UNIFICATION SOURCE-TRACKING WITH APPLICATION TO DIAGNOSIS OF TYPE INFERENCE

Venkatesh Choppella

Computer Science Department

Indiana University

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

August 2002

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Christopher T. Haynes, Ph.D.
(Principal Advisor)

Daniel P. Friedman, Ph.D.

Steven D. Johnson, Ph.D.

Paul W. Purdom, Ph.D.

July 10th, 2002

Lawrence S. Moss, Ph.D.

To my mother Srimati Sundari, and father Sri Choppella Venkata Ramachandra Murti, and to the memory of my grandmother Srimati Parupalli Sitaratnam

Acknowledgements

I owe a great debt of gratitude to Chris Haynes, my thesis advisor. This thesis would not have been accomplished without his enthusiastic support and encouragement. His unfailing confidence in my abilities steered me through the many dark corners and dead ends that are an inevitable part of research.

I was fortunate to have a strong and supportive research committee. I owe a special debt of gratitude to Larry Moss, who taught me co-induction and offered crucial advice on many technical issues. The flat system formalism used in this thesis is inspired by his work. Steve Johnson introduced me to theorem proving and gave me the opportunity to hang out in his lab and collaborate with his students. Dan Friedman taught me what to value in programming. Paul Purdom helped me understand several points about unification and rewrite systems. Each of them read my thesis with critical attention and offered valuable advice. To all of them, my deep-felt gratitude.

I wish to thank Gregor Kiczales and his Aspect-Oriented Programming group for the very enjoyable time I spent at the Xerox Palo Alto Research Center. Aspects provided a fresh perspective to my thinking on the problem of debugging. At a time when people were being lured into Silicon Valley startups, Gregor encouraged me to instead head back to school and attend to my unfinished dissertation. I thank him for his sound advice.

I appreciate and thank the administrative and systems staff of the computer science department at IU for going out of their way to help me on many occasions. They clearly stand out as among the best anywhere.

My stay in the beautiful town of Bloomington was made all the more memorable by the large number of wonderful friends I made during the years. I want to specially thank all of them for their friendship.

My greatest debt of gratitude goes to my family. I thank my parents for their affection and support and dedicate this thesis to them. Finally, I want to thank my dear wife Shailaja. Her love and many sacrifices made this thesis possible.

Abstract

Prior diagnoses in unification-based type reconstruction systems have either missed information that is relevant, presented irrelevant details, or both.

We use a framework based on the Unification Logics of Le Chenadec to define, derive and simplify proof-based source-tracking for term unification. The objects of source-tracking are proofs in this deduction system, and correspond to path expressions over a unification graph whose labels form a semi-Dyck language of balanced parentheses. Simplification of source-tracking information is implemented as proof normalization in the rewrite system for free groups. Subject-reduction properties guarantee that normalization preserves the semantics of deductions. The presentation of the logic facilitates proof construction by a simple extension to standard unification algorithms.

We apply unification source-tracking to type inference in the Curry-Hindley type system. Programs are represented as systems of syntax equations. Program slices correspond

to weakenings of syntax and type equations. A constraint generation function maps weakenings of type equations to weakenings of syntax equations. Source-tracking information is defined in terms of the inverse of this generating function.

Unification is central to many applications of symbolic computation and artificial intelligence, including computer algebra, automated theorem proving, expert systems, and programming language type systems. Source-tracking is a debugging technique based on tracing the execution of a program to identify those subparts that contribute to the result of the execution.

Contents

List of Figures

1

Introduction

1.1 Diagnosis of computer errors

When programs encounter an error, they frequently fail to answer the following questions in a satisfactory manner:

1. Reporting the symptom of the error: *What is the error?*
2. Diagnosis, or source-tracking of the error: *What caused the error?*
3. Correction of the error: *How can the error be removed or fixed?*

Even if the reporting of the symptom is adequate, programs are poor in communicating the diagnosis of the error. On the occasions that a diagnosis is offered, it is either too terse, as seen in operating system error messages (the infamous "bus error" in Unix), or overwhelming in its prolixity, as seen in some parser error messages of compilers. As a result, a program's error messages can sometimes be confusing and hard to understand. Finally, repair of the program is left to the writer of the program. Without adequate diagnosis of the error, it is hard to repair the program, especially in the absence of proper debugging tools.

1.2 Term unification and type inference

This thesis presents a framework for the formulation, derivation and simplification of diagnostic information for errors occurring in implementation of two well-known problems in programming systems: term unification and type inference.

The problem of *term unification*, first studied by Herbrand [45] in the 1930's, consists of automatically solving equations over *terms*. Terms are built from variables and *constructor* symbols. For example, if x, y, u and v denote variables, and f a constructor symbol, then *unifying* the terms $\mathsf{f}(x, y)$ and $\mathsf{f}(u, v)$ means solving the term equation $\mathsf{f}(x, y) = \mathsf{f}(u, v)$. A system of equations is *unifiable* if it has a solution. The example term equation is unifiable because it has the solution $x \mapsto u,\ y \mapsto v$. The rules of unification used for solving term equations consist of the rules of equality, substitution, and equating subterms. Term unification has applications in many areas, including automated deduction, artificial intelligence, and programming languages.

In term unification, errors correspond to the situation when a set of equations is not solvable. For example, if x is a variable and a and b are distinct constant terms, the system of equations $\{x = \mathsf{a}, x = \mathsf{b}\}$ is unsolvable.

