.9.

Theorem 3.11 $A \vdash e : \sigma$ *if and only if* $A \vdash' e : \sigma$.

Proof: The proof of Theorem 2.9 suffices, except that when the derivation of $A \vdash e : \sigma$ ends with $(\subseteq)$, we use the fact that $A \vdash \tau \subseteq \tau'$ if and only if $A \vdash' \tau \subseteq \tau'$. **QED**

As in Chapter 2, we get the following corollaries:

Corollary 3.12 *If* x *does not occur in* A, $x \neq y$, *and* $A \cup \{x : \sigma\} \vdash y : \tau$, *then* $A \vdash y : \tau$.

Corollary 3.13 *If* $A \vdash e : \sigma$ *then* $A \cup B \vdash e : \sigma$, *provided that no identifier occurring in* B *is* λ*-bound or let-bound in* e.

Corollary 3.14 *Let* B *be a set of* τ*-assumptions. If* $A \cup B \vdash e : \sigma$ *and* $A \vdash B$, *then* $A \vdash e : \sigma$.

Lemma 3.15 *If* $A \vdash e : \sigma$ *then no identifier occurring in* A *is* λ*-bound or let-bound in* e.

Corollary 3.16 *If* $A \vdash e : \sigma$ *then* $A \cup B \vdash e : \sigma$, *provided that all identifiers occurring in* B *also occur in* A.

As in Chapter 2, we define the notion of an assumption set with *satisfiable constraints*; both of the lemmas about assumption sets with satisfiable constraints carry over without change.

Lemma 3.17 *If* A *has satisfiable constraints and* $A \vdash e : \sigma$, *then there exists a type* τ *such that* $A \vdash e : \tau$.

Lemma 3.18 *If* A *has satisfiable constraints and there exists a substitution* $[\bar{\alpha} := \bar{\pi}]$ *such that* $A \vdash C[\bar{\alpha} := \bar{\pi}]$, *then* $A \cup \{x : \forall\bar{\alpha}\ \textbf{with}\ C \,.\, \tau\}$ *has satisfiable constraints.*

Algorithm W_{os} and the new *close* given below make certain demands about the inclusions in the assumption sets that they are given. In particular, they disallow assumptions like $int \subseteq (int \rightarrow int)$ in which the two sides of the inclusion do not have the same 'shape'. Furthermore, they disallow 'cyclic' sets of inclusions such as $bool \subseteq int$ together with $int \subseteq bool$. The precise restrictions are given in the following definitions. A *constant inclusion* is an inclusion of the form $c \subseteq d$, where c and d are type constants (i.e. 0-ary type constructors).

Definition 3.2 *An assumption set A is said to have* acceptable inclusions *if*

- *A contains only constant inclusions and*
- *the transitive closure of the inclusions in A is antisymmetric.*

We now establish a few properties of assumption sets with acceptable inclusions. Recall that *atomic* types are type constants or type variables.

Definition 3.3 *Types τ and τ' have the* same shape *if either*

- *τ and τ' are atomic or*
- *$\tau = \chi(\tau_1, \ldots, \tau_n)$, $\tau' = \chi(\tau'_1, \ldots, \tau'_n)$, where χ is an n-ary type constructor, $n \geq 1$, and for all i, τ_i and τ'_i have the same shape.*

This definition appears in [Mit84] under the name 'matching', and also in [Sta88] and [FM90]. Note that the 'same shape' relation is an equivalence relation.

Define an *atomic inclusion* to be an inclusion of the form $\tau \subseteq \tau'$, where τ and τ' are atomic. Any constant inclusion is an atomic inclusion, so the following lemmas apply to any assumption set with acceptable inclusions.

Lemma 3.19 *If A contains only atomic inclusions and $A \vdash \tau \subseteq \tau'$, then τ and τ' have the same shape.*

Proof: By induction on the length of the derivation of $A \vdash \tau \subseteq \tau'$. **QED**

Lemma 3.20 *If A contains only atomic inclusions and $A \vdash \tau \rightarrow \rho \subseteq \tau' \rightarrow \rho'$, then $A \vdash \tau' \subseteq \tau$ and $A \vdash \rho \subseteq \rho'$.*

Proof: By induction on the length of the derivation of $A \vdash \tau \rightarrow \rho \subseteq \tau' \rightarrow \rho'$. Consider the last step of the derivation. Since A contains only atomic inclusions, the last step cannot be (hypoth). Hence the last step is (reflex), (trans), or $((-) \rightarrow (+))$. In the (reflex) and $((-) \rightarrow (+))$ cases, the result follows immediately. Suppose the derivation ends with (trans):

$$\frac{\begin{array}{l} A \vdash (\tau \rightarrow \rho) \subseteq \pi \\ A \vdash \pi \subseteq (\tau' \rightarrow \rho') \end{array}}{A \vdash (\tau \rightarrow \rho) \subseteq (\tau' \rightarrow \rho')}$$

By Lemma 3.19, $\tau \rightarrow \rho$ and π have the same shape, so π is of the form $\pi' \rightarrow \pi''$. By induction, we have $A \vdash \pi' \subseteq \tau$, $A \vdash \rho \subseteq \pi''$, $A \vdash \tau' \subseteq \pi'$, and $A \vdash \pi'' \subseteq \rho'$. Hence by (trans) we have $A \vdash \tau' \subseteq \tau$ and $A \vdash \rho \subseteq \rho'$. **QED**

In the same way, one can prove similar lemmas for the other type constructors.

Lemma 3.21 *If A has acceptable inclusions, $A \vdash \tau \subseteq \tau'$, and $A \vdash \tau' \subseteq \tau$, then $\tau = \tau'$.*

Proof: Since A contains only atomic inclusions, τ and τ' have the same shape. Hence we may use induction on the structure of τ and τ'. If τ and τ' are atomic, then both $\tau \subseteq \tau'$ and $\tau' \subseteq \tau$ are in the reflexive transitive closure of the inclusions in A. Since by assumption the transitive closure of the inclusions in A is antisymmetric, it follows that $\tau = \tau'$.

Now suppose that $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau_1' \to \tau_2'$. By Lemma 3.20, we have $A \vdash \tau_1' \subseteq \tau_1$, $A \vdash \tau_2 \subseteq \tau_2'$, $A \vdash \tau_1 \subseteq \tau_1'$, and $A \vdash \tau_2' \subseteq \tau_2$. So by induction, $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.

Other type constructors are handled similarly. QED

3.3 Algorithm W_{os}

Algorithm W_{os} is given in Figure 3.2. The algorithm differs from W_o in two ways. First, there is a small change to the **let** case; $close(A, B, \tau)$ now returns a substitution S as well as a constraint set B' and a type scheme σ. The idea is that *close* may discover that some of the type variables in B can be instantiated in only one way if B is to be satisfied; any such instantiations are returned in the substitution S.[1] The major departure from W_o is in the case of an application $e'e''$. Intuitively, W_{os} returns the smallest possible type for an expression. Normally, such a type can be used in a derivation in place of a larger type. The one exception is in the ($\to$-elim) case, where the type of the argument e'' is required to be the same as the domain of the function e'; in this case, it may be necessary to pass from the type found by W_{os} for the argument e'' to a supertype.

We now discuss *close*. For now, we trivially modify the *close* of Figure 2.3 so that it always returns the identity substitution. In Section 3.6 we will give a fancy *close* and prove that it satisfies the two lemmas needed to establish the soundness and completeness of W_{os}.

3.4 Properties of W_{os}

3.4.1 Soundness of W_{os}

The soundness of *close* is given by the following lemma.

Lemma 3.22 *If $(S, B', \sigma) = close(A, B, \tau)$ succeeds, then for any e, if $A \cup B \vdash e : \tau$ then $AS \cup B' \vdash e : \sigma$. Also, every identifier occurring in B' or in the constraints of σ occurs in B.*

[1] W_o could also be modified to use a version of *close* that returns a substitution, but without subtyping it doesn't seem important to do so.

$W_{os}(A, e)$ is defined by cases:

1. e is x

   if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
   return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else if $(x : \forall\bar{\alpha}$ **with** $C\,.\,\tau\,) \in A$,
   return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
   else *fail.*

2. e is $\lambda x.e'$

   if x occurs in A, *fail.*
   let $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ where α is new;
   return $(S_1, B_1, \alpha S_1 \to \tau_1)$.

3. e is $e'e''$

   let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
   let $(S_2, B_2, \tau_2) = W_{os}(AS_1, e'')$;
   let $S_3 = \mathit{unify}(\tau_1 S_2, \alpha \to \beta)$ where α and β are new;
   return $(S_1S_2S_3, B_1S_2S_3 \cup B_2S_3 \cup \{\tau_2 S_3 \subseteq \alpha S_3\}, \beta S_3)$.

4. e is **let** $x = e'$ **in** e''

   if x occurs in A, *fail.*
   let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
   let $(S_2, B_1', \sigma_1) = \mathit{close}(AS_1, B_1, \tau_1)$;
   let $(S_3, B_2, \tau_2) = W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'')$;
   return $(S_1S_2S_3, B_1'S_3 \cup B_2, \tau_2)$.

Figure 3.2: Algorithm W_{os}

Next we give the soundness of W_{os}; the proof is very similar to the proof of Theorem 2.18.

Theorem 3.23 *If $(S, B, \tau) = W_{os}(A, e)$ succeeds, then $AS \cup B \vdash e : \tau$. Furthermore, every identifier occurring in B is overloaded in A or occurs in some constraint of some assumption in A.*

3.4.2 Completeness of W_{os}

The properties of *close* needed to prove the completeness of W_{os} are extracted more carefully here than they were in Chapter 2, thereby giving *close* more freedom to do type simplification. This freedom will be exploited in Section 3.6.

Lemma 3.24 *Suppose that A has acceptable inclusions and $AR \vdash BR$. Then $(S, B', \sigma) = close(A, B, \tau)$ succeeds and*

- *if A has no free type variables then $B' = \{\}$,*
- *$fv(\sigma) \subseteq fv(AS)$, and*
- *there exists T such that*
  1. *$R = ST$,*
  2. *$AR \vdash B'T$, and*
  3. *$\sigma T \geq_{AR} \tau R$.*

The next two lemmas have straightforward inductive proofs.

Lemma 3.25 *If $(S, B', \sigma) = close(A, B, \tau)$ succeeds, then every type variable occurring free in S, B', or σ occurs free in A, B, or τ or else is generated by the new type variable generator.*

Lemma 3.26 *If $(S, B, \tau) = W_{os}(A, e)$ succeeds, then every type variable occurring in S, B, or τ occurs free in A or else is generated by the new type variable generator.*

The definition of $A \succeq A'$ needs to be revised to account for inclusions; we simply demand that A and A' contain the same inclusions.

Definition 3.4 *Let A and A' be assumption sets. We say that A is* stronger than *A', written $A \succeq A'$, if*

1. *$A' \vdash x : \tau$ implies $A \vdash x : \tau$,*
2. *$id(A) \subseteq id(A')$, and*
3. *A and A' contain the same inclusions.*

We now show the completeness of W_{os}.

Theorem 3.27 *Suppose that $A' \vdash e : \tau$, A' has satisfiable constraints, $AS \succeq A'$, and A has acceptable inclusions. Then $(S_0, B_0, \tau_0) = W_{os}(A, e)$ succeeds and there exists a substitution T such that*

1. *$S = S_0 T$, except on new type variables of $W_{os}(A, e)$,*
2. *$AS \vdash B_0 T$, and*
3. *$AS \vdash \tau_0 T \subseteq \tau$.*

Proof: By induction on the structure of e. By Theorem 3.11, $A' \vdash' e : \tau$.

- <u>e is x</u>

  By the definition of $AS \succeq A'$, we have $AS \vdash' x : \tau$. Without loss of generality, we may assume that the derivation of $AS \vdash' x : \tau$ ends with a (possibly trivial) use of ($\forall$-elim$'$) followed by a (possibly trivial) use of ($\subseteq$). If $(x : \forall\bar{\alpha}$ **with** $C \,.\, \rho) \in A$, then $(x : \forall\bar{\beta}$ **with** $C[\bar{\alpha} := \bar{\beta}]S \,.\, \rho[\bar{\alpha} := \bar{\beta}]S\,) \in AS$, where $\bar{\beta}$ are the first distinct type variables not free in $\forall\bar{\alpha}$ **with** $C \,.\, \rho$ or in S. Hence the derivation $AS \vdash' x : \tau$ ends with

$$\cfrac{\cfrac{\begin{array}{l}(x : \forall\bar{\beta}\ \textbf{with}\ C[\bar{\alpha} := \bar{\beta}]S \,.\, \rho[\bar{\alpha} := \bar{\beta}]S\,) \in AS \\ AS \vdash' C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]\end{array}}{AS \vdash' x : \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]} \qquad AS \vdash' \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau}{AS \vdash' x : \tau}$$

  We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds and that there exists T such that

  1. $S = S_0 T$, except on new type variables of $W_{os}(A, x)$,
  2. $AS \vdash B_0 T$, and
  3. $AS \vdash \tau_0 T \subseteq \tau$.

  Now, $W_{os}(A, x)$ is defined by

  > if x is overloaded in A with *lcg* $\forall\bar{\alpha}.\tau$,
  > return $([\,], \{x : \tau[\bar{\alpha} := \bar{\beta}]\}, \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
  > else if $(x : \forall\bar{\alpha}$ **with** $C \,.\, \tau\,) \in A$,
  > return $([\,], C[\bar{\alpha} := \bar{\beta}], \tau[\bar{\alpha} := \bar{\beta}])$ where $\bar{\beta}$ are new
  > else *fail.*

  If x is overloaded in A with *lcg* $\forall\bar{\gamma}.\rho_0$, then $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds with $S_0 = [\,]$, $B_0 = \{x : \rho_0[\bar{\gamma} := \bar{\delta}]\}$, and $\tau_0 = \rho_0[\bar{\gamma} := \bar{\delta}]$, where $\bar{\delta}$ are new. Since $\forall\bar{\gamma}.\rho_0$ is the *lcg* of x , $\bar{\gamma}$ are the only variables in ρ_0 and there exist $\bar{\phi}$ such that $\rho_0[\bar{\gamma} := \bar{\phi}] = \rho$. Let

$$T = S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]].$$

Then

$$
\begin{array}{ll}
& \tau_0 T \\
= & \quad \ll \text{definition} \gg \\
& \rho_0[\bar{\gamma} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\delta} \text{ occur in } \rho_0[\bar{\gamma} := \bar{\delta}] \gg \\
& \rho_0[\bar{\gamma} := \bar{\delta}][\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
& \rho_0[\bar{\gamma} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] \\
= & \quad \ll \text{only } \bar{\gamma} \text{ occur in } \rho_0 \gg \\
& \rho_0[\bar{\gamma} := \bar{\phi}][\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \\
= & \quad \ll \text{by above} \gg \\
& \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}].
\end{array}
$$

So

1. $S_0T = S \oplus [\bar{\delta} := \bar{\phi}[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]] = S$, except on $\bar{\delta}$. That is, $S_0T = S$, except on the new type variables of $W_{os}(A, x)$.
2. Since $B_0T = \{x : \tau_0T\} = \{x : \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]\}$, it follows that $AS \vdash B_0T$.
3. We have $\tau_0T = \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ and $AS \vdash \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau$, so $AS \vdash \tau_0T \subseteq \tau$.

If x is not overloaded in A, then $(S_0, B_0, \tau_0) = W_{os}(A, x)$ succeeds with $S_0 = [\,]$, $B_0 = C[\bar{\alpha} := \bar{\delta}]$, and $\tau_0 = \rho[\bar{\alpha} := \bar{\delta}]$, where $\bar{\delta}$ are new. Observe that $[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}])$ and $[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]$ agree on C and on ρ.

> (The only variables in C or ρ are the $\bar{\alpha}$ and variables ε not among $\bar{\beta}$ or $\bar{\delta}$. Both substitutions map $\alpha_i \mapsto \pi_i$ and $\varepsilon \mapsto \varepsilon S$.)

Let T be $S \oplus [\bar{\delta} := \bar{\pi}]$. Then

1. $S_0T = S \oplus [\bar{\delta} := \bar{\pi}] = S$, except on $\bar{\delta}$. That is, $S_0T = S$, except on the new type variables of $W_{os}(A, x)$.
2. Also,

$$
\begin{array}{ll}
& B_0 T \\
= & \quad \ll \text{definition} \gg \\
& C[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) \\
= & \quad \ll \text{by above observation} \gg \\
& C[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}].
\end{array}
$$

   Hence $AS \vdash B_0T$.

3. Finally,

$$\begin{array}{ll} & \tau_0 T \\ = & \quad \ll \text{definition} \gg \\ & \rho[\bar{\alpha} := \bar{\delta}](S \oplus [\bar{\delta} := \bar{\pi}]) \\ = & \quad \ll \text{by above observation} \gg \\ & \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}]. \end{array}$$

Since $AS \vdash \rho[\bar{\alpha} := \bar{\beta}]S[\bar{\beta} := \bar{\pi}] \subseteq \tau$, we have $AS \vdash \tau_0 T \subseteq \tau$.

