TYPE INFERENCE IN PROGRAMMING LANGUAGES

By

GAVIN ALEXANDER VOGT

A Thesis Submitted to The W.A. Franke Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 2 3

Approved by:

Dr. Saumya Debray
Department of Computer Science

Abstract

The purpose of this paper is to familiarize a beginner with the field of type inference. By the end, the reader should be comfortable with common concepts in type inference and understand how to implement it. The paper introduces types and the lambda calculus as preliminary concepts that are important to understand when reading papers in the field. It walks through a simple example to build the reader's intuition for type inference and motivate further discussion. Then, it delves into the unification process, which appears when solving the type equations that are generated according to typing rules. This paper focuses specifically on type inference for the Hindley-Milner type system, which is readily applied to functional programming languages. Hindley-Milner typing provides parametric polymorphism and guarantees that code is well-typed. The limitations and potential extensions to Hindley-Milner typing are examined by considering the ideas contributed by various papers in the field. Use of type inference in real-world programming languages is explored to show its practical applications and how it can benefit programmers. My implementation of unification and type inference algorithms discussed can be found at `https://github.com/gavinvogt/type-inference-survey`.

Contents

1 Introduction

Type inference is a technique for calculating the type of every expression and identifier in a program, without (or minimally) requiring type annotations from the programmer. A program that passes type inference is ensured to be well-typed, thus making it a form of type checking. All operations in a *well-typed* program are consistent with the language's type system and will exhibit well-defined behavior. However, it can be argued that statically typed languages requiring explicit type annotations provide the same type safety guarantee. What, then, provides a compelling argument for type inference?

One immediate benefit is fewer physical keystrokes—type annotations can become bulky and tedious to write for more complex data structures. Although the annotations can be useful when reading code, this advantage has diminished in the advent of intelligent code editors. A code editor linked with type inference can provide accurate hints and type information about variables. Type inference also allows for a less involved refactoring process when updating code. For a type-annotated language, annotations must be updated to stay correct as the code changes. In contrast, a type-inferred language avoids this additional tedium since the types of expressions are calculated based on the current code rather than retaining annotation artifacts.

In the case of dynamically typed languages such as Python and JavaScript, there is no built-in type checking to speak of. Type inference therefore becomes an invaluable tool to verify the type safety, or well-typedness, of programs, as it can be applied on top of the original code. For example, the type inference library Mypy has been used by Dropbox to type check millions of lines of Python in their codebase, without forcing a rewrite of all the code. TypeScript is a superset of JavaScript that was created with the primary purpose of type inference in mind.

Overall, type inference is important because it ensures type safety while staying out of the way of the programmer—most expressions can be type-inferred with little to no explicit type annotation.

1.1 Roadmap

I will discuss types in section 2 to clarify ideas used in the paper. I will continue by gently introducing the lambda calculus in section 3. A thorough understanding of the subject is not necessary, but it will help lay the foundation for understanding type inference. In addition, it is a potential notational stumbling block to a reader unfamiliar with the field. In section 4, I will motivate the type inference process by walking through an example. My goal is to show the relative simplicity of the technique and introduce unification as a crucial step in type inference algorithms. Then, section 5 will explain unification in detail and discuss some of the efforts to produce more efficient unification algorithms. Section 6 explores type inference and a few algorithms, with special attention given to the Hindley-Milner type system. It is followed by a discussion of extensions to Hindley-Milner typing, such as overloading and subtyping, in section 7. Finally, section 8 explores some of the practical applications of type inference to real-world languages. By the end of this paper, you will have a complete understanding of how to apply type inference to a purely functional programming language, as well as access to my own Python implementation of the technique.

2 Types

In a paper devoted to type inference, it is critical to nail down exactly what is meant by a *type*. This section explores the definition of a type, the flexibility offered by various forms of polymorphism, and how types are formally treated using typing rules.

2.1 Type definition

What is a type? Types are integral to the programming experience, but it can be difficult to describe precisely what constitutes a type. A type can be thought of as a set of possible values, or as an interpretation of the underlying bits. For example, an `int` could be represented by the set $\{-2147483648, -2147483647, \ldots, -1, 0, 1, \ldots, 2147483646, 2147483647\}$. Alternatively, the latter is a common way of thinking in C, where the programmer has access to the bit-level representation of data and can choose how to interpret it.

However, this definition of a type is limited in sophistication. Rather than asking what a type *is*, [Cardelli 85] asks why types are *needed* in programming languages. They narrow in on the intuition that an untyped universe of computational objects, such as bit strings in memory, naturally decomposes into subsets of objects with uniform behavior. Such sets of objects are distinguished by the set of operations that can be applied to them and become known as types. Therefore, we define a type by a set of values and the operations they support, such as addition for integers and logical operators for booleans.

Types are simply a construct built on top of some untyped internal representation, which can easily be violated by inconsistent operations to cause undesired results. As a result, type systems are instituted to ensure that the underlying untyped data is compatible with its intended use. Type inference takes advantage of the knowledge of types and their supported operations to infer an expression's type based on the operations performed on it. When the type of every expression in a program can be determined before running, it is *statically* typed and can prevent type-related inconsistencies. In contrast, a program is *dynamically* typed when types are assigned at runtime. Dynamic typing provides more flexibility, but it loses out on the benefits of assured type safety.

2.2 Polymorphism

Polymorphism is a key concept that grants type flexibility, so the same code does not need to be re-written to handle each input type. Following from the Greek roots, polymorphism means that a type can have "many shapes." A type variable represents an arbitrary type that can be instantiated with concrete types according to the context in which it is used. A monomorphic type, or *monotype*, is a type containing no occurrences of such type variables. A type containing at least one type variable is referred to as a polymorphic type, or *polytype* [Milner 78].

The most trivial example of polymorphism is the identity function $(\lambda x.\, x)$, which has type $(\forall \alpha.\, \alpha \to \alpha)$. This lambda expression possesses a polytype, since it contains a type variable represented by α. If the lambda were to be called with `int` or `bool` arguments, for example, it would take on a monotype of `int` $\to$ `int` or `bool` $\to$ `bool` in its respective context. Polymorphism also appears in languages such as Java and C++ through generics, as in `List<T>`. Additionally, polymorphism can take on different forms, each of which will be covered in this paper.

Parametric polymorphism is attained when a function is defined for a range of types, with the same behavior for each one. It is considered the purest form of polymorphism since it is truly universal, accepting an argument of any type, without any restrictions. It allows abstraction over the types of function arguments, making it easier to develop reusable software components [Kaes 92].

Ad hoc polymorphism, more frequently referred to as overloading, occurs when a function produces different behavior for each type [Wadler 89]. For example, the `+` operator is commonly overloaded to handle both integer and floating-point arguments, and will return an integer or floating-point value based on the arguments' types. Method overloading is a prominent feature in Java that allows the user to write a different implementation for each unique method signature (deriving from the order and types of parameters).

Subtype polymorphism is the idea of allowing a certain type and any of its subtypes. This particular form of polymorphism arises for *nominal* subtyping, where the subtype is contained in a hierarchy of types deriving from the root "parent" type. Subtype polymorphism is found in Java, C#, and other languages that use nominal subtyping.

Row polymorphism is similar to subtype polymorphism, but instead applies to the case of *structural* subtyping. It requires that types have compatible shapes, but they need not be related in a nominal type hierarchy. For example, in TypeScript:

```
type Person = {
    name: string,
    age: number,
};

function showPerson(p: Person) {
    console.log(p.name + ", age " + p.age);
}

// Valid structure - contains required `name` and `age` fields
showPerson({ name: "Alice", age: 35, eyeColor: "green" });

// Not valid - missing the `age` field
showPerson({ name: "Bob" });
```

2.3 Typing rules

As programmers, we have a natural intuition for what type an expression will have. However, in order to algorithmically determine the type of an expression, we must have some formal system of typing rules, also known as type judgments. Typing rules are written as natural deduction rules, which can be difficult to understand. Unfortunately, nearly every paper you read about type inference will list an assortment of these formal rules under the assumption that you are already familiar with their form. A natural deduction rule is actually quite simple; it consists of a premise and conclusion, written as

$$\frac{\text{Premise}}{\text{Conclusion}}\,[\text{Label}]\ .$$

For example, consider the following rule:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,[\text{hypoth}]\ .$$

Let us begin by examining the top line of the rule. The Γ (capital gamma) on the righthand side represents the context, or assumption set. It is also seen frequently as A, but I will use Γ throughout this paper for consistency. The typing expression $x : \sigma$ means that the variable x has some type σ. Therefore, the premise is that the context Γ contains a variable x with type σ.

In the bottom line, the vdash $\vdash$ can be read as "executed under." Due to the premise that x has type σ in Γ, we conclude that $\Gamma \vdash x$ (the value of variable x when executed under context Γ) has a type of σ. This typing rule may take effect if you are accessing a variable defined earlier in the code, or when using a global built-in function like `map`. For example, after assigning the value `4` to `y`, an assumption set could look like $\{y$: `int`$\}$.

2.4 Related work

[Cardelli 85] explains types and polymorphism in far greater detail than this paper, and it is very helpful for building intuition on the topic. [Milner 78] discusses polymorphism and how it can be included in programming languages.