An important application of term unification consists of automatically assigning certain syntactic entities, called types, to programs. A *type*, roughly speaking, identifies a set of values. The execution of a program may result in a state where a primitive operation is applied to values that are outside its domain, or type. Such situations, if unchecked, can lead to unpredictable program behavior. *Safe languages* prevent the occurrence of such unpredictable behavior by ensuring when a primitive operation is supplied argument values that are outside its domain, program execution halts, signaling a *type error*. Other languages may choose to go ahead an carry out the application of the primitive operation, ignoring the consequences. C and C++ are examples of such *unsafe languages*. For example, we expect evaluation of the simple program $5 + \mathtt{true}$ in a safe language to signal an error, because one of the arguments of the addition operator $+$ is a boolean and not a numerical value. A program is *type safe* if execution of the program never results in the application of a primitive operation on invalid arguments. In other words, a type safe program comes with a guarantee that certain type invariants are maintained at runtime.

In *latently typed* type safe languages like LISP and its derivatives, type safety is achieved by performing runtime checks at each application of a primitive operation, verifying that the operation is receiving arguments of the expected type.

In *statically typed* type safe languages like Java and ML [77], type safety is ensured by analyzing the program at compile-time and assigning a type to the program and all its subexpressions. A type is represented by a symbolic term. We use the word "type" to denote a set of values and the term used to symbolically represent the set of values. A program is *well-typed* if it is possible to assign it a particular type that is consistent with the types of all the program's subexpressions. A well-typed program carries with it a guarantee of type safety. If the program cannot be assigned a type, it is said to be *ill-typed*. For the sake of decidability, static typing necessarily flags some safe programs as ill-typed. However, because well-typed programs are guaranteed to be safe, some runtime checks that verify the type of operands may be omitted, thereby making well-typed programs more efficient. The type computed by static typing is also an important component of a program's documentation.

In some statically typed languages, variable declarations also include an explicit declaration of their types, called a *type declaration*. Such languages are classified as being *explicitly typed*. Pascal and Java are examples of explicitly typed languages. The process of verifying that the usage of program variables is consistent with their type declarations is called *compile-time type checking*, or *type checking*, for short.

In *implicitly typed* programming languages like ML, however, variable declarations may be absent: variable types are deduced by analyzing the context in which variables are used. The process of inferring types of programs in the absence of explicit type declarations is known as *type inference*[1]. Whether a program can be assigned a type is determined by a

[1] Type inference is also referred to as *type assignment* and *type reconstruction*.

set of logical rules, collectively called a *type system*. For simple type systems, the process of inferring a program's type may be divided into three phases:

1. Assign type variables to denote the types of each program location.

2. Generate a system of type equations from the program using the rules of the type system.

3. Solve the system of type equations using term unification.

In the case of type inference, errors correspond to situations in which a program cannot be assigned a type conforming to the set of rules specified by the type system. The symptom of the type error is a unification failure. However, the reported symptom of the error is often inadequate for systematically deducing a diagnosis of the error.

Information generated by other static analyses, such as data-flow analysis, binding-time analysis, and register allocation is typically used by compilers for object code generation. On the other hand, the information generated by type inference is of direct interest to the programmer.

Languages like ML that employ type inference may have complex type systems. The complexity is due to the presence of two important features: higher-order functions and polymorphism. Higher-order functions are functions that take other functions as arguments or yield functions as results. Higher-order functions can therefore have complex types.

Polymorphism means that a function can possess a general type that can be instantiated to specific types depending on its usage. Polymorphism increases the complexity of type system rules and the syntax of types, making error diagnosis harder. Poor error diagnosis compromises the user-friendliness and popularity of these otherwise powerful and elegant languages.

1.3 Inadequate type error diagnosis in ML: two simple programs

To illustrate the problematic nature of type error messages, let us consider two simple experiments. Experiment 1 (Figure 1.1) shows a simple program written in the 1998 New Jersey dialect of Standard ML and the compiler's messages reporting a type error. The body of the function consists of a simple `if` expression containing three references to a bound variable `x` (one each on lines 2, 3 and 4), and a predefined constant `inc` assumed to be of type `int` $\rightarrow$ `int` (line 3). The inexperienced ML programmer might find the error message confusing. The confusion is exacerbated by the compiler converting the `if` expression into a `case` expression along the way.[2] The type error in this case has been triggered by the unsuccessful attempt to unify `int` with `bool`. According to the error message, the type

[2]ML treats `if` as a macro. Tracking the source of errors in the presence of macros and rewriting systems is the subject of a related, but separate study (see, for example, the work on origin tracking [10, 104]).

of the variable `x` in the program fragment `case x` is an `int` (integer) does not match the expected type `bool` (boolean). In reporting the type error, the compiler has localized the source of the error to the entire `case` expression corresponding to lines 2, 3, and 4 of the source program, which suggests that all three references to the variable `x` are needed to construct the type error. By considering the type rules for `if`, and the type `int` $\rightarrow$ `int` of the constant inc, however, it is possible to explain the type mismatch using the occurrences of `x` only on lines 2 and 3 of the original program: Line 2 requires the type of `x` to be `bool` because it is the test part of the `if` expression, whereas line 3 requires that the type of `x` be `int` because it is the argument to the constant inc of type `int` $\rightarrow$ `int`. There is another way of explaining the type mismatch, this time using the occurrences of `x` on lines 2 and 4: Again, line 2 requires `x` to be of type `bool`. The type of `x` on line 4 must match the type of the expression `inc(x)`, because lines 3 and 4 are the `then` and `else` clauses of an `if` expression. But the type of `inc(x)` is the return type of the function `inc`, which is `int`.

Experiment 1 shows that implementations of Standard ML could report more information than necessary when signaling a type error. While manageable for small programs, debugging type errors for moderately large programs becomes difficult. (Consider an ill-typed program containing a `case` expression with ten, rather than two variants, for example.)