- <u>e is $\lambda x.e'$</u>

Without loss of generality, we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of ($\rightarrow$-intro) followed by a (possibly trivial) use of ($\subseteq$):

$$\cfrac{\cfrac{A' \cup \{x : \tau'\} \vdash' e' : \tau''}{A' \vdash' \lambda x.e' : \tau' \rightarrow \tau''} \quad \\ A' \vdash' (\tau' \rightarrow \tau'') \subseteq \tau}{A' \vdash' \lambda x.e' : \tau}$$

where x does not occur in A'.

We must show that $(S_0, B_0, \tau_0) = W_{os}(A, \lambda x.e')$ succeeds and that there exists T such that

1. $S = S_0 T$, except on new type variables of $W_{os}(A, \lambda x.e')$,
2. $AS \vdash B_0 T$, and
3. $AS \vdash \tau_0 T \subseteq \tau$.

Now, $W_{os}(A, \lambda x.e')$ is defined by

> if x occurs in A, *fail.*
> let $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ where α is new;
> return $(S_1, B_1, \alpha S_1 \rightarrow \tau_1)$.

Since x does not occur in A' and (by $AS \succeq A'$) $id(AS) \subseteq id(A')$, it follows that x does not occur in AS. This implies that x does not occur in A, so the first line does not fail.

Now we wish to use induction to show that the recursive call succeeds. The new type variable α is not free in A, so

$$AS \cup \{x : \tau'\} = (A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']).$$

By Lemma 3.18, $A' \cup \{x : \tau'\}$ has satisfiable constraints. Next we argue that $AS \cup \{x : \tau'\} \succeq A' \cup \{x : \tau'\}$. Clearly, $id(AS \cup \{x : \tau'\}) \subseteq id(A' \cup \{x : \tau'\})$, and $AS \cup \{x : \tau'\}$ and $A' \cup \{x : \tau'\}$ contain the same inclusions. Now suppose that $A' \cup \{x : \tau'\} \vdash y : \rho$. If $y \neq x$, then by Corollary 3.12,

$A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Corollary 3.13, $AS \cup \{x : \tau'\} \vdash y : \rho$. On the other hand, if $y = x$, then the derivation $A' \cup \{x : \tau'\} \vdash' y : \rho$ must be by (hypoth) followed by a (possibly trivial) use of ($\subseteq$):

$$\frac{\begin{array}{c} A' \cup \{x : \tau'\} \vdash' x : \tau' \\ A' \cup \{x : \tau'\} \vdash' \tau' \subseteq \rho \end{array}}{A' \cup \{x : \tau'\} \vdash' x : \rho}$$

Since $AS \succeq A'$, AS and A' contain the same inclusions. Therefore, $AS \cup \{x : \tau'\} \vdash' \tau' \subseteq \rho$, so by (hypoth) followed by ($\subseteq$), $AS \cup \{x : \tau'\} \vdash' x : \rho$. Finally, $A \cup \{x : \alpha\}$ has acceptable inclusions. So we have

- $A' \cup \{x : \tau'\} \vdash e' : \tau''$,
- $A' \cup \{x : \tau'\}$ has satisfiable constraints,
- $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \succeq A' \cup \{x : \tau'\}$, and
- $A \cup \{x : \alpha\}$ has acceptable inclusions.

So by induction, $(S_1, B_1, \tau_1) = W_{os}(A \cup \{x : \alpha\}, e')$ succeeds and there exists T_1 such that

1. $S \oplus [\alpha := \tau'] = S_1 T_1$, except on new variables of $W_{os}(A \cup \{x : \alpha\}, e')$,
2. $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \vdash B_1 T_1$, and
3. $(A \cup \{x : \alpha\})(S \oplus [\alpha := \tau']) \vdash \tau_1 T_1 \subseteq \tau''$.

So $(S_0, B_0, \tau_0) = W_{os}(A, \lambda x.e')$ succeeds with $S_0 = S_1$, $B_0 = B_1$, and $\tau_0 = \alpha S_1 \rightarrow \tau_1$.

Let T be T_1. Then

1. Observe that

$$\begin{array}{ll} & S_0 T \\ = & \quad \ll \text{definition} \gg \\ & S_1 T_1 \\ = & \quad \ll \text{by part 1 of the use of induction above} \gg \\ & S \oplus [\alpha := \tau'], \text{ except on new variables of } W_{os}(A \cup \{x : \alpha\}, e') \\ = & \quad \ll \text{definition of } \oplus \gg \\ & S, \text{ except on } \alpha. \end{array}$$

   Hence $S_0 T = S$, except on the new type variables of $W_{os}(A \cup \{x : \alpha\}, e')$ and on α. That is, $S_0 T = S$, except on the new type variables of $W_{os}(A, \lambda x.e')$.

2. $B_0T = B_1T_1$ and, by part 2 of the use of induction above, we have $AS \cup \{x : \tau'\} \vdash B_1T_1$. By Theorem 3.23, every identifier occurring in B_1 is overloaded in $A \cup \{x : \alpha\}$ or occurs in some constraint of some assumption in $A \cup \{x : \alpha\}$. Since x does not occur in A and α has no constraints, this means that x does not occur in B_1. Hence Corollary 3.12 and Lemma 3.2 may be applied to each member of B_1T_1, yielding $AS \vdash B_1T_1$.

3. Finally,

$$\begin{array}{ll} & \tau_0 T \\ = & \qquad \ll \text{definition} \gg \\ & \alpha S_1 T_1 \rightarrow \tau_1 T_1 \\ = & \qquad \ll \alpha \text{ is not a new type variable of } W_{os}(A \cup \{x:\alpha\}, e') \gg \\ & \alpha(S \oplus [\alpha := \tau']) \rightarrow \tau_1 T_1 \\ = & \qquad \ll \text{definition of } \oplus \gg \\ & \tau' \rightarrow \tau_1 T_1. \end{array}$$

Now by part 3 of the use of induction above, $AS \cup \{x : \tau'\} \vdash \tau_1 T_1 \subseteq \tau''$, so by Lemma 3.2, $AS \vdash \tau_1 T_1 \subseteq \tau''$. Since by (reflex), $AS \vdash \tau' \subseteq \tau'$, it follows from $((-) \rightarrow (+))$ that $AS \vdash (\tau' \rightarrow \tau_1 T_1) \subseteq (\tau' \rightarrow \tau'')$. In other words, $AS \vdash \tau_0 T \subseteq (\tau' \rightarrow \tau'')$. Next, because $A' \vdash (\tau' \rightarrow \tau'') \subseteq \tau$ and $AS \succeq A'$, we have $AS \vdash (\tau' \rightarrow \tau'') \subseteq \tau$. So by (trans) we have $AS \vdash \tau_0 T \subseteq \tau$.

- <u>e is $e'e''$</u>

Without loss of generality, we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of ($\rightarrow$-elim) followed by a use of ($\subseteq$):

$$\cfrac{\cfrac{A' \vdash' e' : \tau' \rightarrow \tau'' \qquad A' \vdash' e'' : \tau'}{A' \vdash' e'\,e'' : \tau''} \qquad A' \vdash' \tau'' \subseteq \tau}{A' \vdash' e'\,e'' : \tau}$$

We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, e'e'')$ succeeds and that there exists T such that

1. $S = S_0T$, except on new type variables of $W_{os}(A, e'e'')$,
2. $AS \vdash B_0T$, and
3. $AS \vdash \tau_0 T \subseteq \tau$.

Now, $W_{os}(A, e'e'')$ is defined by

let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
let $(S_2, B_2, \tau_2) = W_{os}(AS_1, e'')$;
let $S_3 = \mathit{unify}(\tau_1 S_2, \alpha \to \beta)$ where α and β are new;
return $(S_1S_2S_3, B_1S_2S_3 \cup B_2S_3 \cup \{\tau_2 S_3 \subseteq \alpha S_3\}, \beta S_3)$.

By induction, $(S_1, B_1, \tau_1) = W_{os}(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1 T_1$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1 T_1$, and
3. $AS \vdash \tau_1 T_1 \subseteq (\tau' \to \tau'')$.

Since A has acceptable inclusions, AS contains only atomic inclusions. Hence by Lemma 3.19, $\tau_1 T_1$ is of the form $\rho \to \rho'$, and by Lemma 3.20, we have $AS \vdash \tau' \subseteq \rho$ and $AS \vdash \rho' \subseteq \tau''$.

Now $AS = A(S_1T_1) = (AS_1)T_1$, as the new type variables of $W_{os}(A, e')$ do not occur free in A. So by induction, $(S_2, B_2, \tau_2) = W_{os}(AS_1, e'')$ succeeds and there exists T_2 such that

1. $T_1 = S_2 T_2$, except on new type variables of $W_{os}(AS_1, e'')$,
2. $(AS_1)T_1 \vdash B_2 T_2$, and
3. $(AS_1)T_1 \vdash \tau_2 T_2 \subseteq \tau'$.

By Lemma 3.26, the new type variables α and β do not occur in A, S_1, B_1, τ_1, S_2, B_2, or τ_2. So consider $T_2 \oplus [\alpha, \beta := \rho, \rho']$:

$$\begin{array}{ll} & (\tau_1 S_2)(T_2 \oplus [\alpha, \beta := \rho, \rho']) \\ = & \qquad \ll \alpha \text{ and } \beta \text{ do not occur in } \tau_1 S_2 \gg \\ & \tau_1 S_2 T_2 \\ = & \qquad \ll \text{no new type variable of } W_{os}(AS_1, e'') \text{ occurs in } \tau_1 \gg \\ & \tau_1 T_1 \\ = & \qquad \ll \text{by above} \gg \\ & \rho \to \rho' \end{array}$$

In addition,

$$\begin{array}{ll} & (\alpha \to \beta)(T_2 \oplus [\alpha, \beta := \rho, \rho']) \\ = & \qquad \ll \text{definition of } \oplus \gg \\ & \rho \to \rho' \end{array}$$

Hence $S_3 = \mathit{unify}(\tau_1 S_2, \alpha \to \beta)$ succeeds and there exists T_3 such that

$$T_2 \oplus [\alpha, \beta := \rho, \rho'] = S_3 T_3.$$

So $(S_0, B_0, \tau_0) = W_{os}(A, e'e'')$ succeeds with $S_0 = S_1 S_2 S_3$, $B_0 = B_1 S_2 S_3 \cup B_2 S_3 \cup \{\tau_2 S_3 \subseteq \alpha S_3\}$, and $\tau_0 = \beta S_3$.

Let T be T_3. Then

1. Observe that

$$
\begin{array}{ll}
 & S_0T \\
= & \quad \ll \text{definition} \gg \\
 & S_1S_2S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
 & S_1S_2(T_2 \oplus [\alpha,\beta := \rho,\rho']) \\
= & \quad \ll \alpha \text{ and } \beta \text{ do not occur in } S_1S_2 \gg \\
 & S_1S_2T_2, \text{ except on } \alpha \text{ and } \beta \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
 & \quad \ll \text{new variables of } W_{os}(AS_1, e'') \text{ don't occur in } S_1 \gg \\
 & S_1T_1, \text{ except on the new type variables of } W_{os}(AS_1, e'') \\
= & \quad \ll \text{by part 1 of the first use of induction above} \gg \\
 & S, \text{ except on the new type variables of } W_{os}(A, e').
\end{array}
$$

Hence we see that $S_0T = S$ except on α, on β, on the new type variables of $W_{os}(AS_1, e'')$, and on the new type variables of $W_{os}(A, e')$. That is, $S_0T = S$ except on the new type variables of $W_{os}(A, e'e'')$.

2. Next,

$$
\begin{array}{ll}
 & B_0T \\
= & \quad \ll \text{definition} \gg \\
 & B_1S_2S_3T_3 \cup B_2S_3T_3 \cup \{\tau_2S_3T_3 \subseteq \alpha S_3T_3\} \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
 & B_1S_2(T_2 \oplus [\alpha,\beta := \rho,\rho']) \cup B_2(T_2 \oplus [\alpha,\beta := \rho,\rho']) \\
 & \qquad \cup\{\tau_2(T_2 \oplus [\alpha,\beta := \rho,\rho']) \subseteq \alpha(T_2 \oplus [\alpha,\beta := \rho,\rho'])\} \\
= & \quad \ll \alpha \text{ and } \beta \text{ do not occur in } B_1S_2,\ B_2, \text{ or } \tau_2 \gg \\
 & B_1S_2T_2 \cup B_2T_2 \cup \{\tau_2T_2 \subseteq \rho\} \\
= & \quad \ll \text{by part 1 of second use of induction and since the} \gg \\
 & \quad \ll \text{new variables of } W_{os}(AS_1, e'') \text{ don't occur in } B_1 \gg \\
 & B_1T_1 \cup B_2T_2 \cup \{\tau_2T_2 \subseteq \rho\}
\end{array}
$$

By part 2 of the first and second uses of induction above, $AS \vdash B_1T_1$ and $AS \vdash B_2T_2$. By part 3 of the second use of induction above, $AS \vdash \tau_2T_2 \subseteq \tau'$. Also, we found above that $AS \vdash \tau' \subseteq \rho$. So by (trans), $AS \vdash \tau_2T_2 \subseteq \rho$. Therefore, $AS \vdash B_0T$.

3. Finally,

$$
\begin{array}{ll}
 & \tau_0T \\
= & \quad \ll \text{definition} \gg \\
 & \beta S_3T_3 \\
= & \quad \ll \text{by above property of unifier } S_3 \gg \\
 & \beta(T_2 \oplus [\alpha,\beta := \rho,\rho']) \\
= & \quad \ll \text{definition of } \oplus \gg \\
 & \rho'
\end{array}
$$

Now, $AS \vdash \rho' \subseteq \tau''$ and, since $A' \vdash \tau'' \subseteq \tau$ and $AS \succeq A'$, also $AS \vdash \tau'' \subseteq \tau$. So it follows from (trans) that $AS \vdash \tau_0 T \subseteq \tau$.

- $\underline{e \text{ is } \mathbf{let}\ x = e' \ \mathbf{in}\ e''}$

Without loss of generality we may assume that the derivation of $A' \vdash' e : \tau$ ends with a use of (let) followed by a (possibly trivial) use of ($\subseteq$):

$$\dfrac{\dfrac{\begin{array}{l} A' \vdash' e' : \sigma \\ A' \cup \{x : \sigma\} \vdash' e'' : \tau' \end{array}}{A' \vdash' \mathbf{let}\ x = e' \ \mathbf{in}\ e'' : \tau'} \quad A' \vdash' \tau' \subseteq \tau}{A' \vdash' \mathbf{let}\ x = e' \ \mathbf{in}\ e'' : \tau}$$

where x does not occur in A'.

We need to show that $(S_0, B_0, \tau_0) = W_{os}(A, \mathbf{let}\ x = e' \ \mathbf{in}\ e'')$ succeeds and that there exists T such that

1. $S = S_0 T$, except on new type variables of $W_{os}(A, \mathbf{let}\ x = e' \ \mathbf{in}\ e'')$,
2. $AS \vdash B_0 T$, and
3. $AS \vdash \tau_0 T \subseteq \tau$.

Now, $W_{os}(A, \mathbf{let}\ x = e' \ \mathbf{in}\ e'')$ is defined by

> if x occurs in A, *fail*.
> let $(S_1, B_1, \tau_1) = W_{os}(A, e')$;
> let $(S_2, B_1', \sigma_1) = close(AS_1, B_1, \tau_1)$;
> let $(S_3, B_2, \tau_2) = W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$;
> return $(S_1 S_2 S_3, B_1' S_3 \cup B_2, \tau_2)$.

Since $AS \succeq A'$, x does not occur in AS or in A. So the first line does not fail.

Since A' has satisfiable constraints, by Lemma 3.17 there exists τ'' such that $A' \vdash e' : \tau''$. Hence by induction, $(S_1, B_1, \tau_1) = W_{os}(A, e')$ succeeds and there exists T_1 such that

1. $S = S_1 T_1$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1 T_1$, and
3. $AS \vdash \tau_1 T_1 \subseteq \tau''$.

By 1 and 2 above, we have $(AS_1)T_1 \vdash B_1 T_1$. Since AS_1 has acceptable inclusions, by Lemma 3.24 it follows that $(S_2, B_1', \sigma_1) = close(AS_1, B_1, \tau_1)$ succeeds, $fv(\sigma_1) \subseteq fv(AS_1 S_2)$, and there exists a substitution T_2 such that $T_1 = S_2 T_2$ and $(AS_1)T_1 \vdash B_1' T_2$.