3 Lambda calculus

The lambda calculus, also referred to as λ-calculus, is a formal system of symbolic logic used for representing functions and function applications. It was initially developed by Alonzo Church in his work regarding number theory, but it has proven greatly applicable to the field of computer science as well. For example, [Morris 69] performs a 130-page analysis of recursion and types, using lambda calculus as a model of programming languages. Lambda calculus is well suited to the task because it is very simple and closely resembles a programming language. The functional programming paradigm, which is based on the composition and application of functions, evolved from λ-calculus. A brief introduction to lambda calculus will be worthwhile to help us formalize ideas about type inference.

3.1 Definition

Expressions in λ-calculus are described concisely by the following Backus-Naur form (BNF):

$$M, N ::= x \mid (MN) \mid (\lambda x.\, M)\,.$$

In the above grammar, M and N are lambda terms, and x represents a *variable* contained by an infinite set of variables $\mathcal{V}$. A term of the form (MN) is known as an *application*, and the parenthesis are omitted by convention. Additionally, function applications are left-associative, so fxy would refer to $(f(x))(y)$, which is known as “currying.” The third form, $(\lambda x.\, M)$, is known as a *lambda abstraction*. This is comparable to the lambda expressions found in many programming languages, taking argument x and returning the result of expression M. For example, `fn x => x + 1` in ML could be represented as $\lambda x.\,(x+1)$ in a more permissive[1] λ-calculus.

There exist variations to λ-calculus, including simply typed and polymorphic. In this paper, a limited understanding of λ-calculus will suffice.

3.2 Free and bound variables

A lambda term may contain a combination of *free* and *bound* variables. As illustration, consider the lambda abstraction $\lambda x.\, xy$. The x in this term is *bound* by the binder λx, whereas the variable y is not bound and is therefore *free*.

3.3 Reductions

The λ-calculus allows several reduction operations that result in an equivalent, *reduced* lambda term.

[1]The basic λ-calculus syntax I describe clearly does not contain an addition (+) operator; however, an interested reader may explore the λ-calculus for numerals and addition to see that an abstraction for the operation could be defined and subsequently abstracted away to form a more “permissive” λ-calculus.

3.3.1 α-conversion

An α-conversion refers to renaming a variable in a term. Given term M and variables x and y, $M\{y/x\}$ represents the result of renaming x to y in M. Note that it follows the order $M\{\text{new}/\text{old}\}$. For example, for $M = \lambda x.\, (\lambda y.\, y)x$, the α-conversion $M\{z/x\}$ results in $\lambda z.\, (\lambda y.\, y)z$. If two lambda terms differ only in the names of bound variables, they are said to be α-equivalent since they can be made identical by a series of α-conversions.

The α-conversion operation is analogous to renaming a function parameter in a program and replacing all its instances with the new name. The name of the variable should not affect the behavior of the code, just as it will not affect the result of the lambda term.

3.3.2 β-reduction

Substitution, similar to the renaming operation, is the process of replacing a variable with a lambda term. $M[N/x]$ represents the result of substituting N for x in term M. Only free variables in M should be replaced, since a substitution should have the same result regardless of a bound variable's name (refer to α-equivalence). Additionally, we must take care not to accidentally "capture" a variable that was free in N but is bound in M. For example, for $M = \lambda x.\, yx$ and $N = \lambda z.\, xz$, the substitution $M[N/y]$ will consider the free x in N to be bound by the λx in M. To avoid this undesirable effect, we can rename the bound variable in M such that $M = \lambda x'.\, yx'$ to prevent a name collision. This results in the following:

$$M[N/y] = (\lambda x'.\, yx')[N/y] = \lambda x'.\, Nx' = \lambda x'.\, (\lambda z.\, xz)x' \,.$$

A β-reduction is comparable to passing arguments to a function. In λ-calculus, this is seen as the result of an abstraction (the function) applied to another term (the argument). Thus, in the β-reduction of a term $(\lambda x.\, M)N$ we replace the bound variable of the abstraction with the argument to produce $M[N/x]$.

3.3.3 η-reduction

An η-reduction simplifies a term by dropping an unnecessary abstraction. Consider, for example, the abstraction $\lambda x.\, fx$. Clearly, this term is redundant and can be simplified to f. The term $\lambda x.\, fx$ is η-equivalent to f, and therefore the terms are *extensionally*[2] equal.

3.4 System F

System F is the polymorphic lambda calculus. It is a typed λ-calculus that extends simply typed λ-calculus with universally quantified ($\forall$) types. The system formalizes the parametric polymorphism used in programming languages such as Haskell and ML. For example, the type of the identity function would be formalized as

$$\forall \alpha.\, \alpha \to \alpha \,,$$

taking an argument of any type α and returning a result of that same type.

[2]The mathematical concept of extensionality means that two functions produce the same output for all inputs.

Unfortunately, type inference in System F is undecidable without explicit type annotations [Wells 99]. For this reason, Hindley-Milner uses a subset of System F for a decidable type inference algorithm. This is not to say that System F is obsolete; in fact, it couldn't be further from the truth. System F can provide much more robust type inference capabilities, with the tradeoff of requiring the programmer to provide some type annotations. As a result, it is often seen in conjunction with bidirectional type inference for imperative languages.

3.5 Related work

For a more comprehensive study of λ-calculus, I found Peter Selinger's notes [Selinger 13] on the subject to be very useful, and I used them as a guide to write this section. [Church 36] is also a good introduction to the λ-calculus. [Wells 99] discusses type checking in System F and why the additional power it provides is desirable for programming languages.

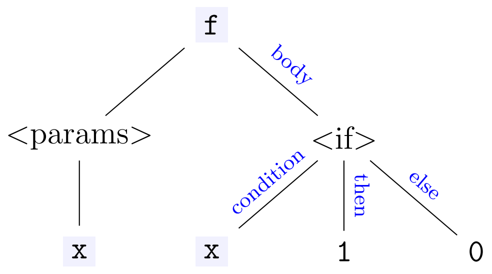
4 Motivation

In this section, I will provide a brief example of the type inference process to motivate the rest of the paper. My goal is to demystify the technique by demonstrating how simple and intuitive the basic ideas are. Consider the following function `f`, written according to functional style in various languages:

Language	**Code**
ML	`fun f x = if x then 1 else 0;`
Haskell	`f x = if x then 1 else 0`
TypeScript	`const f = (x) => x ? 1 : 0;`

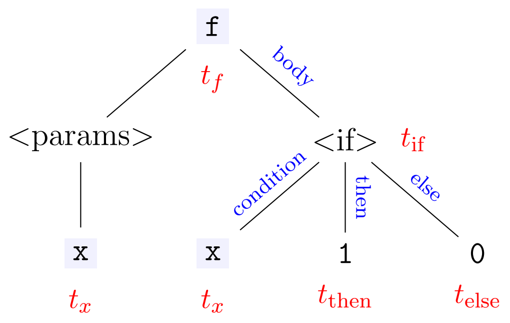
Table 1: Example function to type infer

As a human, we can easily look at the code for `f` and infer that it takes a single parameter `x`, which should be a boolean value since it is used as the condition to an *if* expression. The function will return either `1` or `0`, therefore giving it an integer return type. As a result, the function `f` has type `bool → int`. Now, how might a computer be able to analyze the code to arrive at this same conclusion? First, let's visualize the abstract syntax tree[3] (AST) that a compiler could generate for the code:



In order to infer the type of `f`, we need to also infer the type of every expression and sub-expression that appears in the tree. We can walk the tree to annotate each node with a type variable:

[3]An AST is a tree representation of the syntactic structure of code, often used during compilation.



Next, we would like to convert the annotated AST into a solvable set of type equations. This can be done analytically by walking the tree and applying the relevant type judgments, which will hopefully seem familiar from section 2.3.

First, take the following type judgment for a function:

$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x.\, e) : t_1 \to t_2}$$

The function $(\lambda x.e)$, taking a parameter x of type t_1 and returning the value of expression e with type t_2, has type $t_1 \to t_2$. In the context of our problem, the type t_f of the function will be $t_x \to t_{\text{if}}$.

Next, the type judgment for an *if* expression:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash (\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3) : t}$$

The condition expression e_1 must be a boolean, and the two expressions e_2 and e_3 must have the same type t as the overall expression. As a result, we formulate the type equations $t_x = \texttt{bool}$ and $t_{\text{if}} = t_{\text{then}} = t_{\text{else}} = t_0$, where t_0 is some arbitrary type variable to solve for.

Finally, a type judgment so obvious it nearly seems unnecessary to declare it explicitly:

$$\frac{}{0 : \texttt{int} \qquad 1 : \texttt{int} \qquad 2 : \texttt{int} \qquad \cdots}$$

The `1` and `0` in the AST are clearly integers, so we judge their types to be $t_{\text{then}} = \texttt{int}$ and $t_{\text{else}} = \texttt{int}$.

Collecting all the type equations that were generated, we have the following system:

$$\begin{aligned} t_f &= t_x \to t_{\text{if}} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= t_0 \\ t_{\text{else}} &= t_0 \\ t_{\text{if}} &= t_0 \\ t_{\text{then}} &= \texttt{int} \\ t_{\text{else}} &= \texttt{int}\,. \end{aligned}$$

This form is in fact extremely useful and can be converted to the solution for our type inference problem in one fell swoop using *unification* as shown.