Experiment 2 (Figure 1.2) shows that ML type error reporting suffers from the opposite problem as well: sometimes the error message contains too little information to reconstruct

```
Standard ML of New Jersey Version 110.0.3, January 30, 1998
fn x =>
    if x
      then inc(x)                // inc: int → int
    else x;

Error: case object and rules don't agree [tycon mismatch]
rule domain: bool, object: int in expression:
   case x of
     true => inc x
     | false => x
```

Figure 1.1: Experiment 1: Example program showing redundant type error information generated by the Standard ML of New Jersey compiler. Text in boldface typewriter font refers to user input. Text in regular typewriter font refers to the output of the compiler. Text following `//` denotes a comment.

```
Standard ML of New Jersey Version 110.0.3, January 30, 1998
fn (f,g,x) =>
    if f(g(x)) then
      let val v = f(x)
        in g(v) end
    else
      let val z = g(x)
        in inc(z) end;


Error:  case object and rules don't agree [tycon mismatch]
rule domain:  bool,  object:  int  in expression:
   case f(g(x)) of
     true => let val <pat> = <exp> in g(v)
     | false => let val <pat> = <exp> in inc(z)
```

Figure 1.2: Experiment 2: Example program showing insufficient type error information generated by the Standard ML of New Jersey compiler. Text enclosed in angle brackets of the error message denotes program parts deemed unnecessary for the reconstruction of the error by the compiler.

the type error. The program in Experiment 2 is ill-typed and its untypability may be explained by the following sequence of inferences. From line 2, we can infer that the return type of `f` is `bool`. By lines 3 and 4, the type of `v`, which is an argument of `g`, is also `bool`. Hence, the argument type of `g` is `bool`. Now, from lines 6 and 7, we can infer that the return type of `g` is `int`. From this, and the application `f(g(x))` on line 1, it is clear that the argument type of `f` is `int`. Thus `f` has type `int` → `bool` and `g` has type `bool` → `int`. The variable `x` being passed to `f` (line 3) is therefore of type `int`. This clashes with the type `bool` required of `x` when it is passed to `g` in line 6.

The error message correctly identifies a "clash" between `int` and `bool`. Additionally, the compiler suggests that the two `let` binding clauses (lines 3 and 6) are irrelevant to the reconstruction of the type error. However, without using the type constraints gathered from these two clauses, it is impossible to reconstruct the type error.

Haskell[50] is another popular functional programming language with a powerful type inference system. When run under Haskell, the above experiments produce even more inscrutable error messages, mainly due to the additional complexity of the type inference introduced by Haskell's overloading mechanism.

These experiments suggest a basic problem with the practical usability of type-inference-based functional programming languages. Merely dressing up error messages produced by existing compilers with a graphical user interface will only be of limited value. Instead, useful error reporting requires integrating the generation of diagnostic information with the type inference process. A rigorous foundation for diagnosis of type errors should construct formal proofs of untypability, because inferring the type or untypability of a program is, in a technical sense, equivalent to proving a theorem. Therefore, error diagnoses should correspond to proofs of untypability in an appropriate logic. Since type inference is based on term-unification, it should be possible to reconstruct a proof of the type error starting from the symptom of the type error, which is unification failure. These ideas of a proof-based framework are illustrated in the next section where we analyze the program of Experiment 1 in more detail.

1.3.1 Diagnostic analysis of the program in Experiment 1

We show the type inference process involved in trying to infer the type of Experiment 1's example program, which we call e:

$$\lambda\, x.\, \mathsf{if}\ x\ (@\ \mathsf{inc}^{\mathtt{int}\longrightarrow\mathtt{int}}\ x)\ x$$

We use an abstract syntax based on λ-calculus, replacing ML's `fn` keyword with λ, dispensing with the keywords `then` and `else`, and denoting function application with $@$. $\mathsf{inc}^{\mathtt{int}\longrightarrow\mathtt{int}}$ denotes the constant inc annotated with its type. The parse tree for e is given in Figure 1.3. Each vertex in the parse tree corresponds to a location in e and is identified with an integer. The parse tree of e may alternatively be specified using a set of *syntax*

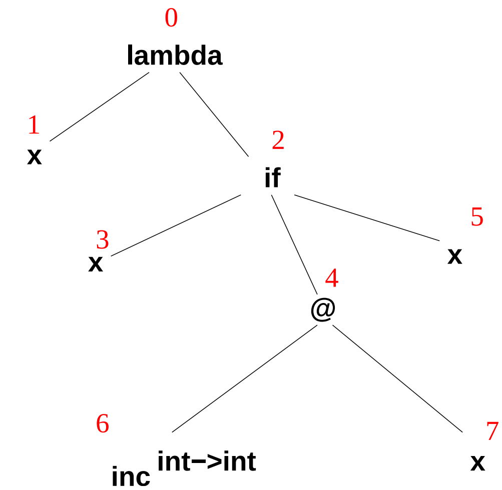


Figure 1.3: Parse tree of example program e of Experiment 1

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
e_0 & = & \lambda(e_1, e_2) & e_4 & = & @(e_6, e_7) \\
e_1 & = & \mathsf{formal} & e_5 & = & \lambda\mathsf{var}(e_1) \\
e_2 & = & \mathsf{if}(e_3, e_4, e_5) & e_6 & = & \mathsf{const}(\mathsf{inc}, \mathtt{int}{\rightarrow}\mathtt{int}) \\
e_3 & = & \lambda\mathsf{var}(e_1) & e_7 & = & \lambda\mathsf{var}(e_1)
\end{array}
$$

Figure 1.4: Syntax equations describing the parse tree of the example program

equations as shown in Figure 1.4. The expression at location i is denoted e_i. Most syntax equations relate a location with other locations, which correspond to subterms or variable bindings. Syntax equations for constants relate the name of a location with the name of a constant and its type. The syntax equations obviate the need for names of bound variables.