Now, in order to apply the induction hypothesis to the second recursive call we need to show that

$$AS \cup \{x : \sigma_1 T_2\} \succeq A' \cup \{x : \sigma\}.$$

There are three parts to this:

1. $A' \cup \{x : \sigma\} \vdash y : \rho$ implies $AS \cup \{x : \sigma_1 T_2\} \vdash y : \rho$,
2. $id(AS \cup \{x : \sigma_1 T_2\}) \subseteq id(A' \cup \{x : \sigma\})$,
3. $AS \cup \{x : \sigma_1 T_2\}$ and $A' \cup \{x : \sigma\}$ contain the same inclusions.

The second and third parts are quite simple; we show them first. By Lemma 3.22 and Theorem 3.23, $id(\sigma_1) \subseteq id(B_1) \subseteq id(A)$. Hence

$$\begin{array}{cl} & id(AS \cup \{x : \sigma_1 T_2\}) \\ = & \qquad \ll \text{by above} \gg \\ & id(AS) \cup \{x\} \\ \subseteq & \qquad \ll AS \succeq A' \gg \\ & id(A') \cup \{x\} \\ \subseteq & \\ & id(A' \cup \{x : \sigma\}), \end{array}$$

showing 2.

Since $AS \succeq A'$, it follows that $AS \cup \{x : \sigma_1 T_2\}$ and $A' \cup \{x : \sigma\}$ contain the same inclusions, showing 3.

Now we show 1. Suppose that $A' \cup \{x : \sigma\} \vdash y : \rho$. If $y \neq x$, then by Corollary 3.12, $A' \vdash y : \rho$. Since $AS \succeq A'$, we have $AS \vdash y : \rho$ and then by Corollary 3.13, $AS \cup \{x : \sigma_1 T_2\} \vdash y : \rho$.

If, on the other hand, $y = x$, then our argument will begin by establishing that $A' \vdash e' : \rho$. By Theorem 3.11 we have $A' \cup \{x : \sigma\} \vdash' x : \rho$ and we may assume that the derivation ends with a use of ($\forall$-elim$'$) followed by a use of ($\subseteq$): if σ is of the form $\forall \bar{\beta}$ **with** $C . \rho'$ then we have

$$\cfrac{\cfrac{x : \forall \bar{\beta} \textbf{ with } C . \rho' \ \epsilon\ A' \cup \{x : \sigma\} \qquad A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}]}{A' \cup \{x : \sigma\} \vdash' x : \rho'[\bar{\beta} := \bar{\pi}]} \qquad A' \cup \{x : \sigma\} \vdash' \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho}{A' \cup \{x : \sigma\} \vdash' x : \rho}$$

It is evident that x does not occur in C (if x occurs in C, then the derivation of $A' \cup \{x : \sigma\} \vdash' C[\bar{\beta} := \bar{\pi}]$ will be infinitely high), so by Corollary 3.12 applied to each member of $C[\bar{\beta} := \bar{\pi}]$, it follows that $A' \vdash C[\bar{\beta} := \bar{\pi}]$. Also, by Lemma 3.2, $A' \vdash \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho$. Therefore the derivation $A' \vdash e' : \forall\bar{\beta}$ **with** $C \,.\, \rho'$ may be extended using ($\forall$-elim) and ($\subseteq$):

$$\cfrac{\cfrac{A' \vdash e' : \forall\bar{\beta}\ \textbf{with}\ C \,.\, \rho' \qquad A' \vdash C[\bar{\beta} := \bar{\pi}]}{A' \vdash e' : \rho'[\bar{\beta} := \bar{\pi}]} \qquad A' \vdash \rho'[\bar{\beta} := \bar{\pi}] \subseteq \rho}{A' \vdash e' : \rho}$$

So by induction there exists a substitution T_3 such that

1. $S = S_1 T_3$, except on new type variables of $W_{os}(A, e')$,
2. $AS \vdash B_1 T_3$, and
3. $AS \vdash \tau_1 T_3 \subseteq \rho$.

By 1 and 2 above, we have $(AS_1)T_3 \vdash B_1 T_3$, so by Lemma 3.24 it follows that there exists a substitution T_4 such that

1. $T_3 = S_2 T_4$,
2. $AS_1 T_3 \vdash B_1' T_4$, and
3. $\sigma_1 T_4 \geq_{AS_1 T_3} \tau_1 T_3$.

Now,

$$\begin{array}{ll} & AS_1 S_2 T_2 \\ = & \qquad \ll \text{by first use of Lemma 3.24 above} \gg \\ & AS_1 T_1 \\ = & \qquad \ll \text{by part 1 of the first use of induction above} \gg \\ & AS \\ = & \qquad \ll \text{by part 1 of the second use of induction above} \gg \\ & AS_1 T_3 \\ = & \qquad \ll \text{by second use of Lemma 3.24 above} \gg \\ & AS_1 S_2 T_4 \end{array}$$

Hence T_2 and T_4 are equal when restricted to the free variables of $AS_1 S_2$. Since, by Lemma 3.24, $fv(\sigma_1) \subseteq fv(AS_1 S_2)$, it follows that $\sigma_1 T_2 = \sigma_1 T_4$.

So, by part 3 of the second use of Lemma 3.24 above,

$$\sigma_1 T_2 \geq_{AS} \tau_1 T_3.$$

Write $\sigma_1 T_2$ in the form $\forall\bar{\alpha}$ **with** $D \,.\, \phi$. Then there exists a substitution $[\bar{\alpha} := \bar{\psi}]$ such that

$$AS \vdash D[\bar{\alpha} := \bar{\psi}]$$

and

$$AS \vdash \phi[\bar{\alpha} := \bar{\psi}] \subseteq \tau_1 T_3.$$

Using Corollary 3.13 and Lemma 3.3, then, we have the following derivation:

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \forall\bar{\alpha} \textbf{ with } D \,.\, \phi$$

by (hypoth),

$$AS \cup \{x : \sigma_1 T_2\} \vdash D[\bar{\alpha} := \bar{\psi}]$$

by above,

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \phi[\bar{\alpha} := \bar{\psi}]$$

by ($\forall$ -elim),

$$AS \cup \{x : \sigma_1 T_2\} \vdash \phi[\bar{\alpha} := \bar{\psi}] \subseteq \tau_1 T_3$$

by above,

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \tau_1 T_3$$

by ($\subseteq$),

$$AS \cup \{x : \sigma_1 T_2\} \vdash \tau_1 T_3 \subseteq \rho$$

by part 3 of the second use of induction above, and finally

$$AS \cup \{x : \sigma_1 T_2\} \vdash x : \rho$$

by ($\subseteq$). This completes that proof that $AS \cup \{x : \sigma_1 T_2\} \succeq A' \cup \{x : \sigma\}$, which is the same as

$$(AS_1 S_2 \cup \{x : \sigma_1\})T_2 \succeq A' \cup \{x : \sigma\}.$$

Because $A' \vdash e' : \sigma$ and A' has satisfiable constraints, it is easy to see that the constraints of σ are satisfiable with respect to A'. Hence by Lemma 3.18, $A' \cup \{x : \sigma\}$ has satisfiable constraints.

Finally, $AS_1 S_2 \cup \{x : \sigma_1\}$ has acceptable inclusions.

This allows us to apply the induction hypothesis a third time, showing that $(S_3, B_2, \tau_2) = W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$ succeeds and that there exists T_5 such that

1. $T_2 = S_3 T_5$, except on new variables of $W_{os}(AS_1 S_2 \cup \{x : \sigma_1\}, e'')$,
2. $AS \cup \{x : \sigma_1 T_2\} \vdash B_2 T_5$, and
3. $AS \cup \{x : \sigma_1 T_2\} \vdash \tau_2 T_5 \subseteq \tau'$.

So $(S_0, B_0, \tau_0) = W_{os}(A, \textbf{let } x = e' \textbf{ in } e'')$ succeeds with $S_0 = S_1 S_2 S_3$, $B_0 = B_1' S_3 \cup B_2$, and $\tau_0 = \tau_2$.

Let T be T_5. Then

1. Observe that

$$
\begin{array}{ll}
 & S_0T \\
= & \quad \ll \text{definition} \gg \\
 & S_1S_2S_3T_5 \\
= & \quad \ll \text{part 1 of third use of induction; the new variables} \gg \\
 & \quad \ll \text{of } W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'') \text{ don't occur in } S_1S_2 \gg \\
 & S_1S_2T_2, \text{ except on new variables of } W_{os}(AS_1S_2 \cup \{x : \sigma_1\}, e'') \\
= & \quad \ll \text{by the first use of Lemma 3.24 above} \gg \\
 & S_1T_1 \\
= & \quad \ll \text{by part 1 of first use of induction} \gg \\
 & S, \text{ except on new variables of } W_{os}(A, e').
\end{array}
$$

So $S_0T = S$, except on new variables of $W_{os}(A, \mathbf{let}\ x = e'\ \mathbf{in}\ e'')$.

2. Next,

$$
\begin{array}{ll}
 & B_0T \\
= & \quad \ll \text{definition} \gg \\
 & B_1'S_3T_5 \cup B_2T_5 \\
= & \quad \ll \text{by part 1 of third use of induction and Lemma 3.25} \gg \\
 & B_1'T_2 \cup B_2T_5
\end{array}
$$

By the first use of Lemma 3.24 above, $AS \vdash B_1'T_2$. By part 2 of the third use of induction above, $AS \cup \{x : \sigma_1T_2\} \vdash B_2T_5$. By Theorem 3.23, every identifier occurring in B_1 occurs in A. So, since x does not occur in A, x does not occur in B_1. By Lemma 3.22, every identifier occurring in the constraints of σ_1 occurs in B_1. Hence x does not occur in the constraints of σ_1. By Theorem 3.23, every identifier occurring in B_2 is overloaded in $AS_1S_2 \cup \{x : \sigma_1\}$ or occurs in some constraint of some assumption in $AS_1S_2 \cup \{x : \sigma_1\}$. Since x does not occur in AS_1S_2 or in the constraints of σ_1, it follows that x does not occur in B_2. Hence Corollary 3.12 and Lemma 3.2 may be applied to each member of B_2T_5, yielding $AS \vdash B_2T_5$. So $AS \vdash B_0T$.

3. Now, $\tau_0T = \tau_2T_5$ and by part 3 of the third use of induction above and by Lemma 3.2, we have $AS \vdash \tau_0T \subseteq \tau'$. Also since $A' \vdash \tau' \subseteq \tau$ and since $AS \succeq A'$, it follows that $AS \vdash \tau' \subseteq \tau$. So by (trans), $AS \vdash \tau_0T \subseteq \tau$.

QED

Theorem 3.27 gives as an immediate corollary the completeness of W_{os}:

Corollary 3.28 *If $AS \vdash e : \tau$, AS has satisfiable constraints, and A has acceptable inclusions, then $(S_0, B_0, \tau_0) = W_{os}(A, e)$ succeeds and there exists a substitution T such that*

1. $S = S_0 T$, *except on new type variables of* $W_{os}(A, e)$,
2. $AS \vdash B_0 T$, *and*
3. $AS \vdash \tau_0 T \subseteq \tau$.

Finally, we get the following principal typing result:

Corollary 3.29 *Let* A *be an assumption set with satisfiable constraints, acceptable inclusions, and no free type variables. If* e *is well typed with respect to* A, *then* $(S, B, \tau) = W_{os}(A, e)$ *succeeds,* $(S', B', \sigma) = close(A, B, \tau)$ *succeeds, and the typing* $A \vdash e : \sigma$ *is principal.*

3.5 An Example Assumption Set

In this section we define an example assumption set, A_0, which will be used in a number of examples in the remainder of the thesis. A_0 is given in Figure 3.3.

We make a few remarks about A_0.

- A_0 has satisfiable constraints, acceptable inclusions, and no free type variables, so W_{os} can be used to infer principal types with respect to A_0.
- Type *matrix* is intended to denote the set of all real 2×2 matrices. The operators $+$, $-$ and $*$ denote matrix addition, subtraction, and multiplication respectively. 0 denotes the zero matrix and 1 denotes the identity matrix.
- Because of the undecidability of equality of functions, it is common to give $=$ a weaker type then we have given. Using overloading, one can give $=$ precisely those types for which equality is decidable.[2] Whether applying $=$ to functions should be viewed as a type error or not is debatable.
- The presence of $int \subseteq real$ makes fewer overloadings necessary. In particular, it is not necessary to include the overloadings $0 : real$ and $1 : real$, since these typings follow from $0 : int$ and $1 : int$. Note however that both $+ \ : int \rightarrow int \rightarrow int$ and $+ \ : real \rightarrow real \rightarrow real$ are still needed. (From $int \subseteq real$ and $+ \ : real \rightarrow real \rightarrow real$, it follows that $+ : int \rightarrow int \rightarrow real$, but it does not follow that $+ : int \rightarrow int \rightarrow int$.)

[2] For example, equality is decidable on sequences provided that it is decidable on the type of the elements of the sequences. This fact is reflected by the typing

$$= : \forall\alpha \text{ with } = : \alpha \rightarrow \alpha \rightarrow bool \,.\, seq(\alpha) \rightarrow seq(\alpha) \rightarrow bool \,.$$

$$A_0 = \left\{ \begin{array}{l}
int \subseteq real, \\
fix : \forall\alpha.(\alpha \to \alpha) \to \alpha, \\
if : \forall\alpha.bool \to \alpha \to \alpha \to \alpha, \\
= : \forall\alpha.\alpha \to \alpha \to bool, \\
0 : int, \quad 0 : matrix, \\
1 : int, \quad 1 : matrix, \\
2 : int, \\
\pi : real, \\
+ \ : int \to int \to int, \quad + \ : real \to real \to real, \\
+ \ : matrix \to matrix \to matrix, \\
- \ : int \to int \to int, \quad - \ : real \to real \to real, \\
- \ : matrix \to matrix \to matrix, \\
* \ : int \to int \to int, \quad * \ : real \to real \to real, \\
* \ : matrix \to matrix \to matrix, \\
\div \ : int \to int \to int, \\
/ \ : real \to real \to real, \\
even? : int \to bool, \\
true : bool, \quad false : bool, \\
cons : \forall\alpha.\alpha \to seq(\alpha) \to seq(\alpha), \\
car : \forall\alpha.seq(\alpha) \to \alpha, \\
cdr : \forall\alpha.seq(\alpha) \to seq(\alpha), \\
nil : \forall\alpha.seq(\alpha), \\
null? : \forall\alpha.seq(\alpha) \to bool, \\
\leq : real \to real \to bool, \quad \leq : char \to char \to bool, \\
\leq : \forall\alpha \ \mathbf{with} \ \leq : \alpha \to \alpha \to bool \, . \, seq(\alpha) \to seq(\alpha) \to bool
\end{array} \right\}$$

Figure 3.3: Assumption Set A_0

- We use two different symbols for division: $\div$ denotes truncating integer division; / denotes real division. One might propose instead that / be overloaded to denote both truncating integer division and real division, but it would be a design mistake to do so. For consider the expression $3/2$ as a real value. Since 3 and 2 are integers, we could use truncating integer division, yielding the integer 1, which gives the real value 1. But we could also take 3 and 2 as reals and use real division, yielding the real value 1.5, so the expression would be ambiguous. Reynolds discusses these design issues in [Rey80].

3.6 Type Simplification

Let *lexicographic* be the program of Figure 2.4 for comparing sequences lexicographically. As shown in Corollary 3.29, the computation

$$\begin{aligned}(S, B, \tau) &= W_{os}(A_0, \mathit{lexicographic});\\ (S', B', \sigma) &= \mathit{close}(A_0, B, \tau).\end{aligned}$$

produces a principal type σ for *lexicographic*. But if we use the simple *close* of Figure 2.3 we discover, to our horror, that we obtain the principal type

$$\forall \alpha, \gamma, \zeta, \varepsilon, \delta, \theta, \eta, \lambda, \kappa, \mu, \nu, \xi, \pi, \rho, \sigma, \iota, \tau, \upsilon, \phi \textbf{ with}$$
$$\left\{\begin{array}{l} \gamma \subseteq \mathit{seq}(\zeta),\ \mathit{bool} \subseteq \varepsilon,\ \delta \subseteq \mathit{seq}(\theta),\ \mathit{bool} \subseteq \eta,\ \gamma \subseteq \mathit{seq}(\lambda), \\ \lambda \subseteq \kappa,\ \delta \subseteq \mathit{seq}(\mu),\ \mu \subseteq \kappa,\ \gamma \subseteq \mathit{seq}(\nu),\ \mathit{seq}(\nu) \subseteq \xi, \\ \delta \subseteq \mathit{seq}(\pi),\ \mathit{seq}(\pi) \subseteq \rho,\ \sigma \subseteq \iota,\ \leq : \tau \to \tau \to \mathit{bool}, \\ \gamma \subseteq \mathit{seq}(\upsilon),\ \upsilon \subseteq \tau,\ \delta \subseteq \mathit{seq}(\phi),\ \phi \subseteq \tau,\ \mathit{bool} \subseteq \iota,\ \iota \subseteq \eta, \\ \eta \subseteq \varepsilon,\ (\xi \to \rho \to \sigma) \to (\gamma \to \delta \to \varepsilon) \subseteq (\alpha \to \alpha) \end{array}\right\} . \alpha$$

Such a type is clearly useless to a programmer, so, as a practical matter, it is essential for *close* to simplify the types that it produces.