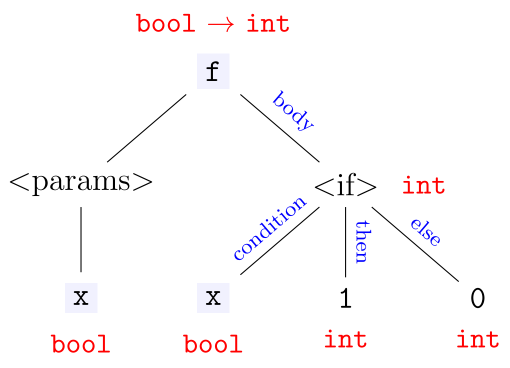
$$\begin{aligned} t_f &= t_x \to t_{\text{if}} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= t_0 \\ t_{\text{else}} &= t_0 \\ t_{\text{if}} &= t_0 \\ t_{\text{then}} &= \texttt{int} \\ t_{\text{else}} &= \texttt{int} \end{aligned}$$

$$\begin{aligned} t_f &= t_x \to t_{\text{if}} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= t_0 \\ t_{\text{else}} &= t_0 \\ t_{\text{if}} &= t_0 \\ t_0 &= \texttt{int} \\ t_0 &= \texttt{int} \end{aligned}$$

$$\begin{aligned} t_f &= t_x \to t_{\text{if}} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= \texttt{int} \\ t_{\text{else}} &= \texttt{int} \\ t_{\text{if}} &= \texttt{int} \\ \texttt{int} &= \texttt{int} \end{aligned}$$

$$\begin{aligned} t_f &= \texttt{bool} \to \texttt{int} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= \texttt{int} \\ t_{\text{else}} &= \texttt{int} \\ t_{\text{if}} &= \texttt{int} \end{aligned}$$

We can now update the AST with the types determined by the type inference process:



As we can see, the function `f` was correctly inferred to have type `bool → int`, as expected. The process we followed is known as Wand's algorithm, which will be explained in section 6.1.4.

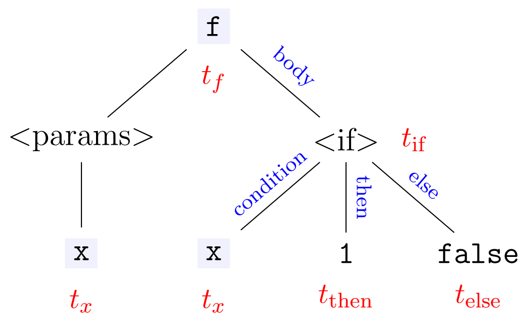
Let's also examine a similar case where type inference will fail due to a typing error:

Language	**Code**
ML	`fun f x = if x then 1 else false;`
Haskell	`f x = if x then 1 else False`
TypeScript	`const f = (x) => x ? 1 : false;`

Table 2: Function with type mismatch

In the above code, slightly modified from table 1, the function `f` will return either an `int` or a `bool`. Many languages would allow such code and perform implicit type coercion, but for the sake of the example we will remain strict and require a single return type.

As before, we visualize the AST and annotate its nodes with type variables:



Applying type judgments to each sub-expression, we find

$$\begin{aligned} t_f &= t_x \to t_{\text{if}} \\ t_x &= \texttt{bool} \\ t_{\text{then}} &= t_0 \\ t_{\text{else}} &= t_0 \\ t_{\text{if}} &= t_0 \\ t_{\text{then}} &= \texttt{int} \\ t_{\text{else}} &= \texttt{bool}\,. \end{aligned}$$

Immediately, we can see that the type equations will fail to unify due to the substitution of t_0 for t_{then} and t_{else}, leaving us with

$$\begin{aligned} &\ldots \\ t_0 &= \texttt{int} \\ t_0 &= \texttt{bool}\,, \end{aligned}$$

which leads to the faulty equation `int` $=$ `bool`. As a result, the type inference algorithm fails, detecting a type error as expected.

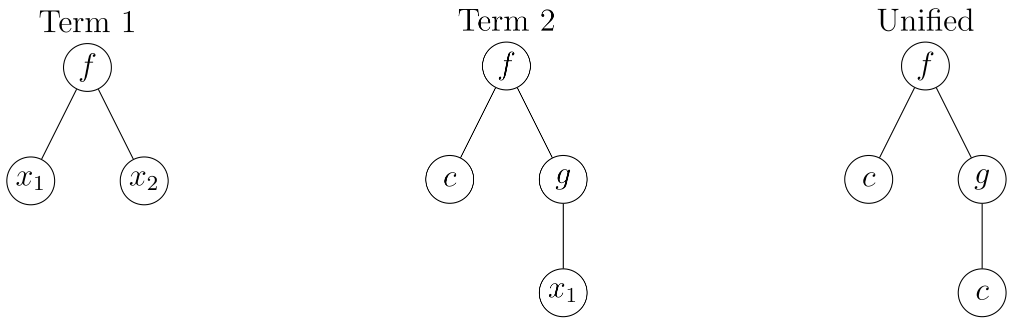
5 Unification

In this section, I will discuss the topic of unification, explaining a few practical algorithms and surveying the literature. I will examine *syntactic* unification, which is the process of making terms equal by applying some substitution (discussed in section 5.1.2). *Equational* unification (or E-unification) is very similar, but the definition of term equality is expanded using what is known as an *equational theory*. See the below table for a few examples illustrating the technique:

Terms	**Acceptable Substitution**	**Unified Term**
x_1 a	$\{x_1 \mapsto a\}$	a
x_1 x_2	$\{x_1 \mapsto x_2\}$	x_2
$f(x_1,\, b)$ $f(a,\, x_2)$	$\{x_1 \mapsto a,\, x_2 \mapsto b\}$	$f(a,\, b)$
$f(x_1,\, x_2)$ $f(c,\, g(x_1))$	$\{x_1 \mapsto c,\, x_2 \mapsto g(c)\}$	$f(c,\, g(c))$

Table 3: Basic unification examples

As illustration, the fourth example in table 3 can be represented as the following trees:



The symbol matches at the root of both trees, so it remains unchanged. Then, each sub-tree is unified so the complete trees become equivalent.

In 1965, Robinson presented his so-called resolution principle in a formal investigation of machine-oriented logic (as opposed to human-oriented logic, which utilizes much simpler inference principles). His admittedly inefficient procedure for resolution utilized the process of *unification*, which generates a substitution that makes every term equal. Unification is not limited to use in resolution—its pattern-matching ability can be applied to interpreting equation languages, automated theorem proving, and as we have seen, the process of type inference. It is an essential implementation feature of Prolog, used as a peculiar alternative to the "typical" function calls you may be used to in languages like Java and C.

5.1 Formal definitions

Literary resources vary somewhat in their representation of the unification problem and its formal preliminaries. For example, Robinson [Robinson 65] considers unification of an unbounded set of terms, Martelli and Montanari [Martelli 82] consider a pair of terms (an equation) as well as sets of equations, and Paterson and Wegman [Paterson 78] unify a single pair of terms but show that their method is equipped to easily handle a set of pairs. As a result, I will define the unification problem myself and proceed in a manner consistent with this definition.

5.1.1 Terms

Variables and *applications* are instances of *terms*, drawing a suggestive parallel to λ-calculus. An application is comparable to a function call in programming languages. In the special case of an application with arity 0, this term is known as a *constant* symbol. For example, the 0-adic function `nil` in ML is a constant symbol. A more "traditional" constant like `7` could be thought of as an implicitly-called, 0-adic function returning the integer `7`.

Let $\mathcal{V}$ be the alphabet of variable symbols $\{x_1,\, x_2,\, x_3,\, \ldots\}$, and let $\mathcal{F}$ be the set of function symbols $\{f,\, g,\, h,\, \ldots\}$. Applications can be generated by applying a function symbol $f \in \mathcal{F}$ to zero or more comma-separated terms enclosed by parenthesis. Let $\mathcal{T}$ be the set of all terms (variables and applications) that can be generated by $\mathcal{V}$ and $\mathcal{F}$. For notational convenience, constant symbols will come from the set $\{a,\, b,\, c,\, \ldots\}$ and the parenthesis will be left implicit. Some example terms include a, x_1, $f(x_1)$, $h(x_1,\, x_2)$, and $f(b,\, g(x_1,\, c))$.

5.1.2 Substitutions

A substitution σ is a mapping from variables to terms, written as $\{x_1 \mapsto t_1,\, x_2 \mapsto t_2,\, \ldots,\, x_k \mapsto t_k\}$, where $x_i \in \mathcal{V}$ and $t_i \in \mathcal{T}$ for $i = 1, 2, \ldots, k$. The application of a substitution σ to a term t is denoted $t\sigma$ and is the result of simultaneously replacing every variable x_i that appears in t and has a mapping $(x_i \mapsto t_i) \in \sigma$ with the term t_i. It can be inductively defined as follows:

$$t\sigma := \begin{cases} x & \text{if } t = x \text{ and } x \text{ does not occur in } \sigma\,, \\ t' & \text{if } (x \mapsto t') \in \sigma\,, \\ f(t_1\sigma,\, t_2\sigma,\, \ldots,\, t_n\sigma) & \text{if } t = f(t_1,\, t_2,\, \ldots,\, t_n) \text{ for integer } n \geq 0\,. \end{cases}$$

For example, consider the term $t = f(x_1,\, x_2,\, x_3)$ and the substitution $\sigma = \{x_1 \mapsto h(a, x_2),\, x_2 \mapsto x_3\}$. The application $t\sigma$ results in $f(h(a,\, x_2),\, x_3,\, x_3)$. Note that the variable x_2 appears in both t and $t\sigma$; a single application of a substitution does not necessarily eliminate every variable that has a mapping since replacements occur simultaneously.