The type system is informally described by *rules* as shown in Figure 1.5. These rules

$$
\begin{array}{lcl}
e_i = \lambda(e_j, e_k) & \Longrightarrow & t_i \stackrel{?}{=} t_j \rightarrow t_k \\
e_i = @(e_j, e_k) & \Longrightarrow & t_j \stackrel{?}{=} t_k \rightarrow t_i \\
\Box = \mathsf{if}(e_j, \Box, \Box) & \Longrightarrow & t_j \stackrel{?}{=} \mathtt{bool} \\
e_i = \mathsf{if}(\Box, e_k, \Box) & \Longrightarrow & t_i = t_k \\
\Box = \mathsf{if}(\Box, e_k, e_l) & \Longrightarrow & t_k = t_l \\
e_i = \lambda\mathsf{var}(e_j) & \Longrightarrow & t_i \stackrel{?}{=} t_j \\
e_i = \mathsf{const}(c, \tau) & \Longrightarrow & t_i \stackrel{?}{=} \tau
\end{array}
$$

Figure 1.5: Type Rules used for computing the type of the example program

relate syntax equations with *type equations*, which are equations relating types of locations. In each type rule, a syntax equation implies one or more type equations. t_i denotes a variable representing the type of the subexpression e_i. When generating type equations, we write $\stackrel{?}{=}$ rather than $=$ to suggest that type equations are entities that need to be solved, rather than identities that are true. We say that the type equations are *generated* by the syntax equations. Each type equation is associated with the syntax equation that generated it. The set of type equations generated in the type inference of e and the syntax equations generating each type equation is shown in Figure 1.6. Labels are used to refer to type equations. A type equation $\tau \stackrel{?}{=} \tau'$ labeled l is written $l : \tau \stackrel{?}{=} \tau'$.

The special symbol $\Box$ denotes "irrelevant" information. For example, the syntax equation $\Box = \mathsf{if}(e_3, \Box, \Box)$ generates the type equation $t_3 \stackrel{?}{=} \mathtt{bool}$. The syntax equation in which e_3 is the test expression of an if expression will always generate the type equation $\mathtt{t}_3 \stackrel{?}{=} \mathtt{bool}$. The names of the locations corresponding to the if expression, and the then

Syntax equation	**Type equation**	**Label**
$e_0 = \lambda(e_1, e_2)$	$t_0 \stackrel{?}{=} t_1 \rightarrow t_2$	a
$e_2 = \mathsf{if}(\square, e_4, \square)$	$t_2 \stackrel{?}{=} t_4$	b
$\square = \mathsf{if}(e_3, \square, \square)$	$t_3 \stackrel{?}{=} \mathtt{bool}$	c
$\square = \mathsf{if}(\square, e_4, e_5)$	$t_4 \stackrel{?}{=} t_5$	d
$e_3 = \lambda\mathsf{var}(e_1)$	$t_3 \stackrel{?}{=} t_1$	e
$e_4 = @(e_6, e_7)$	$t_6 \stackrel{?}{=} t_7 \rightarrow t_4$	f
$e_5 = \lambda\mathsf{var}(e_1)$	$t_5 \stackrel{?}{=} t_1$	g
$e_6 = \mathsf{const}(\mathtt{int} \rightarrow \mathtt{int})$	$t_6 \stackrel{?}{=} \mathtt{int} \rightarrow \mathtt{int}$	h
$e_7 = \lambda\mathsf{var}(e_1)$	$t_7 \stackrel{?}{=} t_1$	i

Figure 1.6: Syntactic constraints and type constraints generated from them for the example program.

and else clauses are irrelevant to the generation of the equation $t_3 \stackrel{?}{=} \mathtt{bool}$. The result of replacing zero or more subterms of a symbolic term equation with $\square$ is called a *weakening* of the term equation.

The set of type constraints generated form a system of term equations and are solved using unification. The process of unification involves deriving additional equations from an initial set of equations. These equations are derived using the *unification closure* rules specified in Figure 1.7. The rules are specified as logical implications. Each rule consists of one or more antecedents and one or more consequents. The antecedents are separated from the consequent by a horizontal line. For each rule, if the antecedents are true, then the consequents are assumed true. In the rules, τ and its subscripted and primed variants denote terms constructed using variables, the binary constructor symbol $\rightarrow$, and nullary

$$\text{REF} \qquad \frac{}{\tau \stackrel{?}{=} \tau}$$

$$\text{SYM} \qquad \frac{\tau \stackrel{?}{=} \tau'}{\tau' \stackrel{?}{=} \tau}$$

$$\text{TRANS} \qquad \frac{\tau \stackrel{?}{=} \tau' \qquad \tau' \stackrel{?}{=} \tau''}{\tau \stackrel{?}{=} \tau''}$$

$$\text{DN} \qquad \frac{\tau_1 {\rightarrow} \tau_2 \stackrel{?}{=} \tau_1' {\rightarrow} \tau_2'}{\tau_i \stackrel{?}{=} \tau_i'} \qquad i = 1, 2$$

Figure 1.7: Unification closure rules for type terms formed using a single binary functor $\rightarrow$.

constructors `int` and `bool`. The rules REF, SYM and TRANS are respectively the reflexivity, symmetry and transitivity rules of equality. The DN rule is downward-closure, asserting the injectivity of term constructors: if two terms are equal, then their corresponding subterms are equal.