3.6.1 An Overview of Type Simplification

We begin with an informal overview of the simplification process, showing how type simplification works on *lexicographic*.

The call $W_{os}(A_0, \mathit{lexicographic})$ returns $([\beta, o := \xi \to \rho \to \sigma, \rho \to \sigma],\ B,\ \alpha)$, where

$$B = \left\{\begin{array}{l} \gamma \subseteq \mathit{seq}(\zeta),\ \mathit{bool} \subseteq \mathit{bool},\ \mathit{bool} \subseteq \varepsilon,\ \delta \subseteq \mathit{seq}(\theta),\ \mathit{bool} \subseteq \eta,\ \gamma \subseteq \mathit{seq}(\lambda), \\ \lambda \subseteq \kappa,\ \delta \subseteq \mathit{seq}(\mu),\ \mu \subseteq \kappa,\ \gamma \subseteq \mathit{seq}(\nu),\ \mathit{seq}(\nu) \subseteq \xi,\ \delta \subseteq \mathit{seq}(\pi), \\ \mathit{seq}(\pi) \subseteq \rho,\ \sigma \subseteq \iota,\ \leq : \tau \to \tau \to \mathit{bool},\ \gamma \subseteq \mathit{seq}(\upsilon),\ \upsilon \subseteq \tau,\ \delta \subseteq \mathit{seq}(\phi), \\ \phi \subseteq \tau,\ \mathit{bool} \subseteq \iota,\ \iota \subseteq \eta,\ \eta \subseteq \varepsilon,\ (\xi \to \rho \to \sigma) \to (\gamma \to \delta \to \varepsilon) \subseteq (\alpha \to \alpha) \end{array}\right\}$$

This means that for any instantiation S of the variables in B such that $A_0 \vdash BS$, *lexicographic* has type αS. The problem is that B is so complicated that it is not at all clear what the possible satisfying instantiations are. But perhaps we can make (generally partial) instantiations for some of the variables in B that are in a certain sense optimal, in that they yield a simpler, yet equivalent, type. This is the basic idea behind type simplification.

There are two ways for an instantiation to be optimal. First, an instantiation of some of the variables in B is clearly optimal if it is 'forced', in the sense that those variables can be instantiated in only one way if B is to be satisfied. The second way for an instantiation to be optimal is more subtle. Suppose that there is an instantiation T that makes B no harder to satisfy and that makes the body (in this example, α) no larger. More precisely, suppose that $A_0 \cup B \vdash BT$ and $A_0 \cup B \vdash \alpha T \subseteq \alpha$. Then by using rule ($\subseteq$), BT and αT can produce the same types as can B and α, so the instantiation T is optimal. We now look at how these two kinds of optimal instantiation apply in the case of *lexicographic.*

We begin by discovering a number of forced instantiations. Now, A_0 has acceptable inclusions, so by Lemma 3.19, A_0 can prove only inclusions of the form $\tau \subseteq \tau'$, where τ and τ' have the same shape. So the constraint $\gamma \subseteq seq(\zeta)$ in B can be satisfied only if γ is instantiated to some type of the form $seq(\chi)$; the partial instantiation $[\gamma := seq(\chi)]$ is forced. There is a procedure, *shape-unifier*, that finds the most general substitution U such that all the inclusions in BU are between types of the same shape. In this case, U is

$$\left[\begin{array}{l} \gamma := seq(\chi), \\ \delta := seq(\psi), \\ \xi := seq(\omega), \\ \rho := seq(\alpha_1), \\ \alpha := seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow \beta_1 \end{array}\right]$$

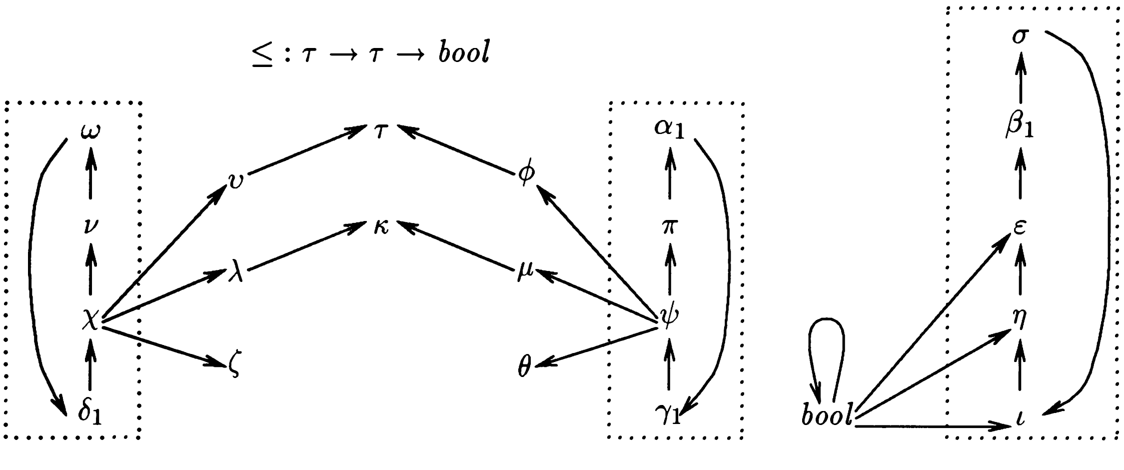
The instantiations in U are all forced by shape considerations; making these forced instantiations produces the constraint set

$$\left\{\begin{array}{l} seq(\chi) \subseteq seq(\zeta),\ bool \subseteq bool,\ bool \subseteq \varepsilon,\ seq(\psi) \subseteq seq(\theta), \\ bool \subseteq \eta,\ seq(\chi) \subseteq seq(\lambda),\ \lambda \subseteq \kappa,\ seq(\psi) \subseteq seq(\mu), \\ \mu \subseteq \kappa,\ seq(\chi) \subseteq seq(\nu),\ seq(\nu) \subseteq seq(\omega),\ seq(\psi) \subseteq seq(\pi), \\ seq(\pi) \subseteq seq(\alpha_1),\ \sigma \subseteq \iota,\ \leq : \tau \rightarrow \tau \rightarrow bool,\ seq(\chi) \subseteq seq(\upsilon), \\ \upsilon \subseteq \tau,\ seq(\psi) \subseteq seq(\phi),\ \phi \subseteq \tau,\ bool \subseteq \iota,\ \iota \subseteq \eta,\ \eta \subseteq \varepsilon, \\ (seq(\omega) \rightarrow seq(\alpha_1) \rightarrow \sigma) \rightarrow (seq(\chi) \rightarrow seq(\psi) \rightarrow \varepsilon) \subseteq \\ (seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow \beta_1) \rightarrow (seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow \beta_1) \end{array}\right\}$$

and the body

$$seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow \beta_1.$$

We have made progress; we can now see that *lexicographic* is a function that takes two sequences as input and returns some output.



body: $seq(\delta_1) \to seq(\gamma_1) \to \beta_1$

Figure 3.4: Atomic Inclusions for *lexicographic*

The new constraint set contains the complicated inclusion

$$(seq(\omega) \to seq(\alpha_1) \to \sigma) \to (seq(\chi) \to seq(\psi) \to \varepsilon) \subseteq \\ (seq(\delta_1) \to seq(\gamma_1) \to \beta_1) \to (seq(\delta_1) \to seq(\gamma_1) \to \beta_1).$$

However, by Lemma 3.20 and rule $((-) \to (+))$, we have

$$A_0 \vdash \tau \to \rho \subseteq \tau' \to \rho'$$

if and only if

$$A_0 \vdash \tau' \subseteq \tau \text{ and } A_0 \vdash \rho \subseteq \rho'.$$

Hence our complicated inclusion is equivalent to the simpler inclusions

$$(seq(\delta_1) \to seq(\gamma_1) \to \beta_1) \subseteq (seq(\omega) \to seq(\alpha_1) \to \sigma)$$

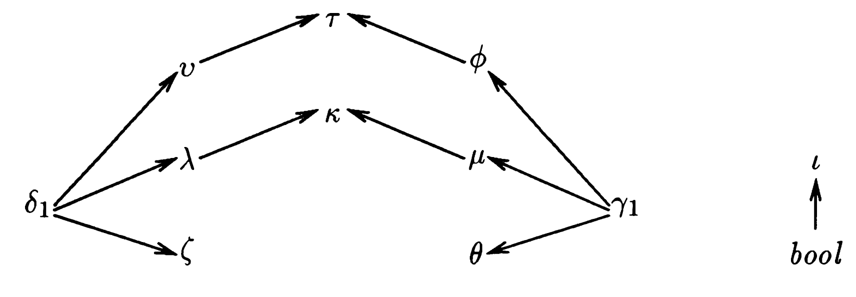
and

$$(seq(\chi) \to seq(\psi) \to \varepsilon) \subseteq (seq(\delta_1) \to seq(\gamma_1) \to \beta_1).$$

These simpler inclusions, too, can be broken down into still simpler inclusions. The result of this process is to convert the constraint set into an equivalent set containing only *atomic inclusions.* The result of this transformation is shown graphically in Figure 3.4, where an inclusion $\tau_1 \subseteq \tau_2$ is denoted by drawing an arrow from τ_1 to τ_2. Below the representation of the constraint set we give the body.

Now notice that the constraint set in Figure 3.4 contains cycles; for example ω and ν lie on a common cycle. This means that if S is any instantiation that

$\leq : \tau \rightarrow \tau \rightarrow \mathit{bool}$



body: $\mathit{seq}(\delta_1) \rightarrow \mathit{seq}(\gamma_1) \rightarrow \iota$

Figure 3.5: Collapsed Components of *lexicographic*

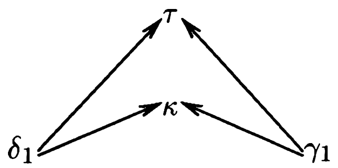
satisfies the constraints, we will have both $A_0 \vdash \omega S \subseteq \nu S$ and $A_0 \vdash \nu S \subseteq \omega S$. Hence, by Lemma 3.21, it follows that $\omega S = \nu S$. In general, any two types within the same strongly connected component must be instantiated in the same way. If a component contains more than one type constant, then, it is unsatisfiable; if it contains exactly one type constant, then all the variables must be instantiated to that type constant; and if it contains only variables, then we may instantiate all the variables in the component to any chosen variable. We have surrounded the strongly connected components of the constraint set with dotted rectangles in Figure 3.4; Figure 3.5 shows the result of collapsing those components and removing any trivial inclusions of the form $\rho \subseteq \rho$ thereby created.

At this point, we are finished making forced instantiations; we turn next to instantiations that are optimal in the second sense described above. These are the monotonicity-based instantiations.

Consider the type $\mathit{bool} \rightarrow \alpha$. By rule $((-) \rightarrow (+))$, this type is *monotonic* in α: as α grows, a larger type is produced. In contrast, the type $\alpha \rightarrow \mathit{bool}$ is *antimonotonic* in α: as α grows, a smaller type is produced. Furthermore, the type $\beta \rightarrow \beta$ is both monotonic and antimonotonic in α: changing α has no effect on it. Finally, the type $\alpha \rightarrow \alpha$ is neither monotonic nor antimonotonic in α: as α grows, incomparable types are produced.

Refer again to Figure 3.5. The body $\mathit{seq}(\delta_1) \rightarrow (\mathit{seq}(\gamma_1) \rightarrow \iota)$ is antimonotonic in δ_1 and γ_1 and monotonic in ι. This means that to make the body smaller, we must boost δ_1 and γ_1 and shrink ι. Notice that ι has just one type smaller than it, namely bool. This means that if we instantiate ι to bool, all the inclusions involving ι will be satisfied, and ι will be made smaller. Hence the instantiation $[\iota := \mathit{bool}]$ is optimal. The cases of δ_1 and γ_1 are trickier—they both have more than one successor, so it does not appear that they can

$$\leq : \tau \rightarrow \tau \rightarrow bool$$



body: $seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow bool$

Figure 3.6: Result of Shrinking ι, υ, λ, ζ, ϕ, μ, and θ

be boosted. If we boost δ_1 to υ, for example, then the inclusions $\delta_1 \subseteq \lambda$ and $\delta_1 \subseteq \zeta$ may be violated.

The variables υ, λ, ζ, ϕ, μ and θ, however, do have unique predecessors. Since the body is monotonic (as well as antimonotonic) in all of these variables, we may safely shrink them all to their unique predecessors. The result of these instantiations is shown in Figure 3.6.

Now we are left with a constraint graph in which no node has a unique predecessor or successor. We are still not done, however. Because the body $seq(\delta_1) \rightarrow seq(\gamma_1) \rightarrow bool$ is both monotonic and antimonotonic in κ, we can instantiate κ arbitrarily, even to an incomparable type, without making the body grow. It happens that the instantiation $[\kappa := \tau]$ satisfies the two inclusions $\delta_1 \subseteq \kappa$ and $\gamma_1 \subseteq \kappa$. Hence we may safely instantiate κ to τ.

Observe that we could have tried instead to instantiate τ to κ, but this would have violated the overloading constraint $\leq : \tau \rightarrow \tau \rightarrow bool$. This brings up a point not yet mentioned: before performing a monotonicity-based instantiation of a variable, we must check that all overloading constraints involving that variable are satisfied.

At this point, δ_1 and γ_1 have a unique successor, τ, so they may now be boosted. This leaves us with the constraint set

$$\{\leq : \tau \rightarrow \tau \rightarrow bool\}$$

and the body

$$seq(\tau) \rightarrow seq(\tau) \rightarrow bool.$$

At last the simplification process is finished; we can now use the simple *close* of Figure 2.3 to produce the principal type

$$\forall \tau \textbf{ with } \leq : \tau \rightarrow \tau \rightarrow bool \,.\, seq(\tau) \rightarrow seq(\tau) \rightarrow bool,$$

which happens to be the same as the type obtained in Chapter 2 without subtyping.

In the next three subsections, we present in more detail the transformation to atomic inclusions, the collapsing of strongly connected components, and the monotonicity-based instantiations. We then present the complete function *close* and prove that it satisfies Lemmas 3.22 and 3.24.

3.6.2 Transformation to Atomic Inclusions

In this subsection we describe the part of $close(A, B, \tau)$ that transforms the inclusions in B to atomic inclusions. We begin by developing the theory of *shape unification.*

First we recall certain definitions about ordinary unification. Given a set $\mathcal{P}$ of pairs (τ, τ'), we say that substitution U *unifies* $\mathcal{P}$ if for every pair (τ, τ') in $\mathcal{P}$, $\tau U = \tau' U$; if there exists such a U we say that $\mathcal{P}$ is *unifiable.* We say that U is a *most general unifier* for $\mathcal{P}$ if

- U unifies $\mathcal{P}$ and
- any V that unifies $\mathcal{P}$ can be factored into UV', for some V'.

There are efficient algorithms for computing most general unifiers [Kni89].

Now, given a set $\mathcal{P}$ of pairs (τ, τ'), we say that substitution U *shape unifies* $\mathcal{P}$ if for every pair (τ, τ') in $\mathcal{P}$, τU and $\tau' U$ have the same shape; in this case we say that $\mathcal{P}$ is *shape unifiable.* We say that U is a *most general shape unifier* for $\mathcal{P}$ if

- U shape unifies $\mathcal{P}$ and
- any V that shape unifies $\mathcal{P}$ can be factored into UV', for some V'.

For example,

$$\{((\alpha \to \beta) \to \alpha,\ \ \gamma \to seq(\delta))\}$$

has most general shape unifier

$$[\gamma := seq(\varepsilon) \to \zeta,\ \ \alpha := seq(\eta)]$$

and

$$\{(seq(\alpha) \to \beta,\ \ \beta \to \alpha)\}$$

has no shape unifier.

Computing most general shape unifiers turns out to be more subtle than computing most general unifiers.[3] Fuh and Mishra [FM90] give a rather complicated algorithm. We can, however, give a simpler algorithm with the help of the following lemma, which is due to Bard Bloom.