5.1.3 Unifiers

Two terms s and t are *unifiable* if there exists a substitution σ such that $s\sigma = t\sigma$. If such a substitution exists, σ is known as a *unifier* of s and t. For example, consider the terms

$$s = f(x_1,\, g(a))\,, \text{ and}$$
$$t = f(h(b),\, x_2)\,.$$

By inspection, we can easily find the unifier $\sigma = \{x_1 \mapsto h(b),\, x_2 \mapsto g(a)\}$ that results in the term $f(h(b),\, g(a))$ when applied to s or t.

5.1.4 Most general unifier

A pair of terms may have many valid unifiers. For example, the most obvious unifier of the terms $f(x_1)$ and $f(g(x_2))$ is $\{x_1 \mapsto g(x_2)\}$. However, $\{x_1 \mapsto g(a),\, x_2 \mapsto a\}$ and $\{x_1 \mapsto g(b),\, x_2 \mapsto b\}$ are also valid unifiers. We say that the former solution is a more *general* unifier because it could reproduce the latter by additionally stipulating that $x_2 = a$ or $x_2 = b$. In fact, it also happens to be the *most general unifier*, or *mgu*, of the terms because it allows the largest possible set of unifying substitutions.

For those interested in a more formal description of the mgu, a unifier σ is an mgu if, for any other unifier θ, there exists a substitution τ such that $\sigma\tau = \theta$ ($\sigma\tau$ is called a *composition*). Most unification algorithms seen in literature are shown by their author(s) to calculate the mgu.

5.1.5 Unification problem

Unification will typically be seen as `UNIFY`$(t_1,\, t_2)$, taking an input of two terms and returning the substitution set that unifies them (or failure if they cannot be unified). It may also be seen as `UNIFY`(E), where E is a set of equations of the form $t_1 = t_2$. Although the second can unify an arbitrary-length set of equations, both forms have the exact same power. For example, unifying the set of equations $\{x_1 = a,\, x_2 = b,\, x_3 = c\}$ is identical to unifying the terms $f(x_1,\, x_2,\, x_3)$ and $f(a,\, b,\, c)$ due to the decomposition of the application into the same set of equations.

5.2 Unification algorithms

Robinson's original algorithm [Robinson 65] is known to be extremely inefficient; it does not recognize shared structure in the input and generates exponentially large structures, resulting in an exorbitant exponential runtime [de Champeaux 86]. However, it has provided the basis for many efforts in improvement. In fact, Robinson returned to the topic of efficient unification in his later work [Robinson 71] where he argued that efficiency of unification is of utmost importance for deduction algorithms. An overview of Robinson's algorithm can be found in appendix A.1.

The time complexity of Robinson's algorithm is governed by the writing and comparing of terms. [Venturini-Zilli 75] gave a simplified algorithm that reduces the amount of rewriting and bounds the number of comparisons by a quadratic polynomial, resulting in an improvement to quadratic time. Huet [Huet 76] studied higher-order unification, improving the time complexity to an almost linear algorithm. Baxter [Baxter 76] presented a nearly linear algorithm as well, showing it to be linear times a very slowly-growing function.

Paterson and Wegman [Paterson 78] defined a linear unification algorithm based on a directed acyclic graph representation of terms. However, the Paterson-Wegman algorithm is considered impractical for use in type inference systems due to the overhead of using complex data structures for unification problems that tend to be small and great in number

[Duggan 96]. Martelli and Montanari presented a linear time algorithm [Martelli 76] that was never published and later proposed another that was nearly linear [Martelli 82]. I have delved further into their nearly linear algorithm in appendix A.2, along with a practical example.

5.3 Related work

For a thorough literary survey of unification, I refer you to [Knight 89]. [Baader 01] explains both syntactic and equational unification, the history, and various algorithms (which should seem familiar).

6 Type inference

Type inference is a form of type checking that synthesizes the types of expressions from context clues available in the code. A type inference algorithm typically functions by generating constraints based on the proper typing rules and solving them with unification. My implementation of type inference for a basic language can be found here.

6.1 Hindley-Milner

The term "Hindley-Milner" is ubiquitous in literature regarding type inference due to the pair's collective influence in shaping the field. Roger Hindley wrote a paper dense with formal logic about constructing the principal type scheme of objects in combinatory logic[4] [Hindley 69]. A *principal* type scheme preserves as much generality as possible and so can be thought of as the "best" type; any other valid type scheme is a generic instance of the principal [Damas 82]. Robin Milner later explored the same topic in the context of programming languages, referring to the principal type scheme as polymorphic typing [Milner 78]. Milner's work was of a very practical nature; the type system he described was already at the time implemented as a core feature of the metalanguage ML, which was developed by researchers at the University of Edinburgh (including Milner himself) and grew into a family of functional programming languages, including SML and OCaml.

Hindley-Milner (HM), sometimes seen as Damas-Milner, is a type system for the λ-calculus that includes parametric polymorphism. It is extremely popular for functional programming languages due to the similarities between λ-calculus and the functional paradigm. Additionally, HM type inference is decidable[5]. It can infer the most general type of every variable, expression, and function in a program, without requiring any explicit type annotations. HM also supports parametric polymorphism, which grants enhanced type flexibility to the programming language [Milner 78].

Recall that a λ-calculus term may be a variable, an abstraction, or an application. We extend the basic λ-calculus terms with an additional term: the *let* expression, of the form `let x = e1 in e2`. It introduces a new variable `x` with a value of `e1`, to be used in the calculation of expression `e2`. If we consider this term further, we can see that it is a simple abstraction for $(\lambda x.\, e_2)\, e_1$.

6.1.1 Typing rules

The HM type system is built on top of a few typing rules, which follow the form described in section 2.3.

Type judgment for a variable:

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t}\,[\text{Var}]$$

[4]Combinatory logic is a branch of symbolic logic related to λ-calculus, where combinators are higher order functions that operate on other functions.

[5]Can be solved by an algorithm that is guaranteed to halt (not run indefinitely) for all inputs.

This type judgment is rather basic; it tells us that if a variable x with type t exists in the context Γ, the expression x will have type t when evaluated under the context Γ.

Type judgment for an abstraction:

$$\frac{\Gamma,\, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x.\, e) : t_1 \to t_2}\,[\text{Abs}]$$

For an abstraction, also known as a lambda expression or function, the premise dictates that the expression e must have type t_2 when evaluated under the context containing variable x with type t_1. It concludes with the realization that the function `(fn x => e)` has type $t_1 \to t_2$.

Type judgment for an application:

$$\frac{\Gamma \vdash f : t_1 \to t_2 \qquad \Gamma \vdash e : t_1}{\Gamma \vdash f\, e : t_2}\,[\text{App}]$$

An application applies the function f to expression e. If f's parameter and e have the same type t_1, the application is successful and has type t_2 matching the return type of f.

Type judgment for a *let* expression:

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma,\, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash (let\ x = e_1\ in\ e_2) : t_2}\,[\text{Let}]$$

If expression e_1 takes on type t_1 when evaluated under context Γ, x will also have type t_1. As a result, $x : t_1$ is added to Γ for the evaluation of expression e_2, which determines the type t_2 of the overall *let* expression.

As we can see, each term of the lambda calculus is accounted for by typing rules. The process of solving the typing of a program using these inference rules is known as type inference.

6.1.2 Algorithm $\mathcal{J}$

Milner introduced two equivalent algorithms for type inference [Milner 78]. The first, Algorithm $\mathcal{W}$, is defined in purely applicative (functional) form and is more conducive to formal proofs of correctness. The second, Algorithm $\mathcal{J}$, is described in imperative form and refines Algorithm $\mathcal{W}$ in ease of comprehension and practicality of implementation. The algorithm improves upon the efficiency of Algorithm $\mathcal{W}$ by applying substitutions less often. It also takes advantage of the fact that a well-typing algorithm typically does not need to produce the typing of every sub-expression present, but only the type of the supplied expression f. Since it is more practically useful, I will discuss Algorithm $\mathcal{J}$ and leave the investigation of Algorithm $\mathcal{W}$ to the interested reader. [Damas 82] analyzes Algorithm $\mathcal{W}$ to show that it always generates the principal type scheme.

Algorithm $\mathcal{J}$ constructs a single substitution, held in a global variable E. It takes as its arguments the context $\overline{p}$, which Milner calls the "typed prefix," and f, the expression to type. For consistency, I will refer to the context as Γ rather than $\overline{p}$. The algorithm makes use of the procedure `UNIFY`(σ, τ), which has no return value but updates the substitution E from the unification of type σ with type τ.