Using the TRANS and SYM rules, from equations h and f we can infer the type equation $t_7 {\rightarrow} t_4 \stackrel{?}{=} \texttt{int}{\rightarrow}\texttt{int}$. Applying the DN rule to this equation yields the pair of equations $j : \texttt{int} \stackrel{?}{=} t_7$ and $k : \texttt{int} \stackrel{?}{=} t_4$. The equations $j : \texttt{int} \stackrel{?}{=} t_7$, $i : t_7 \stackrel{?}{=} t_1$, $e : t_3 \stackrel{?}{=} t_1$ and $c : t_3 \stackrel{?}{=} \texttt{bool}$ form a chain of equality:

$$\texttt{int} \stackrel{?}{=} t_7 \quad t_7 \stackrel{?}{=} t_1 \quad t_1 \stackrel{?}{=} t_3 \quad t_3 \stackrel{?}{=} \texttt{bool}$$

implying $\texttt{int} \stackrel{?}{=} \texttt{bool}$, which is clearly unsolvable. Also, the equations $k : \texttt{int} \stackrel{?}{=} t_4$, $d : t_4 \stackrel{?}{=} t_5$, $g : t_5 \stackrel{?}{=} t_1$, $e : t_3 \stackrel{?}{=} t_1$ and $c : t_3 \stackrel{?}{=} \texttt{bool}$ form the chain

$$\texttt{int} \stackrel{?}{=} t_4 \quad t_4 \stackrel{?}{=} t_5 \quad t_5 \stackrel{?}{=} t_1 \quad t_1 \stackrel{?}{=} t_3 \quad t_3 \stackrel{?}{=} \texttt{bool}$$

again yielding the unsolvable equation $\texttt{int} \stackrel{?}{=} \texttt{bool}$.

The type constraints generated by the example program are unsolvable because together they imply the unsolvable type constraint $\texttt{int} \stackrel{?}{=} \texttt{bool}$. Therefore, the example program is untypable, or ill-typed, because the type constraints it engenders are unsolvable. We say that the unsolvable type constraint $\texttt{int} \stackrel{?}{=} \texttt{bool}$ is a *symptom* of the untypability of e.

A graphical representation of the type constraints is very helpful in understanding unification. The type constraint system is represented as a *unification graph*. The unification graph representing the type constraints of Figure 1.6 is shown in Figure 1.8. Each type variable and type constant is represented as a vertex. Thick edges represent relations between type terms and their subterms. Thin edges represent equational constraints. Each equational edge is oriented in an arbitrary, but fixed direction. Solid edges correspond to original constraints. Dashed edges correspond to constraints derived using the rules of unification.

The process of unification involves deriving special connectivity relations between vertices in the graph. In the example, a proof for the equation $\texttt{int} \stackrel{?}{=} \texttt{bool}$ may be viewed as a

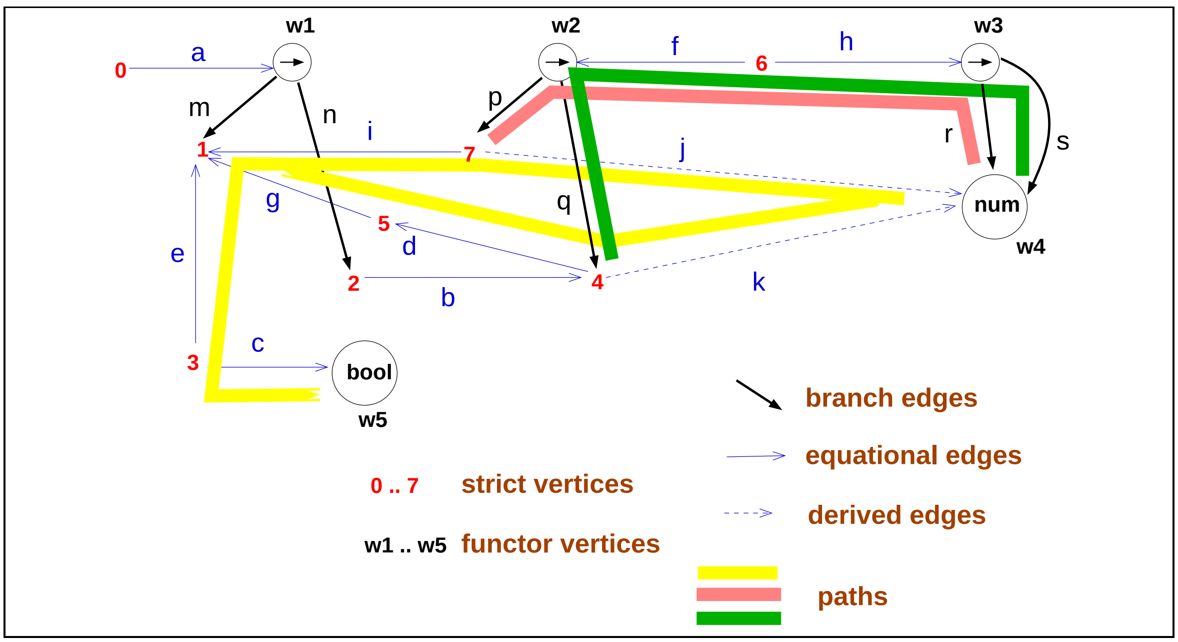


Figure 1.8: Unification graph of example program. Each type variable t_i is represented by the vertex i. Vertices with circles represent type constructors, with the label inside the circle denoting the constructor symbol. Bold edges represent branch edges. Thin edges represent equational edges. Dashed edges represent derived equations.

path connecting the nodes `int` and `bool` in the unification graph, with the assumption that directed and undirected edges may be traversed in either direction. Traversal of an edge y opposite to its orientation is denoted y^{-1}. Thus, the paths $j^{-1}ie^{-1}c$ and $k^{-1}dge^{-1}c$ between the nodes `int` and `bool` in Figure 1.8 are both witnesses to the unsolvability of the system of type equations, and thus the untypability of e. The edges j and k owe their existence to the downward-closure rule and the connectivity of the $\rightarrow$ nodes via the edges f and h. Thus, the edge j is derived from the path $p^{-1}f^{-1}hr$ consisting of edges specified by the original set of type constraints. Similarly, the edge k is derived from the path $q^{-1}f^{-1}hs$.

The problem of diagnosing untypability can therefore be described as identifying a set of original type constraints, or *diagnosis*, and through them a subsystem of syntactic information from the program, or *source* that is sufficient to derive a particular type constraint. In addition, we are interested in obtaining diagnoses of errors that are *minimal* with respect to the weakening ordering. A diagnosis is minimal if it logically implies the symptom, but no proper weakening of that diagnosis implies that symptom.