[3] The difficulty is that, unlike equality, the 'same shape' relation is not preserved under substitution; hence building up a shape unifier from partial solutions is more awkward.

shape-unifier($\mathcal{P}$):

if $\mathcal{P}_c$ is not unifiable, *fail*;
$\mathcal{Q} := \mathcal{P}$;
$S := []$;
while $\mathcal{Q}$ does not contain only atomic pairs **do**
    choose $p \epsilon \mathcal{Q}$ nonatomic;
    case p:
        $\underline{(\alpha, \tau) \text{ or } (\tau, \alpha):}$
            let $R = [\alpha := \textit{fresh-leaves}(\tau)]$;
                $\mathcal{Q} := \mathcal{Q}R$;
                $S := SR$;
        $\underline{(\chi(\tau_1, \ldots, \tau_n),\ \chi(\tau'_1, \ldots, \tau'_n)):}$
            $\mathcal{Q} := (\mathcal{Q} - \{p\}) \cup \{(\tau_1, \tau'_1), \ldots, (\tau_n, \tau'_n)\}$;
return S, restricted to the variables of $\mathcal{P}$.

Figure 3.7: Function *shape-unifier*

Lemma 3.30 *Let $\mathcal{P}_c$ denote the result of replacing all occurrences of type constants in $\mathcal{P}$ by the type constant c. Then $\mathcal{P}$ is shape unifiable if and only if $\mathcal{P}_c$ is unifiable.*

Proof: ($\Leftarrow$) If $\tau_c U = \tau'_c U$, then τU and $\tau' U$ have the same shape.

($\Rightarrow$) If τ and τ' are shape unifiable, then they have a shape unifier $[\bar{\alpha} := \bar{\pi}]$ where, without loss of generality, $\bar{\pi}$ has no occurrences of type constants. Hence $\tau_c[\bar{\alpha} := \bar{\pi}]S = \tau'_c[\bar{\alpha} := \bar{\pi}]S$, where S is a substitution that maps all type variables to c. **QED**

Lemma 3.30 gives us a way to use ordinary unification to test whether a set $\mathcal{P}$ of pairs is shape unifiable, allowing us to use an algorithm for constructing most general shape unifiers that works only when $\mathcal{P}$ is in fact shape unifiable. Algorithm *shape-unifier* is given in Figure 3.7. The call *fresh-leaves*(τ) in *shape-unifier* returns τ with all occurrences of type variables and type constants replaced by new type variables; for example, *fresh-leaves*($seq(\alpha) \rightarrow int \rightarrow \alpha$) is $seq(\beta) \rightarrow \gamma \rightarrow \delta$, where β, γ, and δ are new type variables. The important properties of *shape-unifier* are given in the following lemma.

Lemma 3.31 *If $\mathcal{P}$ is shape unifiable, then shape-unifier($\mathcal{P}$) succeeds and returns a most general shape unifier for $\mathcal{P}$; otherwise, shape-unifier($\mathcal{P}$) fails.*

Proof: The loop invariant is

1. If U shape unifies $\mathcal{P}$, then $U = SU'$ for some U' and U' shape unifies $\mathcal{Q}$.

2. If V shape unifies $\mathcal{Q}$, then SV shape unifies $\mathcal{P}$.

$atomic\text{-}inclusions(B) = B$, if B contains only atomic inclusions;
$atomic\text{-}inclusions(B \cup \{\tau \to \rho \subseteq \tau' \to \rho'\}) =$
$\quad atomic\text{-}inclusions(B \cup \{\tau' \subseteq \tau,\ \rho \subseteq \rho'\})$;
$atomic\text{-}inclusions(B \cup \{seq(\tau) \subseteq seq(\tau')\}) =$
$\quad atomic\text{-}inclusions(B \cup \{\tau \subseteq \tau'\})$;

Figure 3.8: Function *atomic-inclusions*

From 1, if $\mathcal{P}$ has a shape unifier U, then $U = SU'$; it follows that S cannot grow without bound and hence the loop terminates. **QED**

One might wonder whether the test of the unifiability of $\mathcal{P}_c$ at the start of *shape-unifier* could be made unnecessary by adding an "occurs" check to the (α, τ) case that *fails* if α occurs in τ, and by adding an "else" case that *fails* when p is of the form $(\chi(\ldots), \chi'(\ldots))$ and $\chi \neq \chi'$. Unfortunately, this is not good enough: if $\mathcal{P}$ is $\{(seq(\alpha) \to \beta,\ \ \beta \to \alpha)\}$, then even with the proposed tests the **while** loop will not terminate.

Now we turn to the problem of converting a set of inclusions between types of the same shape into an equivalent set of atomic inclusions. This is accomplished by function *atomic-inclusions*, given in Figure 3.8. Each clause in the definition of *atomic-inclusions* takes care of one type constructor and is based on the monotonicities of that constructor in each argument position. The soundness and completeness of *atomic-inclusions* are shown in the following lemmas.

Lemma 3.32 *If* $C = atomic\text{-}inclusions(B)$ *succeeds, then* C *contains only atomic inclusions and* $C \vdash B$.

Proof: This follows easily from the rules $((-) \to (+))$ and $(seq(+))$. **QED**

Lemma 3.33 *If* B *contains only inclusions between types with the same shape, then* $atomic\text{-}inclusions(B)$ *succeeds. Furthermore, if* A *contains only atomic inclusions, then for any* S, $A \vdash BS$ *implies* $A \vdash (atomic\text{-}inclusions(B))S$.

Proof: Follows from Lemma 3.20. **QED**

Given *shape-unifier* and *atomic-inclusions*, we can now state precisely the *transformation to atomic inclusions*, which is the first part of $close(A, B, \tau)$:

let A_{ci} be the constant inclusions in A,
$\quad B_i$ be the inclusions in B,
$\quad B_t$ be the typings in B;
let $U = shape\text{-}unifier(\{(\phi, \phi') \mid (\phi \subseteq \phi') \epsilon B_i\})$;
let $C_i = atomic\text{-}inclusions(B_i U) \cup A_{ci}$,
$\quad C_t = B_t U$;

(The reason for including A_{ci} in C_i will be given later, in the discussion of monotonicity-based instantiations.) The soundness and completeness of the transformation are shown in the following two lemmas.

Lemma 3.34 *If the transformation to atomic inclusions succeeds, then C_i contains only atomic inclusions. Also, if $A \cup B \vdash e : \tau$, then $AU \cup (C_i \cup C_t) \vdash e : \tau U$ and $id(C_t) = id(B_t)$.*

Proof: By Lemma 3.32 and the fact that A_{ci} contains only constant inclusions, C_i contains only atomic inclusions.

By Lemma 3.5, $AU \cup B_iU \cup B_tU \vdash e : \tau U$. By Lemma 3.32, $C_i \vdash B_iU$, and so by Lemmas 3.3, 3.13, and 3.14, $AU \cup C_i \cup B_tU \vdash e : \tau U$. Since $C_t = B_tU$, we have $AU \cup (C_i \cup C_t) \vdash e : \tau U$ and also $id(C_t) = id(B_t)$. **QED**

Lemma 3.35 *If A has acceptable inclusions and $AR \vdash BR$, then the transformation to atomic inclusions succeeds and there exists a substitution R' such that $R = UR'$ and $AUR' \vdash (C_i \cup C_t)R'$.*

Proof: By Lemma 3.2 and the fact that A contains only constant inclusions we have $A_{ci} \vdash B_iR$. So by Lemma 3.19, R shape unifies $\{(\phi, \phi') \mid (\phi \subseteq \phi') \epsilon B_i\}$. Hence by Lemma 3.31, $U = \textit{shape-unifier}(\{(\phi, \phi') \mid (\phi \subseteq \phi') \epsilon B_i\})$ succeeds and U is a most general shape unifier for $\{(\phi, \phi') \mid (\phi \subseteq \phi') \epsilon B_i\}$, so there exists R' such that $R = UR'$.

Now B_iU contains only inclusions between types of the same shape, so by Lemma 3.33, *atomic-inclusions*(B_iU) succeeds. Furthermore, since $A_{ci} \vdash (B_iU)R'$, by Lemma 3.33 we have $A_{ci} \vdash (\textit{atomic-inclusions}(B_iU))R'$, and hence $AUR' \vdash (C_i \cup C_t)R'$. **QED**

3.6.3 Collapsing Strongly Connected Components

As discussed in the overview of type simplification, types belonging to the same strongly connected component of C_i must be instantiated in the same way, so each strongly connected component can be collapsed to a single type. To this end, *component-collapser*(C_i) returns the most general substitution that collapses the strongly connected components of C_i. Function *component-collapser* is given in Figure 3.9. The strongly connected components of G may be found in linear time using depth-first search [AHU74].

Let *nontrivial-inclusions*(B) be the set of inclusions in B not of the form $\pi \subseteq \pi$. Now if $V = \textit{component-collapser}(C_i)$, then the only cycles in C_iV are trivial inclusions; hence *nontrivial-inclusions*(C_iV) is acyclic. An acyclic graph can be further simplified by taking its *transitive reduction* [AGU72]. The transitive reduction of an acyclic graph G is the unique minimal graph having the same transitive closure as G; it may be found by deleting redundant edges *component-collapser*(C):

1. Build a directed graph G with a vertex for each type in C and with an edge $\tau \longrightarrow \tau'$ whenever $(\tau \subseteq \tau')\epsilon C$.

2. Find the strongly connected components of G.

3. $V := [\,]$;
   for each strongly connected component $\mathcal{C}$ of G **do**
       if $\mathcal{C}$ contains more then one type **then**
           if $\mathcal{C}$ contains more than one type constant, *fail;*
           let $\alpha_1, \ldots, \alpha_n$ be the type variables in $\mathcal{C}$;
           if $\mathcal{C}$ contains a type constant c **then**
               $V := V[\alpha_1, \alpha_2, \ldots, \alpha_n := c, c, \ldots, c]$
           else
               $V := V[\alpha_2, \alpha_3, \ldots, \alpha_n := \alpha_1, \alpha_1, \ldots, \alpha_1]$;

4. Return V.

Figure 3.9: Function *component-collapser*

from G using an algorithm that is essentially the inverse of Warshall's transitive closure algorithm [GMvdSU89].

It will be seen in the discussion of monotonicity-based instantiation that the use of the transitive reduction has significance beyond mere simplification.

We can now state precisely the next part of $close(A, B, \tau)$, which is the *collapsing of strongly connected components*:

$$\begin{aligned}
&\text{let } V = \textit{component-collapser}(C_i);\\
&\text{let } S = UV,\\
&\quad D_i = \textit{transitive-reduction}(\textit{nontrivial-inclusions}(C_i V)),\\
&\quad D_t = C_t V;
\end{aligned}$$

The soundness and completeness of these statements are shown in the following two lemmas, whose proofs are straightforward.

Lemma 3.36 *If the collapsing of strongly connected components succeeds then* D_i *is acyclic and reduced; that is,* $\textit{transitive-reduction}(D_i) = D_i$. *Also, if* $AU \cup (C_i \cup C_t) \vdash e : \tau U$, *then* $AS \cup (D_i \cup D_t) \vdash e : \tau S$ *and* $id(D_t) = id(C_t)$.

Lemma 3.37 *If* A *has acceptable inclusions and* $AUR' \vdash (C_i \cup C_t)R'$, *then the collapsing of strongly connected components succeeds and there exists a substitution* R'' *such that* $R' = VR''$ *and* $ASR'' \vdash (D_i \cup D_t)R''$.

3.6.4 Monotonicity-based Instantiations

The final simplifications performed in $close(A, B, \tau)$ are the monotonicity-based instantiations. Monotonicity-based simplification was earlier explored by Curtis [Cur90] and by Fuh and Mishra [FM89].

Monotonicity-based instantiations, in contrast to the instantiations made during shape unification and the collapsing of strongly connected components, are not in general forced. Because of this, there is a restriction on monotonicity-based instantiations: they may be applied only to variables that are not free in AS. This is because the instantiations of type variables free in AS may be forced in subsequent type inference, so to preserve the completeness of W_{os} such variables must not be instantiated unnecessarily.

At this point in the simplification process, we have transformed the original typing $A \cup B \vdash e : \tau$ to the typing $AS \cup (D_i \cup D_t) \vdash e : \tau S$. Now, given a variable α not free in AS, we may (by Lemma 3.5) transform the typing $AS \cup (D_i \cup D_t) \vdash e : \tau S$ to the typing $AS \cup (D_i \cup D_t)[\alpha := \pi] \vdash e : \tau S[\alpha := \pi]$, but we may lose information by doing so. We will not lose information, however, if the instantiation $[\alpha := \pi]$ makes $(D_i \cup D_t)$ no harder to satisfy and makes τS no bigger; precisely we require that

$$AS \cup (D_i \cup D_t) \vdash (D_i \cup D_t)[\alpha := \pi]$$

and

$$AS \cup (D_i \cup D_t) \vdash \tau S[\alpha := \pi] \subseteq \tau S.$$

To see why no information is lost by the transformation in this case, suppose that T is an instantiation of variables not free in AS such that $AS \vdash (D_i \cup D_t)T$, allowing the original typing to produce $AS \vdash e : \tau ST$. Then $AS \vdash (D_i \cup D_t)[\alpha := \pi]T$ and $AS \vdash \tau S[\alpha := \pi]T \subseteq \tau ST$, so the new typing can produce first $AS \vdash e : \tau S[\alpha := \pi]T$ and then $AS \vdash e : \tau ST$ by ($\subseteq$).

We now give the precise formulation of the *monotonicity-based instantiations*:

$E_i := D_i$; $E_t := D_t$; $\rho := \tau S$;
$\bar{\alpha} :=$ variables free in D_i or D_t or τS but not AS;
while there exist α in $\bar{\alpha}$ and π different from α such that
  $AS \cup (E_i \cup E_t) \vdash (E_i \cup E_t)[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}$
do
  $E_i :=$ *transitive-reduction*(*nontrivial-inclusions*($E_i[\alpha := \pi]$));
  $E_t := E_t[\alpha := \pi]$;
  $\rho := \rho[\alpha := \pi]$;
  $\bar{\alpha} := \bar{\alpha} - \alpha$;
od

It is straightforward to show that the **while** loop has the following invariant:

1. If $AS \cup (D_i \cup D_t) \vdash e : \tau S$, then $AS \cup (E_i \cup E_t) \vdash e : \rho$ and $id(E_t) = id(D_t)$.

2. $AS \cup (D_i \cup D_t) \vdash E_i \cup E_t \cup \{\rho \subseteq \tau S\}$.

3. E_i is acyclic and reduced; that is, $\mathit{transitive\text{-}reduction}(E_i) = E_i$.

4. If A has acceptable inclusions, then $AS \cup (E_i \cup E_t) \vdash \phi \subseteq \phi'$ if and only if $E_i \vdash \phi \subseteq \phi'$.

It was to achieve part 4 of the invariant that we included A_{ci} in C_i in the transformation to atomic inclusions.

Notice that the monotonicity-based instantiations always succeed.

The soundness and completeness of the monotonicity-based instantiations are given in the next two lemmas, both of which follow immediately from the loop invariant.

Lemma 3.38 *If* $AS \cup (D_i \cup D_t) \vdash e : \tau S$, *then* $AS \cup (E_i \cup E_t) \vdash e : \rho$ *and* $id(E_t) = id(D_t)$.

Lemma 3.39 *If* $ASR'' \vdash (D_i \cup D_t)R''$ *then* $ASR'' \vdash (E_i \cup E_t)R''$ *and* $ASR'' \vdash \rho R'' \subseteq \tau S R''$.

At this point the only question is how to implement efficiently the guard of the **while** loop: given a variable α in $\bar{\alpha}$, we want to determine whether there is a π not equal to α such that

$$AS \cup (E_i \cup E_t) \vdash (E_i \cup E_t)[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}.$$

We begin by determining under what circumstances

$$AS \cup (E_i \cup E_t) \vdash \rho[\alpha := \pi] \subseteq \rho;$$

the *monotonicity* of ρ in α turns out to be the key.

Definition 3.5 *We say that* ρ *is* monotonic *in* α *if*

- ρ *is atomic,*
- $\rho = \rho_1 \rightarrow \rho_2$, ρ_1 *is antimonotonic in* α, *and* ρ_2 *is monotonic in* α, *or*
- $\rho = seq(\rho_1)$ *and* ρ_1 *is monotonic in* α.

Similarly, ρ *is* antimonotonic *in* α *if*

- ρ *is atomic and* $\rho \neq \alpha$,
- $\rho = \rho_1 \rightarrow \rho_2$, ρ_1 *is monotonic in* α, *and* ρ_2 *is antimonotonic in* α, *or*
- $\rho = seq(\rho_1)$ *and* ρ_1 *is antimonotonic in* α.

Notice that if ρ is both monotonic and antimonotonic in α, then α does not occur in ρ. The importance of monotonicity and antimonotonicity is demonstrated by the following lemma.

Lemma 3.40 *Let A have acceptable inclusions and let α and π be distinct. Then $A \vdash \rho[\alpha := \pi] \subseteq \rho$ if and only if one of the following hold:*

- *ρ is monotonic but not antimonotonic in α and $A \vdash \pi \subseteq \alpha$,*
- *ρ is antimonotonic but not monotonic in α and $A \vdash \alpha \subseteq \pi$, or*
- *ρ is both monotonic and antimonotonic in α.*

Aided by Lemma 3.40, we now return to the implementation of the guard of the **while** loop. Given α in $\bar{\alpha}$, the first step is to check the monotonicity and antimonotonicity of ρ in α; we discuss in turn what to do for each of the four possible outcomes.