Algorithm 1 Algorithm $\mathcal{J}$

```
Require: $\Gamma$ is the assumption set (context mapping variables to type variables)
Require: $f$ is an expression
Require: $E$ is a global substitution set
procedure ALGORITHMJ($\Gamma$, $f$)                              ▷ Infers the type of expression $f$
    if $f$ has form $x$ then                                          ▷ Variable
        if $(x : t) \in \Gamma$ from an abstraction then
            return $t$
        else if $(x : t) \in \Gamma$ from a let expression then
            $t := Et$                                       ▷ Apply substitution set $E$ to $t$
            Substitute new type variables for the type variables in $t$
            return $t$
        end if
    else if $f$ has form $(e_1\, e_2)$ then                                 ▷ Application
        $t_1$ := ALGORITHMJ($\Gamma$, $e_1$)
        $t_2$ := ALGORITHMJ($\Gamma$, $e_2$)
        $t$ := fresh type variable
        UNIFY($t_1$, $t_2 \to t$)
        return $t$
    else if $f$ has form $(\lambda x.\, e)$ then                              ▷ Abstraction
        $t_x$ := fresh type variable
        $t_e$ := ALGORITHMJ(($x : t_x$, $\Gamma$), $e$)
        return $t_x \to t_e$
    else if $f$ has form ($let$ $x = e_1$ $in$ $e_2$) then                 ▷ Let expression
        $t_1$ := ALGORITHMJ($\Gamma$, $e_1$)
        $t_2$ := ALGORITHMJ(($x : t_1$, $\Gamma$), $e_2$)
        return $t_2$
    else if $f$ has form ($if$ $d$ $then$ $e_1$ $else$ $e_2$) then            ▷ If expression
        $t_d$ := ALGORITHMJ($\Gamma$, $d$)
        UNIFY($t_d$, bool)                                 ▷ Condition must be boolean
        $t_1$ := ALGORITHMJ($\Gamma$, $e_1$)
        $t_2$ := ALGORITHMJ($\Gamma$, $e_2$)
        UNIFY($t_1$, $t_2$)                         ▷ Both branches return the same type
        return $t_1$
    end if
end procedure
```

6.1.3 Algorithm $\mathcal{M}$

In contrast to the bottom-up approach of Algorithm $\mathcal{W}$ (and $\mathcal{J}$), Algorithm $\mathcal{M}$ works in a top-down manner [Lee 98]. It carries a type constraint, or expected type, that must be satisfied as it recurses downward to infer the types of sub-expressions. As a result, Algorithm $\mathcal{M}$ is context sensitive and can find type errors sooner than Algorithm $\mathcal{W}$. The procedure `UNIFY` as used in Algorithm $\mathcal{M}$ is assumed to return the substitution set that unifies the two expressions.

Algorithm 2 Algorithm $\mathcal{M}$

```
Require: $\Gamma$ is the assumption set (context mapping variables to type variables)
Require: $f$ is an expression
Require: $t$ is the type constraint (expected type) of $f$
 1: procedure AlgorithmM($\Gamma$, $f$, $t$)                ▷ Infers the type of expression $f$
 2:     if $f$ has form $x$ then                             ▷ Variable
 3:         $t_x := \Gamma(x)$
 4:         Substitute new type variables for the type variables in $t_x$
 5:         return Unify($t$, $t_x$)
 6:     else if $f$ has form $(e_1\, e_2)$ then              ▷ Application
 7:         $t_2$ := fresh type variable
 8:         $S_1$ := AlgorithmM($\Gamma$, $e_1$, $t_2 \to t$)
 9:         $S_2$ := AlgorithmM($S_1\Gamma$, $e_2$, $S_1 t_2$)
10:         return $S_2 S_1$
11:     else if $f$ has form $(\lambda x.\, e)$ then         ▷ Abstraction
12:         $t_x$, $t_e$ := fresh type variables
13:         $S_1$ := Unify($t$, $t_x \to t_e$)
14:         $S_2$ := AlgorithmM(($S_1\Gamma$, $x : S_1 t_x$), $e$, $S_1 t_e$)
15:         return $S_2 S_1$
16:     else if $f$ has form $(let\ x = e_1\ in\ e_2)$ then  ▷ Let expression
17:         $t_x$ := fresh type variable
18:         $S_1$ := AlgorithmM($\Gamma$, $e_1$, $t_x$)
19:         $S_2$ := AlgorithmM(($S_1\Gamma, x : S_1 t_x$), $e_2$, $S_1 t$)
20:         return $S_2 S_1$
21:     end if
22: end procedure
```

6.1.4 Wand's Algorithm

Mitchell Wand's type inference algorithm [Wand 87b] is characterized by its generation of conditions (type equations), as we recall from section 4. It works like a verification-condition generator[6], emitting conditions that will derive the term's type. If the equations produced can be unified, the mgu (discussed in section 5.1.4) contains the principal type of the term and its sub-terms. Wand's algorithm is as follows:

Algorithm 3 Wand's type inference algorithm

```
Require: $M_0$ is a well-formed term of the $\lambda$-calculus
Require: $\Gamma$ is the assumption set (context mapping variables to type variables)
 1: procedure TypeInference($M_0$)                        ▷ Infers the type of term $M_0$
 2:     $E := \emptyset$                                   ▷ Start with empty equation set
 3:     $t_0$ := fresh type variable                       ▷ Type of $M_0$
 4:     $G := \{\Gamma \vdash M_0 : t_0\}$                 ▷ Initial subgoal set
 5:     repeat
 6:         $g := G$.remove()                              ▷ Select and delete a subgoal
 7:         if $g$ has form $\Gamma \vdash x : t$ then     ▷ Variable
 8:             $E$.add($t = \Gamma(x)$)
 9:         else if $g$ has form $\Gamma \vdash (e_1\, e_2) : t$ then   ▷ Application
10:             $t_2$ := fresh type variable
11:             $G$.add($\Gamma \vdash e_1 : t_2 \rightarrow t$)
12:             $G$.add($\Gamma \vdash e_2 : t_2$)
13:         else if $g$ has form $\Gamma \vdash (\lambda x.\, e) : t$ then   ▷ Abstraction
14:             $t_x$, $t_e$ := fresh type variables
15:             $E$.add($t = t_x \rightarrow t_e$)
16:             $G$.add($\Gamma, x : t_x \vdash e : t_e$)
17:         end if
18:     until $G = \emptyset$                              ▷ No more subgoals to satisfy
19:     Unify($E$)                                         ▷ Unify the type equations
20:     return $E(t_0)$
21: end procedure
```

Wand's algorithm allows flexibility in precisely how equations and subgoals are generated on each step, simply requiring that the actions taken guarantee termination and satisfy some "invariant" (a concept introduced to ensure correct typing) at each step. The actions associated with each case are extremely logical, deriving naturally from the type judgment for each term of the λ-calculus. The process could be extended to support extensions based on derived type rules, such as for the *let* expression. Since each action generates subgoals smaller than the original, the algorithm is guaranteed to converge toward $G = \emptyset$ and terminate.

[6]A tool commonly found in automated program verification, used to output formal conditions that must be satisfied to ensure correctness.

For example, we execute Wand's algorithm on the term $M_0 = \lambda x.\, \lambda y.\, xy$, beginning from the goal $\{\Gamma \vdash M_0 : t_0\}$ with initial context $\Gamma = \emptyset$:

Loop	**g**	**Form**	**E**	**G**
1	$\{\} \vdash (\lambda x.\, \lambda y.\, xy) : t_0$	Abstraction	$t_0 = t_1 \to t_2$	$\{x : t_1\} \vdash (\lambda y.\, xy) : t_2$
2	$\{x : t_1\} \vdash (\lambda y.\, xy) : t_2$	Abstraction	$t_0 = t_1 \to t_2,$ $t_2 = t_3 \to t_4$	$\{x : t_1,\, y : t_3\} \vdash xy : t_4$
3	$\{x : t_1,\, y : t_3\} \vdash (xy) : t_4$	Application	$t_0 = t_1 \to t_2,$ $t_2 = t_3 \to t_4$	$\{x : t_1,\, y : t_3\} \vdash x : t_5 \to t_4,$ $\{x : t_1,\, y : t_3\} \vdash y : t_5$
4	$\{\underline{x : t_1},\, y : t_3\} \vdash x : t_5 \to t_4$	Variable	$t_0 = t_1 \to t_2,$ $t_2 = t_3 \to t_4,$ $t_1 = t_5 \to t_4$	$\{x : t_1,\, y : t_3\} \vdash y : t_5$
5	$\{x : t_1,\, \underline{y : t_3}\} \vdash y : t_5$	Variable	$t_0 = t_1 \to t_2,$ $t_2 = t_3 \to t_4,$ $t_1 = t_5 \to t_4,$ $t_3 = t_5$	

Table 4: Wand Algorithm actions

When the loop ends due to the empty goal set G, we unify the resulting equations in E to find that $t_0 = (t_3 \to t_4) \to t_3 \to t_4$. Therefore, term M_0 has principal type $(\alpha \to \beta) \to \alpha \to \beta$.

6.1.5 Comparison

A major critique of Milner's Algorithm $\mathcal{J}$ has been the difficulty it produces in determining the cause of type errors [MacQueen 20]. Depending on the implementation of the algorithm, type checkers typically detect an error when two types conflict, which may occur after a long, implicit flow of type information. The location of a conflict is not always indicative of the true source of an error, which can be especially confusing to less experienced programmers. This is caused by the context-insensitive, bottom-up approach of the algorithm, as opposed to the context-sensitive, top-down approach of Algorithm $\mathcal{M}$ [Lee 98]. Algorithm $\mathcal{M}$ can detect a type error sooner and more precisely localize the source.