Replacing edge j with $p^{-1}f^{-1}hr$ in $j^{-1}ie^{-1}c$ yields $(p^{-1}f^{-1}hr)^{-1}ie^{-1}c$, which simplifies to the path $r^{-1}h^{-1}fpie^{-1}c$ consisting of edges in the original set of type constraints. Similarly, replacing k with $q^{-1}f^{-1}hs$ in $k^{-1}dge^{-1}c$ yields $s^{-1}h^{-1}fqdge^{-1}c$. These two paths connect the vertex `int` to `bool`, and constitute two different diagnoses of unsolvability of the original set of type constraints. Furthermore, no other path consisting of a proper subset of the edges of these paths connects `int` to `bool`.

The set of edges $\{r, h\}$ corresponds to the weakening $t_6 \stackrel{?}{=} \texttt{int} \rightarrow \Box$ of the constraint $h : t_6 \stackrel{?}{=} \texttt{int} \rightarrow \texttt{int}$. The $\Box$ represents a “hole” indicating that the second occurrence of `int` is irrelevant. Each occurrence of a hole is interpreted as an occurrence of a “new” variable not occurring anywhere else in the set of equations. Similarly, $\{f, p\}$ corresponds to the weakening $t_6 \stackrel{?}{=} t_7 \rightarrow \Box$ obtained by replacing the subterm t_4 with $\Box$ in the type equation $f : t_6 \stackrel{?}{=} t_7 \rightarrow t_4$. The path $r^{-1}h^{-1}fpie^{-1}c$ consisting of the two path segments $r^{-1}h$ and fp, along with the edges i, e and c, therefore, correspond to the following set E_1 of minimally non-unifiable type equations:

$$
\begin{array}{lllllll}
t_6 & \stackrel{?}{=} & \texttt{int} \rightarrow \Box & \qquad & t_6 & \stackrel{?}{=} & t_7 \rightarrow \Box \\
t_7 & \stackrel{?}{=} & t_1 & & t_3 & \stackrel{?}{=} & t_1 \\
t_3 & \stackrel{?}{=} & \texttt{bool} & & & &
\end{array}
$$

For each t_i that is replaced with a $\Box$ in the type equation, we replace the corresponding occurrence of e_i in the syntax equation with a $\Box$. This yields the following set S_1 of syntax equations that generate the type equations in E_1:

$$
\begin{array}{lcllcll}
e_6 & = & \mathsf{const}(\square, \mathtt{int}{\rightarrow}\square) & \Longrightarrow & t_6 & \stackrel{?}{=} & \mathtt{int} \rightarrow \square \\
\square & = & @(e_6, e_7) & \Longrightarrow & t_6 & \stackrel{?}{=} & t_7 \rightarrow \square \\
e_7 & = & \lambda\mathsf{var}(e_1) & \Longrightarrow & t_7 & \stackrel{?}{=} & t_1 \\
e_3 & = & \lambda\mathsf{var}(e_1) & \Longrightarrow & t_3 & \stackrel{?}{=} & t_1 \\
\square & = & \mathsf{if}(e_3, \square, \square) & \Longrightarrow & t_3 & \stackrel{?}{=} & \mathtt{bool}
\end{array}
$$

Similarly, the set of minimally non-unifiable type equations E_2 derived from the path $s^{-1}h^{-1}fqdge^{-1}c$ and the set of syntax equations, S_2, generating E_2 are:

$$
\begin{array}{lcllcll}
e_6 & = & \mathsf{const}(\square, \square{\rightarrow}\mathtt{int}) & \Longrightarrow & t_6 & \stackrel{?}{=} & \square \rightarrow \mathtt{int} \\
t_4 & = & @(e_6, \square) & \Longrightarrow & t_6 & \stackrel{?}{=} & \square \rightarrow t_4 \\
\square & = & \mathsf{if}(\square, e_4, e_5) & \Longrightarrow & t_4 & \stackrel{?}{=} & t_5 \\
e_5 & = & \lambda\mathsf{var}(e_1) & \Longrightarrow & t_5 & \stackrel{?}{=} & t_1 \\
e_3 & = & \lambda\mathsf{var}(e_1) & \Longrightarrow & t_3 & \stackrel{?}{=} & t_1 \\
\square & = & \mathsf{if}(e_3, \square, \square) & \Longrightarrow & t_3 & \stackrel{?}{=} & \mathtt{bool}
\end{array}
$$

The syntax equation sets S_1 and S_2 are represented graphically in Figures 1.9(a) and (b) respectively. The graphical representation consists of fragments of the parse tree obtained by selectively erasing location information, or entire subtrees, from the parse tree.

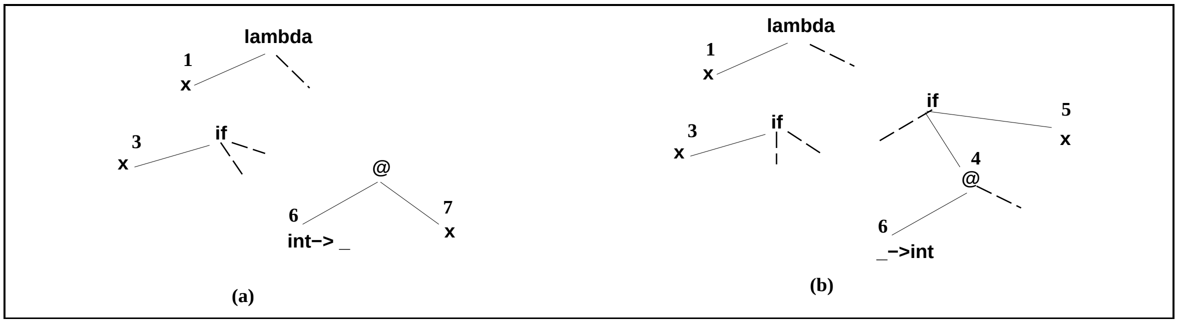


Figure 1.9: Graphical representation of syntax equations S_1 and S_2

The fragments are to be treated as patterns that match various subparts of the program expression's parse tree. For example, there are two different if fragments in S_2 that are both matched by the same if subexpression of e. Besides e, any program whose syntax matches the templates of S_1 or S_2 is also guaranteed to be ill-typed for the same reason as e.