If ρ is neither monotonic nor antimonotonic in α, then by Lemma 3.40 there is no π distinct from α such that $AS \cup (E_i \cup E_t) \vdash \rho[\alpha := \pi] \subseteq \rho$. Hence α cannot be instantiated.

If ρ is monotonic but not antimonotonic in α, then α must be shrunk—by Lemma 3.40 we have

$$AS \cup (E_i \cup E_t) \vdash \rho[\alpha := \pi] \subseteq \rho$$

if and only if

$$AS \cup (E_i \cup E_t) \vdash \pi \subseteq \alpha.$$

Now, if A has acceptable inclusions, then by part 4 of the invariant of the **while** loop, this will hold if and only if $E_i \vdash \pi \subseteq \alpha$. Since E_i contains only atomic inclusions, by Lemma 3.19 π must be atomic; hence $E_i \vdash \pi \subseteq \alpha$ if and only if there is a path of length at least 1 from π to α in E_i. This implies that α must have a *predecessor* in E_i; that is, there must be a type ϕ such that $(\phi \subseteq \alpha) \epsilon E_i$, for otherwise, there are no nontrivial paths to α in E_i.

Suppose now that

$$AS \cup (E_i \cup E_t) \vdash E_i[\alpha := \pi].$$

Then $AS \cup (E_i \cup E_t) \vdash \phi \subseteq \pi$, so by part 4 of the invariant of the **while** loop, $E_i \vdash \phi \subseteq \pi$; hence there is a path from ϕ to π in E_i. So we have the following picture:

$$\begin{tikzcd} \alpha & \\ & \pi \arrow[ul, "+"'] \\ \phi \arrow[uu] \arrow[ur, "*"'] & \end{tikzcd}$$

One possibility is that $\pi = \phi$; in fact this is the only possibility! For suppose that $\pi \neq \phi$. Then since E_i is acyclic, neither the path $\phi \xrightarrow{*} \pi$ nor the path $\pi \xrightarrow{+} \alpha$ contains the edge $\phi \longrightarrow \alpha$. Therefore, the edge $\phi \longrightarrow \alpha$ is redundant, contradicting the fact that E_i is reduced, which is part 3 of the invariant of the **while** loop. It follows, then, that if α has more than one predecessor in E_i then there is no π such that

$$AS \cup (E_i \cup E_t) \vdash E_i[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}.$$

If, on the other hand, π is the *unique* predecessor of α in E_i, then one sees readily that

$$AS \cup (E_i \cup E_t) \vdash E_i[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}.$$

This gives us an efficient implementation of the guard of the **while** loop in the case when ρ is monotonic but not antimonotonic in α. First check whether α has a unique predecessor in E_i; if not, α cannot be instantiated.[4] If α has a unique predecessor π, then it suffices to check that $AS \cup (E_i \cup E_t) \vdash E_t[\alpha := \pi]$. (We will discuss the implementation of this check in Section 3.7.)

Next we turn to the case where ρ is antimonotonic but not monotonic in α; in this case α must be boosted. By an analysis just like that of the previous case, one can show that α can be instantiated if and only if it has a unique *successor* π in E_i (that is, there is a unique type π such that $(\alpha \subseteq \pi) \epsilon E_i$) and $AS \cup (E_i \cup E_t) \vdash E_t[\alpha := \pi]$.

Finally, we must consider the case where ρ is both monotonic and antimonotonic in α, which implies that α does not occur in ρ. In this case $AS \cup (E_i \cup E_t) \vdash \rho[\alpha := \pi] \subseteq \rho$ holds for *every* π, so we must do some searching. (Of course, the fact that π can be chosen so freely has benefits as well as costs— it is what makes possible the instantiation of κ in the example of *lexicographic* discussed in the overview.)

If α occurs in E_i, then we can limit the search for π to types occurring in E_i. For suppose that $(\alpha \subseteq \phi) \epsilon E_i$. Then if

$$AS \cup (E_i \cup E_t) \vdash (E_i \cup E_t)[\alpha := \pi],$$

then (by part 4 of the **while** loop invariant) we have $E_i \vdash \pi \subseteq \phi$, so there is a path in E_i from π to ϕ; that is, π must occur in E_i.

If, however, α does not occur in E_i, then it is not clear that the search for π can be bounded at all. Indeed, this case amounts to a search for a satisfying assignment for a set of overloading constraints, which is of course undecidable in general. Two solutions may be considered. First, by limiting overloading we may be able to bound the search. Second, we may limit the search for π arbitrarily,

[4] Of course, later in the simplification process, after other variables have been instantiated, it may become possible to instantiate α.

perhaps to types occurring in E_i; if we do this we will still produce principal types, we just may not produce the simplest possible types. This approach seems to work well in practice.

We conclude the discussion of monotonicity-based instantiation with two examples that show the necessity of taking transitive reductions.

The call

$$W_{os}(A_0, \mathit{if}\ \mathit{true}\ 2\ \ \pi)$$

returns

$$([\,], \{\mathit{bool} \subseteq \mathit{bool}, \mathit{int} \subseteq \alpha, \mathit{real} \subseteq \alpha\}, \alpha),$$

so if A_{ci} were not included in C_i or if E_i were not reduced, then the analysis requiring that α have a unique predecessor would be wrong and we would arrive at the principal type

$$\forall\alpha \ \textbf{with}\ \mathit{int} \subseteq \alpha, \mathit{real} \subseteq \alpha \,.\, \alpha$$

rather than *real*.

The example

$$\mathit{if}\ \mathit{true}\ ((\lambda x.x)2)\ \ \pi$$

shows the need for taking transitive reductions repeatedly. In this example, the initial value of ρ is α and the initial value of $(E_i \cup E_t)$ is

$$\begin{tikzcd}
\alpha & \\
\beta \arrow[u] & \mathit{real} \arrow[ul] \\
\mathit{int} \arrow[u] \arrow[ur] &
\end{tikzcd}$$

which is reduced. If the monotonicity-based instantiations begin by boosting β to α, then we will be stuck unless we take another transitive reduction, because α will be left with two predecessors, *int* and *real*.

3.6.5 Function *close*

At last, we can put all the pieces together and give the new function *close*. Function *close* performs the transformation to atomic inclusions, the collapsing of strongly connected components, the monotonicity-based instantiations, and finally the simple *close* of Figure 2.3. The complete function *close* is given in Figure 3.10.

Finally, we can combine the soundness and completeness lemmas of the previous three subsections to prove Lemmas 3.22 and 3.24.

$close(A, B, \tau)$:

let A_{ci} be the constant inclusions in A,
B_i be the inclusions in B,
B_t be the typings in B;
let $U = \textit{shape-unifier}(\{(\phi, \phi') \mid (\phi \subseteq \phi') \epsilon B_i\})$;
let $C_i = \textit{atomic-inclusions}(B_i U) \cup A_{ci}$,
$C_t = B_t U$;
let $V = \textit{component-collapser}(C_i)$;
let $S = UV$,
$D_i = \textit{transitive-reduction}(\textit{nontrivial-inclusions}(C_i V))$,
$D_t = C_t V$;
$E_i := D_i$; $E_t := D_t$; $\rho := \tau S$;
$\bar{\alpha} :=$ variables free in D_i or D_t or τS but not AS;
while there exist α in $\bar{\alpha}$ and π different from α such that
$AS \cup (E_i \cup E_t) \vdash (E_i \cup E_t)[\alpha := \pi] \cup \{\rho[\alpha := \pi] \subseteq \rho\}$
do $E_i := \textit{transitive-reduction}(\textit{nontrivial-inclusions}(E_i[\alpha := \pi]))$;
$E_t := E_t[\alpha := \pi]$;
$\rho := \rho[\alpha := \pi]$;
$\bar{\alpha} := \bar{\alpha} - \alpha$
od
let $E = (E_i \cup E_t) - \{\mathcal{C} \mid AS \vdash \mathcal{C}\}$;
let E'' be the set of constraints in E in which some α in $\bar{\alpha}$ occurs;
if AS has no free type variables,
then if $\textit{satisfiable}(E, AS)$ **then** $E' := \{\}$ else *fail*
else $E' := E$;
return $(S, E', \forall\bar{\alpha}$ **with** $E''.\rho)$.

Figure 3.10: Function *close*

3.7 Satisfiability Checking

In this section, we give some thoughts on the problem of checking the satisfiability of constraint sets, which may now contain inclusions as well as typings. In addition, we consider the related problem of checking whether an overloaded identifier has a particular type.

In Section 2.5, however, we saw that these problems are undecidable, even in the absence of subtyping. We therefore restrict ourselves again to the case where the assumption set A is an overloading by constructors, as defined in Section 2.6.

An important consequence of the restriction to overloading by constructors is that, roughly speaking, ($\forall$-elim) and ($\subseteq$) steps can be exchanged within derivations. For example, the derivation

$$\begin{array}{ll} A_0 \vdash \leq : \mathit{real} \rightarrow \mathit{real} \rightarrow \mathit{bool} & \text{(hypoth)} \\ A_0 \vdash \leq : \mathit{seq}(\mathit{real}) \rightarrow \mathit{seq}(\mathit{real}) \rightarrow \mathit{bool} & (\forall\text{-elim}) \\ A_0 \vdash \leq : \mathit{seq}(\mathit{int}) \rightarrow \mathit{seq}(\mathit{int}) \rightarrow \mathit{bool} & (\subseteq) \end{array}$$

can be replaced by the derivation

$$\begin{array}{ll} A_0 \vdash \leq : \mathit{real} \rightarrow \mathit{real} \rightarrow \mathit{bool} & \text{(hypoth)} \\ A_0 \vdash \leq : \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{bool} & (\subseteq) \\ A_0 \vdash \leq : \mathit{seq}(\mathit{int}) \rightarrow \mathit{seq}(\mathit{int}) \rightarrow \mathit{bool} & (\forall\text{-elim}). \end{array}$$

In general, the derivation of a typing $x : \tau$, where x is overloaded and τ is an instance of the *lcg* of x may be written so that all uses of ($\subseteq$) precede all uses of ($\forall$-elim).

As a result of this property, we can simplify typing assumptions in a constraint set by "unwinding" them. For example, the constraint $\leq : \mathit{seq}(\mathit{seq}(\beta)) \rightarrow \mathit{seq}(\mathit{seq}(\beta)) \rightarrow \mathit{bool}$ can be replaced by the constraint $\leq : \beta \rightarrow \beta \rightarrow \mathit{bool}$. In general, we can work backwards from the original constraint, eventually either failing or arriving at a set of *atomic instantiations* of the *lcg*. We can incorporate this simplification into *close* by changing the initialization of E_t from "$E_t := D_t$" to "$E_t := \mathit{unwind}(D_t)$".

Now we can give a way of checking that

$$AS \cup (E_i \cup E_t) \vdash E_t[\alpha := \pi],$$

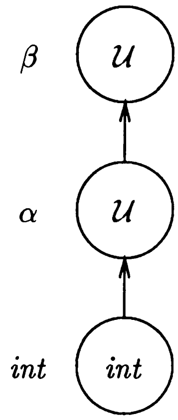
which is required on page 99. For each $x : \tau$ in $E_t[\alpha := \pi]$, it suffices to check that there is some typing $x : \phi$ in $AS \cup (E_i \cup E_t)$ such that

$$AS \cup (E_i \cup E_t) \vdash \phi \subseteq \tau.$$

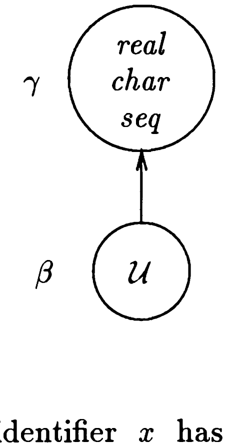
Next we consider the satisfiability problem. We are given a set E of atomic inclusions and overloading constraints; for example, we might have

$$E = \{\alpha \subseteq \beta,\ \mathit{int} \subseteq \alpha,\ \leq : \beta \rightarrow \beta \rightarrow \mathit{bool}\}.$$

We model E as a graph with a node for each type variable or constant in E and with edges corresponding to the inclusions in E. In addition, we associate a set of possible type constructors with each node, indicating that the node may only be instantiated to a type built from those type constructors. We use $\mathcal{U}$ to denote the set of all type constructors. For example, the first two constraints in the above E are modeled by



Next we must model typing constraints. We are again aided by the exchangeability of ($\forall$-intro) and ($\subseteq$) steps in derivations, which allows us to assume that all uses of ($\forall$-elim) precede all uses of ($\subseteq$). This allows us to characterize the set of all types of $\leq$. Indeed, $\leq$ has all supertypes of types of the form $\gamma \rightarrow \gamma \rightarrow bool$, where γ is a type built from the type constructor set for $\leq$, $\{real, char, seq\}$. Since $\gamma \rightarrow \gamma \rightarrow bool$ is antimonotonic in γ, this means that $\leq : \beta \rightarrow \beta \rightarrow bool$ if and only if β is a subtype of some type γ built from $\{real, char, seq\}$. Hence we can model the constraint $\leq : \beta \rightarrow \beta \rightarrow bool$ as



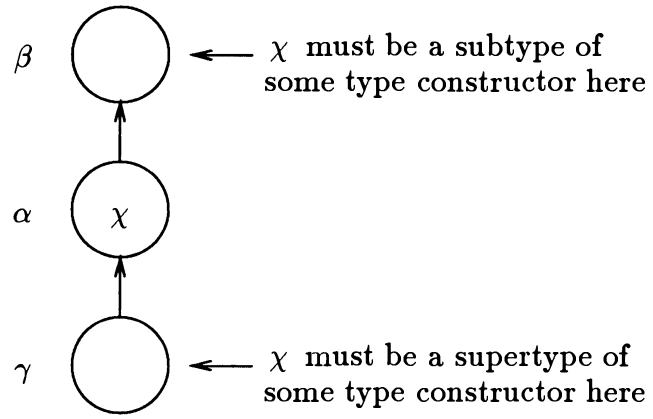
where γ is a new type variable.

In general, if an overloaded identifier x has *lcg* $\forall\alpha.\tau_x$, we must consider the monotonicity of τ_x in α in order to determine how to model a constraint $x : \tau_x[\alpha := \beta]$. If τ_x is monotonic in α, then β must be a supertype of some type γ built from the type constructor set of x. If τ_x is neither monotonic nor antimonotonic in α, then β must itself be built from the type constructor set of x.

Given the graph representation of E, we now try to characterize the possible satisfying instantiations of E. Observe that if any node has an associated type constructor set that is empty, then E is unsatisfiable. Hence, we can make progress by making deductions that allow us to shrink the type constructor sets.

First, we can restrict the type constructor sets to type constants. This is a consequence of the rules of subtyping. For example, if the constraint $\alpha \subseteq \beta$ is satisfied by $[\alpha, \beta := seq(\tau), seq(\tau')]$, then it is also satisfied by $[\alpha, \beta := \tau, \tau']$.[5]

Next, we can work "locally", eliminating type constants that can be seen not to work. We have the following picture:



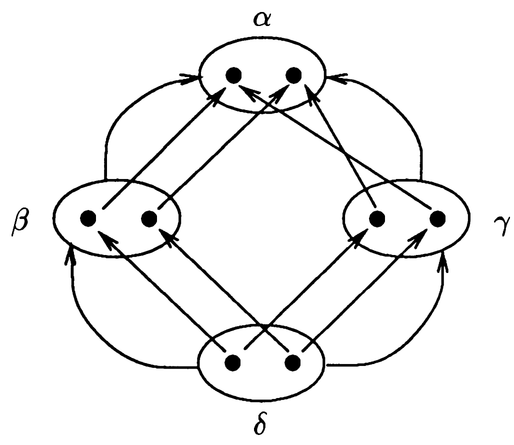
If χ is not a subtype of any of the possible type constructors of β or if χ is not a supertype of any of the possible type constructors of γ, then α certainly cannot be instantiated to χ. We can repeatedly eliminate useless type constructors, stopping when no further constructors can be eliminated, or when some type constructor set becomes empty.

Fuh and Mishra [FM88], in their investigation of a type system with subtyping but no overloading, propose the above local algorithm. They conjecture that, for any unsatisfiable constraint set, the local algorithm eventually produces an empty type constructor set. Unfortunately, this is not so, as may be seen by the following counterexample. Suppose that we are given the constraint set

$$\{\beta \subseteq \alpha,\ \gamma \subseteq \alpha,\ \delta \subseteq \beta,\ \delta \subseteq \gamma\}$$

and that each type variable has a choice of two type constructors whose inclusions are as indicated in the following picture.