The algorithms $\mathcal{W}$ and $\mathcal{J}$ provided by [Milner 78] as well as Algorithm $\mathcal{M}$ from [Lee 98] interleave the production of type equations with the solution, using the unification procedure several times throughout. In contrast, the algorithm of [Wand 87b] generates the complete set of type equations for the term before unifying.

6.2 Bidirectional typing

Bidirectional typing is called so due to the bidirectional flow of type information. It consists of two modes: type synthesis (inference) and type checking. The concept originated in [Pierce 00], which proposed a technique for local type inference. Local type inference infers the locally best types and allows the types of parameters to anonymous functions to be inferred. The bidirectional technique lends itself to concepts such as subtyping and polymorphism more freely than HM [Dunfield 21]. Local type inference can be implemented with relative simplicity, with an algorithm that is only a little more complicated than for a Hindley-Milner type system [Jones 07].

7 Extensions to type inference

In his original paper, Milner [Milner 78] discusses several potential extensions to the "basic" HM type inference. In some cases, the type constraints enforced are too restrictive and prevent a semantically correct term from type checking [Coppo 80]. I will explore various extensions to the HM system of type inference in this section.

7.1 Overloading

Ad hoc polymorphism, more commonly known as *overloading*, is when a function is defined over several different types, acting in a different way for each. The HM type system accounts for parametric polymorphism, but lacks support for other forms such as type coercion and overloading. For example, in many programming languages `0` is treated as falsy and non-zero values as truthy. This is an instance of coercion from an integer to a boolean. It is a useful feature, but one that is forbidden under classical HM. Overloading of operators is another useful feature that is forbidden by HM. For example, the `+` operator should be overloaded for `int + int → int` and `real + real → real`, but cannot be as general as $\alpha + \alpha \to \alpha$ since not all types are suitable for addition.

Kaes approached overloading through a system of *constrained types* [Kaes 92]. His system adds a constraint C to type judgments of the form $C, \Gamma \vdash M : \tau$. These type judgments say that M has type τ under the assumptions Γ, provided that C is satisfiable. He provides an algorithm and type unification procedure, although the algorithm is not complete for recursive types.

Wadler and Blott [Wadler 89] introduced a method of providing overloading for ML using type classes. At the time of writing, there was no widely accepted approach to overloading, and a variety of solutions have been applied in different languages.

ML takes the approach of overloading built-in operators but not functions. For example, the function `fun square x = x * x` has type `int → int`. If the type of `x` were specified as `real`, however, it would have type `real → real`. As we can see, the `*` operator is overloaded for both integer and real types, but it is always constrained to one or the other when actually used. If the use of an overloaded operator is ambiguous, ML will select its default type to avoid complications. ML also provides limited polymorphism for checking equality by defining equality types (shortened to *eqtype*, written as `''a`). This allows the type of the equality operator to be defined as `''a → ''a → bool`, providing polymetric polymorphism without being overly permissive.

The type classes proposed by Wadler and Blott would be implemented by passing dictionaries to function calls. For example, the `square` example would become a function taking two arguments of type class `Num`. The types `Int` and `Float` would be instances of `Num` and define the necessary operations of `+` and `*`, allowing the function to accept either integer or float values and select the correct overloaded method.

Although Wadler and Blott's method was not applied to ML, it has been implemented in Haskell. It has an advantage over Kaes's work in its easier notation and conceptualization, and it is more general. However, it does have the drawback of introducing new parameters in the form of method dictionaries that must be passed at runtime. The work also allows certain nonsensical expressions and can be improved [Smith 93].

7.2 Subtyping

The classical HM system does not support a means of providing subtyping. However, ideas have been proposed to extend the system with this useful feature. Several authors treat type inference with subtyping but do not include a *let* expression, the presence of which increases the complexity of proving an inference algorithm complete [Smith 93].

In one such approach to subtyping, [Wand 87a] supplies a type inference algorithm providing inheritance through record structures, based on an extension of type inference. In another work, Fuh and Mishra [Fuh 90] develop a general framework for type inference in the presence of subtypes, making minimal assumptions about the particular subtype theory. The problem presented by subtypes in type inference is that it is no longer possible to represent the principal type scheme by a type expression alone. The solution consists of a set of coercion statements $t_i \rhd t_j$, where t_i is a subtype of t_j in addition to the type expression. The coercion set can be collected while carrying out the type inference process normally, and any substitution that satisfies the coercion set can be applied to the type expression produced.

[Smith 93] applies *constraints* in order to define the set of quantified types. Each constraint may be a typing $x : t$ or an inclusion[7] $t_i \rhd t_j$, such as $\texttt{int} \rhd \texttt{real}$. The assumption set Γ contains both identifier typings (as is standard) as well as inclusions. In the context of type inference, inclusions must have the same "shape" on either side; for example, the assumption $\texttt{int} \rhd (\texttt{int} \to \texttt{int})$ is nonsensical and would be rejected because the right-hand type is impossible to coerce into the same type as that on the left. Inclusions also cannot be cyclic, as in $\texttt{bool} \rhd \texttt{int}$ and $\texttt{int} \rhd \texttt{bool}$. Additionally, Smith disallows typings with unsatisfiable constraint sets since such types are *vacuous*, having no valid instances. He presents a modified version of Algorithm $\mathcal{W}$, updated to provide overloading and subtyping.

[Mitchell 91] describes algorithms that could be used to extend ML with simple forms of subtyping and coercion. [Cardelli 88] claims that type checking is easier for structural types since the relationship between similar structures can be more easily seen. [Simonet 03] investigates type inference in the presence of structural subtyping, using interwoven unification processes on the structures and terms present in the code.

7.3 Reference types

HM typing applies well to the functional programming paradigm; however, difficulties arise when implementing inference for imperative features such as assignable variables, which add state. For example, consider the classic identity function $I = (\lambda x.\, x)$ with type $(\forall \alpha.\, \alpha \to \alpha)$, with a twist: it assigns the value of x to a reference variable y on each call. Naturally, the type of y must match that of x, which is α. However, previous well-typing rules would not suffice in the scenario of consecutive calls of $I(3)$ and $I(\mathit{true})$, since y is instantiated as an integer but proceeds to take on a boolean value. Another example, given in [Tofte 90], illustrates the problem with polymorphic references:

[7]Type t_i is *included in* type t_j if the set of values of type t_i is a subset of those of t_j, and hence an inclusion describes the subtype relation.

```
let val r = ref (fn x => x)
in
    r := (fn n => n + 1);
    (!r) true
end;
```

Both instances of accessing `r` may appear sound according to its `'a → 'a` type, but in combination they result in an error once the type of `r` narrows to `int → int` and is applied to a `bool`.

[Wright 95] presents a simple solution for imperative typing features, at the cost of being able to type all HM-typable expressions. [King 06] investigates the successes and pitfalls of polymorphic references and presents a type system that extends polymorphism to reference cells.

7.4 Objects

Wand states that his result could be easily extended to type systems with products, disjoint unions, and other constructs [Wand 87b]. By extending it to handle labelled products and unions, one could handle certain kinds of object-oriented programming. Wand makes good on this assertion in his type system for simple objects [Wand 87a]. Since records with named fields can be seen as an extension of pairs, it was easy to extend type inference to handle simple data structures. [Wand 91] further extends the bounds of type inference to handle objects with instance variables and methods.

8 Practical

In this section, I will discuss how type inference has been applied to real-world programming languages.

8.1 ML

ML was originally developed as a metalanguage for theorem proving at the University of Edinburgh in the 1970s. At the time of Milner's theory of type polymorphism, the type checking/inferring Algorithm $\mathcal{W}$ had been in use successfully for nearly 2 years in Edinburgh's LCF theorem proving system [Milner 78]. Nowadays, ML is seen by some as a relic of the past, unfit for use in the modern age. However, ML has undoubtedly had tremendous influence in the world of programming languages, particularly through its use of polymorphism (a feature now taken for granted in functional languages) and type inference. Standard ML was the first statically-typed functional language to gain broader use, and it has served commendably as a vehicle for research on programming language design [MacQueen 20]. Ideas from ML have influenced several other languages, including Haskell, C++, Scala, and Elm.

ML is a implicitly typed and uses Hindley-Milner type inference to guarantee the well-typedness of programs. Occasionally it becomes necessary to help the compiler along with a few type annotations; however, the majority of ML code can be written without them.

A major feature offered by ML is parametric polymorphism. For example, the built-in function `map` has type `('a → 'b) → 'a list → 'b list`. A type beginning with a single quote `'` is a type variable that can be instantiated to any type. As discussed in section 7.1, ML extends the polymorphism of the Hindley-Milner system with equality types, represented with a type variable of the form `''a`.

Reference cells extend ML so that it is not purely functional. The handling of reference cells presents great difficulty to the type inference algorithm in the presence of polymorphism. The issue is handled in various ways in the different flavors of ML; for example, Standard ML differentiates *imperative* type variables from regular *applicative* type variables.

ML does not permit overloading of functions, but some built-in operators are overloaded, such as `+` for `int` and `real` values. The addition of overloaded operators does not overly complicate the type inference process because ML will select a default overload unless the context requires otherwise. ML also lacks subtyping and implicit type coercions. For example, the expression `5 + 3.14` will fail since `5` is not coerced to a real, and `int * real` is not a valid domain for the `+` operator.