The syntax equation sets S_1 and S_2 are minimal in the sense that any further weakening of them will result in a set of type equations not constrained enough to generate a type error.

1.4 Contributions of the thesis

This thesis is based on the premise that a rigorous diagnosis of type inference is possible only after building a framework for diagnosis of term unification which is at the core of type inference. The thesis identifies the three elements of a formal framework for term unification: a graph-theoretic model for the unification diagnosis problem; a logical framework to derive diagnostic information, and a practical algorithm for deriving source-tracking information in the event of unification failure.

Our framework for source-tracking term unification rests on fundamental properties relating paths in labeled directed graphs to paths in their quotient graphs. By suitably labeling the edges of the unification graph, we obtain a characterization of witnesses to membership in the unification closure as paths whose labels form an important class of context-free languages, the semi-Dyck sets. There are advantages of thinking of unification closure purely in terms of connectivity via these special paths, independent of the operational details of any unification algorithm. For example, computation of optimal proofs reduces to searching for optimal (shortest) paths. We define a logic $P^U(G)$ of connectivity over a labeled directed graph G. We present the logic as a simple type system whose well-typed expressions correspond to paths labeled with semi-Dyck set prefixes. We next define a practical algorithm for deriving source-tracking information for unification. When the algorithm terminates with failure, it also returns a proof of why the terms did not unify. The algorithm is obtained by redefining standard unification algorithms to include an extra "proof" parameter. Next, using a rewrite system for groups, we simplify these proofs in order to identify and remove details not relevant to the simulation of the error. The logic $P^U(G)$ has similar soundness and completeness properties as the unification logic LE_0 of Le Chenadec [64], but is adapted to work on graph vertices so as to make the sharing properties of terms explicit. A comparison of the two logics is detailed in Chapter 2.

We next consider the problem of error reconstruction for the classical Curry-Hindley type system of the simply-typed lambda calculus. Our formalization is based on viewing the abstract syntax tree of a program as a set of syntax equations. Remarkably, this simple representation of program syntax is sufficient to define a precise constraint generating function from syntactic equations to type constraints for the Curry-Hindley type system.

Prototype versions of the unification source-tracking algorithm and diagnostic type inference for the Curry-Hindley type system have been implemented in Scheme [20].

Our approach for error diagnosis in unification and type inference is both more rigorous and simpler compared to many existing approaches reviewed in Chapter 2.

1.5 Organization of the thesis

The rest of the thesis is organized as follows: Chapter 2 discusses several previous attempts at addressing the diagnosis problem in various related contexts including type inference, program debugging, logic programming, and Artificial Intelligence. Chapter 3 reviews the main definitions and results of term unification of rational terms. Unification is treated as construction of closure relations on the vertices of a unification graph. Chapter 4 presents a logical deduction system for computing proofs of non-unifiability. These logical deductions are shown to correspond to typed path expressions in an algebra of untyped edge-expressions over the unification graph. Normalization in the untyped algebra is shown to preserve the types of these path expressions and thus forms the basis for removing redundancies from proofs. The practical outcome of this analysis is an extension of the unification algorithm that supports automatic proof construction. Chapter 5 presents the problem of diagnosing type errors in the Curry-Hindley type system. A framework based on syntax and type equations is formally defined. Chapter 6 concludes the thesis with a discussion of future extensions of the work on source-tracking to other other variants of the unification problem, to more sophisticated type regimes, such as polymorphism in ML, and finally to other problems such as debugging in logic programming languages.

2

Related Research

In this chapter, we survey research work specifically devoted to the type error and unification failure diagnosis problem, and also work in other related areas, like program slicing, logic programming, unification logics, and diagnosis in artificial intelligence.

2.1 Type Error Diagnosis: Previous attempts

This section catalogues previous attempts at solving the type error diagnosis problem and their varying degrees of success.

2.1.1 Wand's algorithm

Wand [108] was among the earliest to address the problem of reporting type errors. He provided an implementation of his error diagnosis algorithm within the framework of his

Semantic Prototyping System [107]. Wand's algorithm was also used as an error reporting front-end for the Infer system [43]. Wand's algorithm is a modification of the structure sharing version of the unification algorithm. It is based on the observation that Milner's polymorphic type inference algorithm [75] assigns a type variable to every lexically-bound variable and every application. A unification inference $t_v \doteq \tau$ unifying a type variable t_v with a type expression τ is called a *type variable binding*. A *reason* is a program site, typically a function application. Reasons are accumulated when traversing the binding pointers maintained by the unification algorithm. When a new binding from a type variable t_v to another type expression t_e is made, the reasons associated with this binding are defined to be the set of reasons accumulated while arriving at the vertex representing t_e in the unification graph.