[5] If there are any purely antimonotonic type constructors, a bit more work needs to be done.



Then no type constructors may be eliminated locally, yet the constraint set is unsatisfiable.

In fact, Wand and O'Keefe [WO89] show that the satisfiability problem with subtyping but no overloading is NP-complete. They also show that if the subtype relation has the property that any set of types with an upper bound has a *least* upper bound, then the satisfiability problem can be solved in polynomial time.

In the case of a type system with both overloading and subtyping, more research needs to be done. At worst, we can first apply the local algorithm to shrink the set of possible type constructors; if an empty type constructor set is produced, then the constraint set is unsatisfiable. If not, then we can do a brute-force search for a satisfying instantiation.

An alternative approach is to restrict the subtype relation somehow; perhaps one can show that if the subtype relation has some reasonable property, then the local algorithm works.

3.8 Conclusion

The introduction of constrained quantification has allowed subtyping to be added smoothly to the Hindley/Milner/Damas type system with overloading. The subtype relation needs to be restricted somewhat; roughly speaking, we allow only those type inclusions implied by inclusions among type constants. Also, restrictions on overloading are required for the satisfiability and typability problems to be decidable.

Chapter 4

Conclusion

4.1 A Practical Look at Type Inference

In the last two chapters, we have given a formal account of polymorphic type inference in the presence of overloading and subtyping. The main result is that the inference algorithms W_o and W_{os} are sound and complete and can therefore be used to infer principal types. Formal results, however, do not give much feel for how type inference works in practice; it would be nice to know that type inference tends to produce types that are understandable.

An implementation of algorithm W_{os} and the *close* of Figure 3.10 has been written in Scheme. The implementation is not especially efficient, but it is adequate for our purposes.

Define function *type* as the type of expression e under the initial assumptions A, as determined using our algorithms W_{os} and *close*:

$$\begin{array}{l} \mathit{type}(A, e): \\ \\ \text{let } (S, B, \tau) = W_{os}(A, e); \\ \text{let } (S', B', \sigma) = \mathit{close}(A, B, \tau); \\ \text{return } \sigma . \end{array}$$

As shown in Corollary 3.29, $\mathit{type}(A, e)$ computes a principal type for e with respect to A provided that A has satisfiable constraints, acceptable inclusions, and no free type variables. Let A_0 be the assumption set given in Figure 3.3. We now show the result of $\mathit{type}(A_0, e)$ on various expressions e.

- *fast-expon*

  Function *fast-expon* is given in Figure 4.1; $\mathit{type}(A_0, \mathit{fast\text{-}expon})$ is

  $$\forall \sigma \textbf{ with } 1 : \sigma,\ * : \sigma \rightarrow \sigma \rightarrow \sigma . \sigma \rightarrow \mathit{int} \rightarrow \sigma .$$

  This is the same as the type arrived at without subtyping.

```
fix λ exp.
    λx.λn. if (= n 0)
              1
              if ( even? n)
                 ( exp (* x x) (÷ n 2))
                 (* x ( exp x (− n 1)))
```

Figure 4.1: Example *fast-expon*

Function *fast-expon* can be used to raise 2-by-2 matrices to powers. It can be used, for example, to calculate the n th convergent to a periodic continued fraction in logarithmic time.

- *max*

  Function *max* is $\lambda x.\lambda y.\, \mathit{if}\ (\leq\ y\ x)\ x\ y$. The result of $\mathit{type}(A_0, \mathit{max})$ is

  $$\begin{aligned}&\forall \alpha, \beta, \gamma, \delta \ \textbf{with}\ \alpha \subseteq \gamma,\ \alpha \subseteq \delta,\ \beta \subseteq \gamma,\ \beta \subseteq \delta,\ \leq : \delta \rightarrow \delta \rightarrow \mathit{bool}\,.\\ &\qquad \alpha \rightarrow \beta \rightarrow \gamma\,.\end{aligned}$$

  This surprisingly complicated type cannot, it turns out, be further simplified. A simpler type can be found under stronger assumptions about the subtype relation: if we assume that any two types with an upper bound have a *least* upper bound, then the type of *max* can be simplified to

  $$\forall \gamma \ \textbf{with}\ \leq : \gamma \rightarrow \gamma \rightarrow \mathit{bool}\,.\, \gamma \rightarrow \gamma \rightarrow \gamma\,.$$

- *mergesort*

  Function *mergesort* is given in Figure 4.2; $\mathit{type}(A_0, \mathit{mergesort})$ is

  $$\forall \sigma_4 \ \textbf{with}\ \leq : \sigma_4 \rightarrow \sigma_4 \rightarrow \mathit{bool}\,.\, \mathit{seq}(\sigma_4) \rightarrow \mathit{seq}(\sigma_4)\,.$$

  The type computed for *split* is

  $$\forall \delta_2 . \mathit{seq}(\delta_2) \rightarrow \mathit{seq}(\mathit{seq}(\delta_2))$$

  and the type computed for *merge* is

  $$\begin{aligned}&\forall \iota_1, \theta_1, \eta_1, \kappa \ \textbf{with}\ \theta_1 \subseteq \kappa,\ \theta_1 \subseteq \eta_1,\ \iota_1 \subseteq \kappa,\ \iota_1 \subseteq \eta_1,\ \leq : \kappa \rightarrow \kappa \rightarrow \mathit{bool}\,.\\ &\qquad \mathit{seq}(\iota_1) \rightarrow \mathit{seq}(\theta_1) \rightarrow \mathit{seq}(\eta_1),\end{aligned}$$

  which is very much like the type of *max* above.

```
let split=
    fix λ split.
        λ lst. if ( null? lst)
                   ( cons nil ( cons nil nil))
                   if ( null? ( cdr lst))
                      ( cons lst ( cons nil nil))
                      let pair= split ( cdr ( cdr lst)) in
                          ( cons ( cons ( car lst) ( car pair))
                                 ( cons ( cons ( car ( cdr lst)) ( car ( cdr pair)))
                                        nil)) in
    let merge=
        fix λ merge.
            λ lst1. λ lst2.
                if ( null? lst1)
                   lst2
                   if ( null? lst2)
                      lst1
                      if (≤ ( car lst1) ( car lst2))
                         ( cons ( car lst1) ( merge ( cdr lst1) lst2))
                         ( cons ( car lst2) ( merge lst1 ( cdr lst2))) in
        fix λ mergesort.
            λ lst. if ( null? lst)
                      nil
                      if ( null? ( cdr lst))
                         lst
                         let lst1lst2= split lst in
                             merge ( mergesort ( car lst1lst2))
                                   ( mergesort ( car ( cdr lst1lst2)))
```

Figure 4.2: Example *mergesort*

```
fix λ reduce.
      λf.λa.λl. if ( null? l )
                     a
                     ( f ( car l)  ( reduce f a  ( cdr l)))
```

Figure 4.3: Example *reduce-right*

```
fix λ reduce.
      λf.λa.λl. if ( null? l )
                     a
                     if ( null?  ( cdr l))
                        ( car l)
                        ( f ( car l)  ( reduce f a  ( cdr l)))
```

Figure 4.4: Example *another-reduce-right*

Interestingly, when I first wrote *mergesort* I neglected to test for the case when *lst* has length 1; that is, I wrote

```
fix λ mergesort.
      λ lst. if ( null? lst)
                nil
                let lst1lst2 = split lst in
                     merge ( mergesort  ( car lst1lst2))
                                ( mergesort  ( car  ( cdr lst1lst2)))
```

The type constructed for this version of *mergesort* was

$$\forall \tau_3, \sigma_3 \textbf{ with } \leq : \tau_3 \rightarrow \tau_3 \rightarrow bool\,.\, seq(\sigma_3) \rightarrow seq(\tau_3)\,,$$

which showed me that something is wrong!

- *reduce-right*

  Function *reduce-right* is given in Figure 4.3; $type(A_0, reduce\text{-}right)$ is

  $$\forall \beta_1, \zeta. (\beta_1 \rightarrow \zeta \rightarrow \zeta) \rightarrow \zeta \rightarrow seq(\beta_1) \rightarrow \zeta.$$

- *another-reduce-right*

  Function *another-reduce-right* is given in Figure 4.4; it has type

  $$\forall \beta_1, \zeta \textbf{ with } \beta_1 \subseteq \zeta\,.\,(\beta_1 \rightarrow \zeta \rightarrow \zeta) \rightarrow \zeta \rightarrow seq(\beta_1) \rightarrow \zeta\,.$$

4.2 Related Work

Overloading had long been a neglected topic, but recently it has been addressed in papers by Kaes [Kae88] and, as part of the Haskell project, by Wadler and Blott [WB89]. Kaes' work restricts overloading quite severely; for example he does not permit constants to be overloaded. Both Kaes' and Wadler/Blott's systems ignore the question of whether a constraint set is satisfiable, with the consequence that certain nonsensical expressions are regarded as well typed. For example, if $+$ is overloaded with types $int \to int \to int$ and $real \to real \to real$, then in Wadler/Blott's system the expression $true + false$ is well typed, even though $+$ does not work on booleans. Kaes' system has similar difficulties.

There is a much larger body of literature on subtyping, spurred recently by excitement about object-oriented programming. None of the work of which I am aware considers the problem of type inference in the presence of both overloading and subtyping, but a number of papers do address the problem of type inference with subtyping. Most notable are the papers of Stansifer [Sta88], Curtis [Cur90], and Fuh and Mishra [FM89,FM90].

Stansifer gives a type inference algorithm and proves its completeness, but unfortunately his language does not include a **let** expression, which we saw to be a major source of difficulty. He also does not address type simplification or the problem of determining the satisfiability of constraint sets.

Curtis studies a very rich type system that is not restricted to *shallow* types, i.e. to types where all quantification is outermost. Largely because of the richness of his system, he is unable to characterize formally much of what he does. He gives an algorithm for the satisfiability problem but leaves its correctness as a conjecture. He gives an inference algorithm but does not address completeness at all. He performs elaborate type simplification, including some monotonicity-based instantiation, but does not completely prove the correctness of his algorithm.

Fuh and Mishra give a type inference algorithm and prove its completeness, but, like Stansifer's, their language does not include **let**. They also perform some type simplification.

4.3 Future Directions

We conclude by mentioning a few areas for future research.

4.3.1 Satisfiability of Constraint Sets

The problem of checking the satisfiability of a constraint set needs more study, especially in the case where both overloading and subtyping are present. In particular, it would be valuable to identify a restriction on the subtype relation that, combined with the restriction to overloading by constructors, allows the satisfiability problem to be solved efficiently. Also, the possibility of relaxing somewhat the restriction to overloading by constructors should be explored.

4.3.2 Records

A *record* is a labeled collection of fields. The subtype relation is more complicated on record types than on the types we have considered so far; for example, adding extra fields to a record type produces a subtype. What changes are needed to incorporate record types into our type system?

4.3.3 Semantic Issues

Our type system has the property that a typing $A \vdash e : \tau$ can have more than one derivation. For example, using the assumption set A_0 of Figure 3.3, the typing $A_0 \vdash 1 + 1 : real$ can be derived in two ways:

$$\begin{array}{ll} A_0 \vdash 1 : int & \text{(hypoth)} \\ A_0 \vdash + \ : int \rightarrow int \rightarrow int & \text{(hypoth)} \\ A_0 \vdash 1 + 1 : int & (\rightarrow\text{-elim}) \\ A_0 \vdash int \subseteq real & \text{(hypoth)} \\ A_0 \vdash 1 + 1 : real & (\subseteq) \end{array}$$

as well as

$$\begin{array}{ll} A_0 \vdash 1 : int & \text{(hypoth)} \\ A_0 \vdash int \subseteq real & \text{(hypoth)} \\ A_0 \vdash 1 : real & (\subseteq) \\ A_0 \vdash + \ : real \rightarrow real \rightarrow real & \text{(hypoth)} \\ A_0 \vdash 1 + 1 : real & (\rightarrow\text{-elim}). \end{array}$$

For the language to be unambiguous, it is crucial that these two derivations lead to the same meaning.

In general, the semantics of an expression e of type τ should be independent of the choice of derivation of $e : \tau$. This property has been explored by Reynolds [Rey80], and, more recently, by Breazu-Tannen *et al.* [BTCGS89], who call the property *coherence.*

4.3.4 Imperative Languages

Finally, we would like to extend our results to an imperative language. Unfortunately, extending polymorphic type inference to imperative languages has proved difficult, because the obvious typing rules turn out to be unsound in the presence of assignment. Promising approaches to this problem have been proposed by Tofte [Tof88] and by Leroy and Weis [LW91]; it should be possible to adapt their approaches to our type system.

Appendix A

Omitted Proofs

In this appendix we give the inductive proofs omitted from Sections 2.2 and 3.2.

Lemma 2.6 *If* x *does not occur in* A, $x \neq y$, *and* $A \cup \{x : \sigma\} \vdash' y : \tau$, *then* $A \vdash' y : \tau$.

Proof: By induction on the length of the derivation of $A \cup \{x : \sigma\} \vdash' y : \tau$. The only possible (non-trivial) last steps are (hypoth) and ($\forall$-elim$'$). (($\forall$-intro) and ($\equiv_\alpha$) could be used trivially.)

(hypoth) The derivation is simply $A \cup \{x : \sigma\} \vdash' y : \tau$ where $y : \tau \in A \cup \{x : \sigma\}$. Since $x \neq y$, $y : \tau \in A$, and so $A \vdash' y : \tau$.

($\forall$-elim$'$) The derivation ends with

$$\frac{\begin{array}{l} y : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau' \in A \cup \{x : \sigma\} \\ A \cup \{x : \sigma\} \vdash' C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \cup \{x : \sigma\} \vdash' y : \tau'[\bar{\alpha} := \bar{\pi}]}$$

Since $x \neq y$, $y : \forall \bar{\alpha} \textbf{ with } C \,.\, \tau' \in A$. Since x does not occur in A, no constraint in $C[\bar{\alpha} := \bar{\pi}]$ is of the form $x : \rho$. So the induction hypothesis may be applied to each member of $C[\bar{\alpha} := \bar{\pi}]$, giving $A \vdash' C[\bar{\alpha} := \bar{\pi}]$. So by ($\forall$-elim$'$),

$$A \vdash' y : \tau'[\bar{\alpha} := \bar{\pi}].$$

QED

Notice that the derivation of $AS \vdash' e : \sigma S$ given by Corollary 2.5 can be longer than the derivation of $A \vdash' e : \sigma$, because of the necessity of inserting an ($\equiv_\alpha$) step after each ($\forall$-intro) step. In the proof of Lemma 2.7 below, however, we will need the property that applying a substitution to a derivation does not lengthen the derivation. This can be achieved by defining the notion of a *padded* derivation: a derivation is padded if every ($\forall$-intro) step is followed by a (possibly identity) ($\equiv_\alpha$) step. Any derivation can be padded, of course, and applying a substitution to a padded derivation does not lengthen it, since the extra ($\equiv_\alpha$) steps introduced by the substitution can be combined with the padding ($\equiv_\alpha$) steps that have been provided.

Lemma 2.7 *If* $A \vdash' e : \sigma$ *then* $A \cup B \vdash' e : \sigma$, *provided that no identifier occurring in* B *is* λ*-bound or let-bound in* e.

Proof: By induction on the length of the derivation of $A \vdash' e : \sigma$, which we assume without loss of generality to be padded. Consider the last step of the derivation:

(hypoth) The derivation is simply $A \vdash' x : \sigma$ where $x : \sigma \in A$. So $x : \sigma \in A \cup B$, and $A \cup B \vdash' x : \sigma$.

($\rightarrow$-intro) The derivation ends with

$$\frac{A \cup \{x : \tau\} \vdash' e' : \tau'}{A \vdash' \lambda x.e' : \tau \rightarrow \tau'}$$

where x does not occur in A. No identifier occurring in B is λ-bound or let-bound in e', so by induction,

$$(A \cup \{x : \tau\}) \cup B \vdash' e' : \tau'.$$

Since x *is* λ-bound in $\lambda x.e'$, x does not occur in B, so x does not occur in $A \cup B$. So, since $(A \cup \{x : \tau\}) \cup B = (A \cup B) \cup \{x : \tau\}$, ($\rightarrow$-intro) may be used to give

$$A \cup B \vdash' \lambda x.e' : \tau \rightarrow \tau'.$$

($\rightarrow$-elim) The derivation ends with

$$\frac{\begin{array}{l} A \vdash' e' : \tau \rightarrow \tau' \\ A \vdash' e'' : \tau \end{array}}{A \vdash' e' e'' : \tau'}$$

No identifier occurring in B is λ-bound or let-bound in e' or e'', so by induction, $A \cup B \vdash' e' : \tau \rightarrow \tau'$ and $A \cup B \vdash' e'' : \tau$, and so by ($\rightarrow$-elim),

$$A \cup B \vdash' e' e'' : \tau'.$$

(let) The derivation ends with

$$\frac{\begin{array}{l} A \vdash' e' : \sigma' \\ A \cup \{x : \sigma'\} \vdash' e'' : \tau \end{array}}{A \vdash' \textbf{let } x = e' \textbf{ in } e'' : \tau}$$

where x does not occur in A. No identifier occurring in B is λ-bound or let-bound in e' or e'', so by induction, $A \cup B \vdash' e' : \sigma'$ and $(A \cup \{x : \sigma'\}) \cup B \vdash' e'' : \tau$. Since x *is* let-bound in **let** $x = e'$ **in** e'', x does not occur in B, so x does not occur in $A \cup B$. So (let) may be used to give

$$A \cup B \vdash' \textbf{let } x = e' \textbf{ in } e'' : \tau.$$

($\forall$-intro) A padded derivation cannot end with ($\forall$-intro).