8.2 TypeScript

TypeScript was first released in October of 2012 and is developed and maintained by Microsoft. In the decade since its release, TypeScript has exploded in popularity with developers. The type safety provided and in-depth tooling with code completion and error messages are particularly attractive benefits of using TypeScript.

According to its documentation, the primary objective of TypeScript is "to be a static typechecker for JavaScript programs." TypeScript acts as an extension to JavaScript, which only provides dynamic typing. TypeScript code is transpiled[8] to the equivalent JavaScript code that can be interpreted by the browser. The TypeScript compiler offers hundreds of configuration options, affecting behavior like the strictness of type checking.

TypeScript boasts a delightful array of typing features, such as union types, generics (parametric polymorphism), overloading, and more. Users can define their own types and interfaces for describing the shapes and behaviors of objects. TypeScript uses structural subtyping to determine type compatibility. The language also provides several powerful methods of manipulating types—for example, conditional and mapped types.

TypeScript utilizes bidirectional type checking, which is very robust and sacrifices certain features of Hindley-Milner in exchange for enhanced flexibility in practical matters like subtyping, which HM struggles with. Type inference is applied in several locations, such as in variable assignment and determining function return types. When applicable, TypeScript will calculate the best common type (a union of possible types) of an expression. The types of function parameters must be noted explicitly (or they are assumed to have type `any`, which renders type checking ineffectual). However, TypeScript performs contextual typing[9] for arguments to functions, unifying their type with the type of the associated parameter. This can be incredibly useful when writing anonymous functions, since it will infer the types of the arguments (which are sometimes rather complex and cumbersome to import and type out) and verify the return type.

When I was first introduced to TypeScript, I was stunned by the power of type narrowing. The language recognizes conditionals that narrow the possible set of types a value could have, known as *type guards*. TypeScript performs control flow analysis (CFA) that can determine a variable's type, or even its exact value, in each possible branch. It can also warn the user about branches that will always/never be taken.

A 2017 study found that that a static type system for JavaScript such as TypeScript or Flow (developed by Facebook) could have detected and prevented 15% of bugs in public software projects [Gao 17]. This percentage under-approximates the full positive impact that static type systems have on code due to their aid in documentation, code completion, code navigation, and bug detection during private development. The study also found that TypeScript's strict null checking gave rise to a substantial increase in bug detection.

8.3 Python

Python is widely known for its dynamic typing, which comes along with serious type safety concerns, especially in projects of larger scale. However, several options exist for type checking Python code. Among these are programs such as Mypy, Pyright, and Pytype. Mypy was developed at Dropbox in collaboration with Guido van Rossum and has been used to type check millions of lines of code. Pyright is the Python type checker devised by Microsoft, which integrates conveniently with VS Code through the Pylance extension. Each type checker comes with its own set of features but performs a similar function of providing type-related feedback using optional type annotations and bidirectional type inference.

Python 3.5 introduced the ability to write type hints by adding the built-in library

[8]Transpiling translates source code from one programming language to another.

[9]When the type of an expression is implied by its location.

`typing` to support gradual typing as defined by PEP 484. Since then, many enhancements have been made to further support type checkers for Python. PEP 586 added literal types, which enhances type narrowing. For example, if a function parameter `option` is meant to take on either `"r"` or `"w"`, a type of `Literal["r", "w"]` is more precise and informational to a programmer than `str`, and it can be used to verify that the function is called with a valid string. PEP 647 introduced the `TypeGuard` construct, which can be used to write user-defined functions that ensure a value is of a particular type and propagate this type narrowing to the type checker.

My own experience using Pyright (through Pylance) in VS Code has been excellent. I find the code completion suggestions and informational popups on hover to be invaluable. Additionally, the editor notifies me of any type safety issues that I may have missed or created through my changes.

8.4 Java

Java is statically typed, and it is somewhat infamous for requiring explicit type annotations that can be very complex and unwieldy. Beginning in Java 5, though, support for certain type inference features has been progressively added to the language. For example, type parameters of constructors can be inferred as in `List<String> list = new ArrayList<>()`, whereas previously the generic argument `String` would have appeared between the angle brackets on both sides. Java's lambda expressions have their type inferred from context, allowing the parameters to be written without explicit types. The `var` keyword tells Java to perform local type inference to determine the type of a variable without an explicit type annotation.

Although Java is still largely a type-annotated language, it serves as an illustration of the quality-of-life improvements that can be gained through type inference. It allows code to be written in a more concise and readable manner by eliminating extraneous annotations.

8.5 Others

Many other languages offer some degree of type inference. Haskell, like ML uses Hindley-Milner type inference. C++, like Java, is largely type-annotated but can be told to infer the type of a variable using the `auto` keyword. Scala and Kotlin both utilize local type inference.

8.6 Related work

[MacQueen 20] is a fascinating history of Standard ML and its context, implementation, and impact. This article discusses a few type checkers for Python and their differing attributes. Java's type inference offerings are described in this article.

9 Conclusion

Lambda calculus is used as a vehicle for the formal study of type inference. Since it so closely resembles functional programming languages, conclusions drawn for the type inference of the λ-calculus can be directly applied to programming languages. We familiarized ourselves with the fundamentals of λ-calculus in order to take advantage of its use as seen in papers regarding type inference.

The possible values and behavior of each type is a key factor in type inference, since we take context clues from the operations applied to expressions to derive their type. Additionally, some expressions are polymorphic in type and may take on any from a range of possible types. Having this flexibility is a valuable tool that we wish to preserve during type inference.

We motivated the type inference process using a basic example and intuitively demonstrating how we could determine its type. Unification was seen as an integral part of the algorithm, used to match up the types that were applied to expressions using appropriate typing rules. Since unification is a critical element of type inference, we examined the unification problem and the many efforts that have been made to improve the runtime of unification algorithms.

With a solid basis in λ-calculus, types, and unification, we were prepared to explore the Hindley-Milner type system and its typing rules. HM typing infers types for the λ-calculus, and hence for functional programming languages as well. We explored a few algorithms for HM type inference and their differences. Hindley-Milner has a single flow of typing information; in contrast, bidirectional typing allows information to flow both ways and has found widespread use in providing type inference for imperative programming languages. Although the basic HM type system provides parametric polymorphism, it is quite limited in that it does not allow overloading, subtyping, and imperative-style programming through reference variables. As a result, we investigated various ways HM can be extended to handle a wider assortment of features.

Type inference has been applied to many real-world programming languages, some of which you have likely used. The foremost example is ML, which uses type inference as its built-in type checker. Type inference can even be applied to dynamically typed languages like JavaScript and Python, making it an invaluable tool for ensuring type safety when the language itself does not provide it. It can also be used in languages that do use type annotations, offering a way to delegate the work of type declaration to the language.

References

[Baader 01] Franz Baader & Wayne Snyder. *Unification theory*, 12 2001.

[Baxter 76] Lewis Denver Baxter. *The complexity of unification*, 1976.

[Cardelli 85] Luca Cardelli & Peter Wegner. *On understanding types, data abstraction, and polymorphism.* Computing Surveys, vol. 17, pages 471–523, 12 1985.

[Cardelli 88] Luca Cardelli. *Structural subtyping and the notion of power type.* pages 70–79. Association for Computing Machinery, 1988.

[Church 36] Alonzo Church. *An unsolvable problem of elementary number theory.* American Journal of Mathematics, vol. 58, pages 345–363, 4 1936.

[Coppo 80] Mario Coppo. *An extended polymorphic type system for applicative languages.* pages 194–204. Springer Berlin Heidelberg, 1980.

[Damas 82] Luis Damas & Robin Milner. *Principal type-schemes for functional programs.* pages 207–212. ACM, 1982.

[de Champeaux 86] Dennis de Champeaux. *About the Paterson-Wegman linear unification algorithm.* Journal of Computer and System Sciences, vol. 32, pages 79–90, 2 1986.

[Duggan 96] Dominic Duggan & Frederick Bent. *Explaining type inference.* Sci. Comput. Program., vol. 27, pages 37–83, 1996.

[Dunfield 21] Jana Dunfield & Neel Krishnaswami. *Bidirectional typing.* ACM Comput. Surv., vol. 54, 5 2021.

[Fuh 90] You Chin Fuh & Prateek Mishra. *Type inference with subtypes.* Theoretical Computer Science, vol. 73, pages 155–175, 6 1990.

[Gao 17] Zheng Gao, Christian Bird & Earl T Barr. *To type or not to type: quantifying detectable bugs in JavaScript.* pages 758–769. IEEE Press, 2017.

[Hindley 69] Roger Hindley. *The principal type-scheme of an object in combinatory logic.* Transactions of the American Mathematical Society, vol. 146, pages 29–60, 12 1969.

[Huet 76] Gérard Huet. *Resolution d'equations dans les langages d'ordre 1, 2, ..., omega*, 3 1976.

[Jones 07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich & Mark Shields. *Practical type inference for arbitrary-rank types.* Journal of Functional Programming, vol. 17, pages 1–82, 2007.

[Kaes 92] Stefan Kaes. *Type inference in the presence of overloading, subtyping and recursive types.* 1992.

[King 06] Dave King & John Hannan. *Reference polymorphism in ML.* 2006.