($\forall$-elim$'$) The derivation ends with

$$\frac{\begin{array}{l} x : \forall\bar{\alpha} \textbf{ with } C\,.\,\tau \in A \\ A \vdash' C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash' x : \tau[\bar{\alpha} := \bar{\pi}]}$$

Now $x : \forall\bar{\alpha}$ **with** $C\,.\,\tau \in A \cup B$, and by induction on each member of $C[\bar{\alpha} := \bar{\pi}]$, (using the fact that for any $y : \rho$ in $C[\bar{\alpha} := \bar{\pi}]$, no identifier in B is λ-bound or let-bound in y,) $A \cup B \vdash' C[\bar{\alpha} := \bar{\pi}]$. Hence by ($\forall$-elim$'$),

$$A \cup B \vdash' x : \tau[\bar{\alpha} := \bar{\pi}].$$

($\equiv_\alpha$) The derivation ends with

$$\frac{\begin{array}{l} A \vdash' e : \sigma' \\ \sigma' \equiv_\alpha \sigma \end{array}}{A \vdash' e : \sigma}$$

If the derivation of $A \vdash' e : \sigma'$ is padded, then by induction $A \cup B \vdash' e : \sigma'$, so by ($\equiv_\alpha$),

$$A \cup B \vdash' e : \sigma.$$

If, on the other hand, the derivation of $A \vdash' e : \sigma'$ is not padded, then it ends with ($\forall$-intro):

$$\frac{\begin{array}{l} A \cup C \vdash' e : \tau \\ A \vdash' C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash' e : \forall\bar{\alpha} \textbf{ with } C\,.\,\tau}$$

where $\bar{\alpha}$ are not free in A, and where $\sigma' = \forall\bar{\alpha}$ **with** $C\,.\,\tau$. Let $\bar{\beta}$ be distinct type variables not occurring free in σ', A, or B. By Corollary 2.5,

$$(A \cup C)[\bar{\alpha} := \bar{\beta}] \vdash' e : \tau[\bar{\alpha} := \bar{\beta}],$$

and (by our padding assumption) the derivation is no longer than the derivation of $A \cup C \vdash' e : \tau$. Since no identifier occurring in B is λ-bound or let-bound in e, by induction we have $(A \cup C)[\bar{\alpha} := \bar{\beta}] \cup B \vdash' e : \tau[\bar{\alpha} := \bar{\beta}]$. Since $\bar{\alpha}$ are not free in A, $(A \cup C)[\bar{\alpha} := \bar{\beta}] = A \cup C[\bar{\alpha} := \bar{\beta}]$, so we have

$$A \cup B \cup C[\bar{\alpha} := \bar{\beta}] \vdash' e : \tau[\bar{\alpha} := \bar{\beta}].$$

Also by induction on each member of $C[\bar{\alpha} := \bar{\pi}]$, we have $A \cup B \vdash' C[\bar{\alpha} := \bar{\pi}]$. Since $C[\bar{\alpha} := \bar{\pi}] = C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}]$, we have

$$A \cup B \vdash' C[\bar{\alpha} := \bar{\beta}][\bar{\beta} := \bar{\pi}].$$

So by ($\forall$-intro),

$$A \cup B \vdash' e : \forall \bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}] \,.$$

Since

$$\forall \bar{\beta} \textbf{ with } C[\bar{\alpha} := \bar{\beta}] \,.\, \tau[\bar{\alpha} := \bar{\beta}] \equiv_\alpha \sigma' \equiv_\alpha \sigma,$$

it follows by ($\equiv_\alpha$) that

$$A \cup B \vdash' e : \sigma.$$

QED

Lemma 2.8 *Let B be a set of τ-assumptions. If $A \cup B \vdash' e : \sigma$ and $A \vdash' B$, then $A \vdash' e : \sigma$.*

Proof: By induction on the length of the derivation of $A \cup B \vdash' e : \sigma$. Consider the last step:

(hypoth) The derivation is simply $A \cup B \vdash' x : \sigma$ where $x : \sigma \in A \cup B$. If $x : \sigma \in A$ then $A \vdash' x : \sigma$. If $x : \sigma \in B$ then by the assumption $A \vdash' B$, again $A \vdash' x : \sigma$.

($\rightarrow$-intro) The derivation ends with

$$\frac{(A \cup B) \cup \{x : \tau\} \vdash' e' : \tau'}{A \cup B \vdash' \lambda x.e' : \tau \rightarrow \tau'}$$

where x does not occur in $A \cup B$. By Lemma 2.7 applied to each member of B, $A \cup \{x : \tau\} \vdash' B$. Now $(A \cup B) \cup \{x : \tau\} = (A \cup \{x : \tau\}) \cup B$, so by induction

$$A \cup \{x : \tau\} \vdash' e' : \tau'.$$

Since x does not occur in $A \cup B$, it does not occur in A, so by ($\rightarrow$-intro),

$$A \vdash' \lambda x.e' : \tau \rightarrow \tau'.$$

($\rightarrow$ -elim) The derivation ends with

$$\frac{\begin{array}{l} A \cup B \vdash' e' : \tau \rightarrow \tau' \\ A \cup B \vdash' e'' : \tau \end{array}}{A \cup B \vdash' e'\, e'' : \tau'}$$

By induction, $A \vdash' e' : \tau \rightarrow \tau'$ and $A \vdash' e'' : \tau$, so by ($\rightarrow$ -elim),

$$A \vdash' e'e'' : \tau'.$$

(let) The derivation ends with

$$\frac{\begin{array}{l} A \cup B \vdash' e' : \sigma' \\ (A \cup B) \cup \{x : \sigma'\} \vdash' e'' : \tau \end{array}}{A \cup B \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e''\ : \tau}$$

where x does not occur in $A \cup B$. By induction, $A \vdash' e' : \sigma'$. By Lemma 2.7 applied to each member of B, $A \cup \{x : \sigma'\} \vdash' B$. Since $(A \cup B) \cup \{x : \sigma'\} = (A \cup \{x : \sigma'\}) \cup B$, the induction hypothesis may be applied to give

$$A \cup \{x : \sigma'\} \vdash' e'' : \tau.$$

Since x does not occur in $A \cup B$, it does not occur in A, so by (let),

$$A \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e''\ : \tau.$$

($\forall$ -intro) The derivation ends with

$$\frac{\begin{array}{l} (A \cup B) \cup C \vdash' e : \tau \\ A \cup B \vdash' C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \cup B \vdash' e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau}$$

where $\bar{\alpha}$ are not free in $A \cup B$. By Lemma 2.7 on each member of B, $A \cup C \vdash' B$. Since $(A \cup B) \cup C = (A \cup C) \cup B$, the induction hypothesis may be applied to give

$$A \cup C \vdash' e : \tau.$$

Also by induction,

$$A \vdash' C[\bar{\alpha} := \bar{\pi}].$$

Since $\bar{\alpha}$ are not free in A, ($\forall$ -intro) gives

$$A \vdash' e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau\,.$$

(**$\forall$-elim$'$**) The derivation ends with

$$\frac{\begin{array}{l} x : \forall\bar{\alpha} \textbf{ with } C\,.\,\tau \in A \cup B \\ A \cup B \vdash' C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \cup B \vdash' x : \tau[\bar{\alpha} := \bar{\pi}]}$$

Since B contains only τ-assumptions,

$$x : \forall\bar{\alpha} \textbf{ with } C\,.\,\tau \in A,$$

and by induction, $A \vdash' C[\bar{\alpha} := \bar{\pi}]$. So by ($\forall$-elim$'$),

$$A \vdash' x : \tau[\bar{\alpha} := \bar{\pi}].$$

(**$\equiv_\alpha$**) The derivation ends with

$$\frac{\begin{array}{l} (A \cup B) \vdash' e : \sigma' \\ \sigma' \equiv_\alpha \sigma \end{array}}{(A \cup B) \vdash' e : \sigma}$$

By induction, $A \vdash' e : \sigma'$, so by ($\equiv_\alpha$),

$$A \vdash' e : \sigma.$$

QED

Lemma 2.13 *If $A \vdash e : \sigma$ then no identifier occurring in A is λ-bound or let-bound in e.*

Proof: By induction on the length of the derivation of $A \vdash e : \sigma$. Consider the last step of the derivation:

(hypoth) The derivation is simply $A \vdash x : \sigma$ where $x : \sigma \in A$. Nothing is λ-bound or let-bound in x.

(**$\rightarrow$-intro**) The derivation ends with

$$\frac{A \cup \{x : \tau\} \vdash e' : \tau'}{A \vdash \lambda x.e' : \tau \rightarrow \tau'}$$

where x does not occur in A. By induction, no identifier in $A \cup \{x : \tau\}$ is λ-bound or let-bound in e'. So no identifier in A is λ-bound or let-bound in e'. And since x does not occur in A, no identifier in A is λ-bound or let-bound in $\lambda x.e'$.

($\rightarrow$ -elim) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e' : \tau \rightarrow \tau' \\ A \vdash e'' : \tau \end{array}}{A \vdash e'\, e'' : \tau'}$$

By induction, no identifier in A is λ-bound or let-bound in e' or e''. Hence no identifier in A is λ-bound or let-bound in $e'\, e''$.

(let) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e' : \sigma' \\ A \cup \{x : \sigma'\} \vdash e'' : \tau \end{array}}{A \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e''\ : \tau}$$

where x does not occur in A. By induction, no identifier in A is λ-bound or let-bound in e' and no identifier in $A \cup \{x : \sigma'\}$ is λ-bound or let-bound in e''. Since x does not occur in A, no identifier in A is λ-bound or let-bound in let $x = e'$ in e'' .

($\forall$ -intro) The derivation ends with

$$\frac{\begin{array}{l} A \cup C \vdash e : \tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau}$$

where $\bar{\alpha}$ are not free in A. By induction, no identifier in $A \cup C$ is λ-bound or let-bound in e; hence no identifier in A is λ-bound or let-bound in e.

($\forall$ -elim) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \forall \bar{\alpha}\ \mathbf{with}\ C\,.\,\tau \\ A \vdash C[\bar{\alpha} := \bar{\pi}] \end{array}}{A \vdash e : \tau[\bar{\alpha} := \bar{\pi}]}$$

By induction, no identifier in A is λ-bound or let-bound in e.

($\equiv_\alpha$) The derivation ends with

$$\frac{\begin{array}{l} A \vdash e : \sigma' \\ \sigma' \equiv_\alpha \sigma \end{array}}{A \vdash e : \sigma}$$

By induction, no identifier in A is λ-bound or let-bound in e.

QED

Now we give the proofs from Section 3.2.

Lemma 3.7 *If* x *does not occur in* A, $x \neq y$, *and* $A \cup \{x : \sigma\} \vdash' y : \tau$, *then* $A \vdash' y : \tau$.

Proof: The proof of Lemma 2.6 suffices, except that we must consider one more possible last step of the derivation of $A \cup \{x : \sigma\} \vdash' y : \tau$.

$(\subseteq)$ The derivation ends with

$$\frac{\begin{array}{c} A \cup \{x : \sigma\} \vdash' y : \tau' \\ A \cup \{x : \sigma\} \vdash' \tau' \subseteq \tau \end{array}}{A \cup \{x : \sigma\} \vdash' y : \tau}$$

By induction, $A \vdash' y : \tau'$ and by Lemma 3.2, $A \vdash' \tau' \subseteq \tau$. So by $(\subseteq)$, $A \vdash' y : \tau$.

QED

Lemma 3.8 *If* $A \vdash' e : \sigma$ *then* $A \cup B \vdash' e : \sigma$, *provided that no identifier occurring in* B *is* λ*-bound or let-bound in* e.

Proof: The proof of Lemma 2.7 suffices, except that we must consider the case when the derivation of $A \vdash' e : \sigma$ ends with $(\subseteq)$.

$(\subseteq)$ The derivation ends with

$$\frac{\begin{array}{c} A \vdash' e : \tau \\ A \vdash' \tau \subseteq \tau' \end{array}}{A \vdash' e : \tau'}$$

By induction, $A \cup B \vdash' e : \tau$ and by Lemma 3.3, $A \cup B \vdash' \tau \subseteq \tau'$. So by $(\subseteq)$, $A \cup B \vdash' e : \tau'$.

QED

Lemma 3.9 *Let* B *be a set of* τ*-assumptions. If* $A \cup B \vdash' e : \sigma$ *and* $A \vdash' B$, *then* $A \vdash' e : \sigma$.

Proof: The proof of Lemma 2.8 suffices, except that we must consider the case when the derivation of $A \cup B \vdash' e : \sigma$ ends with $(\subseteq)$.

$(\subseteq)$ The derivation ends with

$$\frac{\begin{array}{c} A \cup B \vdash' e : \tau \\ A \cup B \vdash' \tau \subseteq \tau' \end{array}}{A \cup B \vdash' e : \tau'}$$

By induction, $A \vdash' e : \tau$ and by Lemma 3.4, $A \vdash' \tau \subseteq \tau'$. So by $(\subseteq)$, $A \vdash' e : \tau'$.

QED

Bibliography

[AGU72] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, June 1972.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion (preliminary report). In *IEEE Fourth Annual Symposium on Logic in Computer Science*, pages 112–129, 1989.

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., 1986.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, 1986.

[Cur90] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis.* Ph.D. dissertation, Cornell University, January 1990.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[DD85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer-Verlag, 1988.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[Gir72] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII, 1972.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[GMvdSU89] David Gries, Alain J. Martin, Jan L. A. van de Snepscheut, and Jan Tijmen Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12:151–155, 1989.

[GP85] David Gries and Jan Prins. A new notion of encapsulation. In *Proceedings ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 131–139, 1985.

[GV89] David Gries and Dennis Volpano. The transform—a new language construct. Technical Report TR 89-1045, Department of Computer Science, Cornell University, October 1989.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.

[Kni89] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[KT90] A. J. Kfoury and Jerzy Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Fifth IEEE Symposium on Logic in Computer Science*, pages 2–11, 1990.

[Lak76] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge University Press, 1976. Edited by John Worrall and Elie Zahar.

[Lei83] Daniel Leivant. Polymorphic type inference. In *10th ACM Symposium on Principles of Programming Languages*, pages 88–98, Austin, Texas, 1983.

[Llo84] John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin/New York, 1984.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *18th ACM Symposium on Principles of Programming Languages*, pages 291–302, 1991.

[McC84] Nancy McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1984.

[MH88] John C. Mitchell and Robert Harper. The essence of ML. In *15th ACM Symposium on Principles of Programming Languages*, pages 28–46, 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit84] John C. Mitchell. Coercion and type inference (summary). In *Eleventh ACM Symposium on Priciples of Programming Languages*, pages 175–185, 1984.

[Mor73] James H. Morris, Jr. Types are not sets. In *ACM Symposium on Principles of Programming Languages*, pages 120–124, Boston, Massachusetts, 1973.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[Poi13] Henri Poincaré. *The Foundations of Science.* The Science Press, Lancaster, Pennsylvania, 1913. Authorized English translation by G. B. Halsted.

[Pra65] Dag Prawitz. *Natural Deduction.* Almqvist and Wiksell, Stockholm, 1965.

[Pri87] Jan F. Prins. *Partial Implementations in Program Derivation.* Ph.D. dissertation, Cornell University, August 1987.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rey80] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, 1980.

[Rey81] John C. Reynolds. The essence of ALGOL. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland Publishing Company, 1981.

[Rey85] John C. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, 1985.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Department of Computer Science, Carnegie Mellon University, June 1988.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sta88] Ryan Stansifer. Type inference with subtypes. In *Fifteenth ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* Ph.D. dissertation, University of Edinburgh, 1988.

[vD83] Dirk van Dalen. *Logic and Structure.* Springer-Verlag, 1983.

[VS91] Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ML typability with overloading. In *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts, August 1991. To appear.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc.* In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[WO89] Mitchell Wand and Patrick O'Keefe. On the complexity of type inference with coercion. In *Conference on Functional Programming Languages and Computer Architecture*, 1989.