[Knight 89] Kevin Knight. *Unification: a multidisciplinary survey.* ACM Computing Surveys, vol. 21, pages 93–124, 3 1989.

[Lee 98] Oukseh Lee & Kwangkeun Yi. *Proofs about a folklore let-polymorphic type inference algorithm.* ACM Trans. Program. Lang. Syst., vol. 20, pages 707–723, 7 1998.

[MacQueen 20] David MacQueen, Robert Harper & John Reppy. *The history of standard ML.* Proc. ACM Program. Lang., vol. 4, 6 2020.

[Martelli 76] Alberto Martelli & Ugo Montanari. *Unification in linear time and space: a structured presentation.* 1976.

[Martelli 82] Alberto Martelli & Ugo Montanari. *An efficient unification algorithm.* ACM Transactions on Programming Languages and Systems, vol. 4, pages 258–282, 4 1982.

[Milner 78] Robin Milner. *A theory of type polymorphism in programming.* Journal of Computer and System Sciences, vol. 17, pages 348–375, 12 1978.

[Mitchell 91] John C Mitchell. *Type inference with simple subtypes.* Journal of Functional Programming, vol. 1, pages 245–285, 1991.

[Morris 69] James H. Morris. *Lambda-calculus models of programming languages*, 1969.

[Paterson 78] M. S. Paterson & M. N. Wegman. *Linear unification.* Journal of Computer and System Sciences, vol. 16, pages 158–167, 4 1978.

[Pierce 00] Benjamin C Pierce & David N Turner. *Local type inference.* ACM Trans. Program. Lang. Syst., vol. 22, pages 1–44, 1 2000.

[Robinson 65] John Alan Robinson. *A machine-oriented logic based on the resolution principle.* Journal of the ACM, vol. 12, pages 23–41, 1 1965.

[Robinson 71] John Alan Robinson. *Computational logic: the unification computation.* Machine Intelligence, vol. 6, pages 63–72, 1971.

[Selinger 13] Peter Selinger. *Lecture notes on the lambda calculus*, 2013.

[Simonet 03] Vincent Simonet. *Type inference with structural subtyping: A faithful formalization of an efficient constraint solver.* pages 283–302. Springer, 2003.

[Smith 93] Geoffrey S. Smith. *Polymorphic type inference with overloading and subtyping*. 1993.

[Tofte 90] Mads Tofte. *Type inference for polymorphic references*. Information and Computation, vol. 89, pages 1–34, 1990.

[Venturini-Zilli 75] M Venturini-Zilli. *Complexity of the unification algorithm for first-order expressions*. CALCOLO, vol. 12, pages 361–371, 1975.

[Wadler 89] Philip Wadler & Stephen Blott. *How to make ad-hoc polymorphism less ad-hoc*. pages 60–76. Association for Computing Machinery, 1989.

[Wand 87a] Mitchell Wand. *Complete type inference for simple objects*. 1987.

[Wand 87b] Mitchell Wand. *A simple algorithm and proof for type inference*. Fundamenta Informaticae, vol. 10, pages 115–121, 4 1987.

[Wand 91] Mitchell Wand. *Type inference for record concatenation and multiple inheritance*. Information and Computation, vol. 93, pages 1–15, 1991.

[Wells 99] J B Wells. *Typability and type checking in System F are equivalent and undecidable*. Annals of Pure and Applied Logic, vol. 98, pages 111–156, 1999.

[Wright 95] Andrew K Wright. *Simple imperative polymorphism*. LISP and Symbolic Computation, vol. 8, pages 343–355, 1995.

A Unification algorithms

A.1 Robinson

The crude algorithm given by Robinson [Robinson 65] in his original treatment of unification is given as follows:

Algorithm 4 Robinson's 1965 unification algorithm

```
Require: $A$ is a finite, nonempty set of well-formed terms
 1: procedure UNIFY($A$)                      ▷ Substitution that unifies the set of terms $A$
 2:     $\sigma := \emptyset$                  ▷ Start with empty substitution mapping
 3:     loop                                   ▷ forever
 4:         if $A\sigma$ is a singleton then
 5:             return $\sigma$                ▷ $\sigma$ unifies every term in $A$
 6:         end if
 7:         $B$ := lexical ordering of DISAGREEMENTSET($A\sigma$)
 8:         $t_1$ := first element of $B$
 9:         $t_2$ := second element of $B$
10:         if $t_1$ is a variable and $t_1$ does not occur in $t_2$ then
11:             $\sigma := \sigma\{t_1 \mapsto t_2\}$   ▷ Composition with the new mapping $t_1 \mapsto t_2$
12:         else
13:             Fail.
14:         end if
15:     end loop
16: end procedure
```

As seen on line 5, Robinson's algorithm terminates successfully once a substitution σ is found that unifies every term in A to a single term. It makes use of a few constructs that have not been mentioned. The *disagreement set* is the set of well-formed sub-expressions in A at the first position where they differ. For example,

$$\begin{aligned} A &= \{f(x_1, h(x_1, x_2), x_2), f(x_1, k(x_2), x_2), f(x_1, a, b)\}\,, \\ \text{Disagreement set of } A &= \{h(x_1, x_2), k(x_2), a\}\,. \end{aligned}$$

In the *lexical ordering*, variables precede the other terms. As a result, the lexical ordering of the disagreement set gives us the list of sub-terms to unify, starting with the variables if any exist. Another crucial part of algorithm 4 is the *occurs check* on line 10. If t_1 were to occur as a sub-term of t_2, unification would lead to an infinite term in the mapping for t_1, so it fails instead.

My Python implementation of Robinson's unification algorithm can be found here.

A.2 Martelli and Montanari

In the following algorithm described by Martelli and Montanari [Martelli 82], x matches a variable and a t matches any term (variable, application, or constant) unless otherwise specified. Given a set of equations S, where an equation is a pair of terms $t_1 = t_2$ to unify, repeatedly select an equation matching a form in table 5 and apply the associated rule.

Rule	**Form**	**Action**
Swap	$t = x$, where t is not a variable	Rewrite as $x = t$.
Erase	$x = x$	Erase the equation.
Symbol clash	$t' = t''$, where t' and t'' are not variables, and the root function symbols are different	$\perp$
Decompose	$t' = t''$, where t' and t'' are not variables, and the root function symbols are the same	Apply term reduction.
Occurs check	$x = t$, where x occurs in t	$\perp$
Variable elimination	$x = t$, where x occurs elsewhere in S and x does not occur in t	Apply variable elimination.

Table 5: Efficient unification algorithm

When no more transformations apply, S has been successfully unified. The action $\perp$ signifies that the algorithm terminates in failure (S not unifiable).

Martelli and Montanari's algorithm introduces two new processes that I have not explained: *term reduction* and *variable elimination*. A term reduction takes two applications and decomposes them by eliminating their shared function symbol and creating a new equation for each pair of arguments. A variable elimination eliminates every instance of the lefthand variable x in S by replacing x with the righthand term for every equation.

My Python implementation of the algorithm can be found here. The algorithm described is the first presented by [Martelli 82]; the second and third algorithms, implemented here and here, approach (near) linear runtime efficiency.

For an illustration of the algorithm in practice, consider the example set of equations seen in [Martelli 82]:

$$g(x_2) = x_1\,;$$
$$f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3)\,.$$

Say we choose the second equation first; it matches the form $t' = t''$ where the root symbols are both f (**decompose** rule). We decompose this equation by generating a new equation for each argument of the two applications, resulting in

$$\begin{aligned} g(x_2) &= x_1\,; \\ x_1 &= g(x_3)\,; \\ h(x_1) &= x_4\,; \\ x_2 &= x_3\,. \end{aligned}$$

Next, choose the second equation, $x_1 = g(x_3)$. It matches the form $x = t$ and passes the occurs check (**variable elimination** rule). Applying the substitution $\{x_1 \mapsto g(x_3)\}$ to the other equations leaves us with

$$\begin{aligned} g(x_2) &= g(x_3)\,; \\ x_1 &= g(x_3)\,; \\ h(g(x_3)) &= x_4\,; \\ x_2 &= x_3\,. \end{aligned}$$

We apply the **decompose** rule to the first equation, which has the form $t' = t''$.

$$\begin{aligned} x_2 &= x_3\,; \\ x_1 &= g(x_3)\,; \\ h(g(x_3)) &= x_4\,; \\ x_2 &= x_3\,. \end{aligned}$$

The third equation is of form $t = x$, so we apply the **swap** rule:

$$\begin{aligned} x_2 &= x_3\,; \\ x_1 &= g(x_3)\,; \\ x_4 &= h(g(x_3))\,; \\ x_2 &= x_3\,. \end{aligned}$$

Select the first equation, which has the form $x = t$, and apply the **variable elimination** rule:

$$\begin{aligned} x_2 &= x_3\,; \\ x_1 &= g(x_3)\,; \\ x_4 &= h(g(x_3))\,; \\ x_3 &= x_3\,. \end{aligned}$$

Finally, we see that the last equation has form $x = x$ and apply the **erase** rule:

$$\begin{aligned} x_2 &= x_3\,; \\ x_1 &= g(x_3)\,; \\ x_4 &= h(g(x_3))\,. \end{aligned}$$

There are no longer any possible transformations to apply, so we have found the mgu of the initial set of equations.