With each pair of vertices u and u' being unified, two sets of reasons r and r' are maintained. Intuitively, r is the set of reasons why the unifier arrived at vertex u, and r' is the set of reasons why the unifier arrived at u'. If unification fails when trying to unify u with u', Wand's algorithm simply returns the two sets of reasons at the point of failure. Wand counts the algorithm as "successful in identifying the source of an error if the place of the [error] was one of the error sites produced by the algorithm." Wand also speculates the existence of a 'completeness criterion' for judging the efficacy of his and other algorithms that locate type errors. A plausible test for success is to verify if the reasons returned are sufficient to reconstruct the error that is reported. Unfortunately, Wand's algorithm fails this criterion even for some simple examples. Consider the following sequence of equations: $1 : x \stackrel{?}{=} y$, $2 : x \stackrel{?}{=} z$. (The number preceding the colon indicates a source program site.) Wand's algorithm first performs the binding $x \mapsto y$, associating with it the reason set $\{1\}$. The set of reasons accumulated following the binding pointers from x to y is $\{1\}$, while the set of reasons accumulated following the binding pointers from z to z is $\{\}$, because z is as yet unbound. When the the algorithm is ready to make the binding $y \mapsto z$, it accumulates the reasons $\{1\}$ to reach y from x, and the set $\{\}$ as the reasons to reach z. But, it unfortunately ignores the entire set of reasons accumulated for y when actually making the binding. Instead, it only uses the reasons for reaching z to compute the reason set for the binding $y \mapsto z$. (See the last four lines of the algorithm in Figure 1 of [108].) Thus, y is bound to z, and the set of reasons attributed for this binding is $\{\}$. In other words, the algorithm incorrectly concludes that neither of the premises $x = y$ or $x = z$ are needed to infer the equation $y = z$. This bug can be easily fixed. There is one other bug that causes more reasons to be lost. For example, given the sequence of unifications $1 : x \stackrel{?}{=} y$, $2 : x \stackrel{?}{=} \mathsf{int}$, $3 : x \stackrel{?}{=} \mathsf{bool}$, the error $\mathsf{int} \stackrel{?}{=} \mathsf{bool}$ will return the pair of reason sets $\{1, 2\}$, obtained by traversing binding pointers from x to int, and $\{\}$, obtained from traversing bool to bool. In order to simulate the error, we need the site 3 as well, which is the site of the equation whose unification attempt resulted in the error. But this site is not part of either reason list.

Another problem with the algorithm is that it generates redundant information even for simple inferences. In the example above, the set of reasons $\{1, 2, 3\}$ together have enough information to derive the error. But clearly, equation 1 is irrelevant to the derivation of the error. Eliminating these redundant inferences from reason lists in a systematic way can be done only by abandoning the algorithm's set-based approach in favor of the path-based approach of our algorithm. Furthermore, once all the relevant program sites are pointed out as reasons, mentally reconstructing the chain of inferences from these sites can be difficult. The approach we suggest is proof-based and automatically constructs a complete deduction in a formal logic.

2.1.2 Attribute Grammars and flow information

Johnson and Walz [54, 106] formulate a theory of error correction and detection in unification-based systems and apply it to the problem of unification-based type inference. They view the process of type inference as attribute flow in the parse tree of the program, reasoning that "type information is propagated up the expression tree[1]." The other aspect of information flow is the "propagation of completed constraint information about type variables (including information about type errors) down the tree." In the presence of type errors, the "error information" field of each type variable collects all the conflicting types of that variable along with a numerical "strength" with which the type variable is asserted to have that type.

[1] page 47, [54]

Type inference in the Johnson and Walz approach is based on the idea of "error-tolerant" unification, which involves solving a multi-set of type constraints by considering disjunction rather than conjunction of constraints. For example, error-tolerant unification of the constraint multi-set $\{t \stackrel{?}{=} \texttt{bool}, t \stackrel{?}{=} \texttt{bool}, t \stackrel{?}{=} \texttt{int}\}$ yields $t = \texttt{int}[1] \mid \texttt{bool}[2]$. The solution suggests that t has the 'options' `int` and `bool` with indicated strengths. The Johnson and Walz scheme rests on a complicated algorithm that derives implied type constraints obtained from the original type constraints by applying the rules of substitution and transitivity. Unfortunately, their work presents no soundness or completeness criterion to judge the adequacy of their solution. Error-tolerant unification algorithm may be more useful in type inference under "soft typing" regimes [16, 37] but the types returned by this approach, specially in the presence of nested disjunctions, tend to be large and hard to understand.

The problem with the attribute grammar approach used by Walz and others [8, 35, 69] is that these approaches confuse the generation of type constraints, which is directed by the syntax of the program, with the solution of these constraints, which is instead directed by the geometry of the constraints themselves, and not the parse-tree of the program. Type information flows along specialized paths of the unification graph and not the parse tree.

2.1.3 Partial Type Inference

The goal of partial type inference is to infer as much type information as possible for the 'typable' parts of the program and annotate the rest of the fragments of the program as ‘untypable’. Gomard [40] uses this idea to develop a type inference system essentially based on a modification of Milner’s algorithm W. An ill-typed program is ‘well-annotated’ according to the rules of an extended type system that is permissive on the annotated (read ill-typed) parts of the program and “imposes a ‘normal’ type discipline on the un-annotated (type correct) parts.”

Gomard defines a notion of ‘completion’ between programs, where an expression e' is a ‘completion’ of e if e is well-annotated and is obtainable from e by adding annotations to e. Gomard shows elsewhere that the set of expressions forming a completion of a given expression e form a partial order. He speculates that his algorithm produces minimally annotated completions of a program.

From the point of view of isolating and reporting type errors, Gomard’s approach does not address one important question: what is the relation between the subexpressions of an annotated expression that result in the type error? In fact, Gomard’s analysis makes no attempt at using the actual type error (for example, whether the error was a type mismatch of the form “int = bool” or an occurs check) in trying to diagnose the cause of the error. Thus, Gomard’s technique offers at best an approximation of the result that we strive to achieve: the relations between the subexpressions of the program that contribute to a type error.

2.1.4 Tracing Techniques

Maruyama et al. [69] propose a tracing and debugging front-end to the type inference algorithm. The heart of their algorithm is a strategy for selecting a candidate set of 'locations' that correspond to the site of the type error. Their tracing technique for type error analysis, however, is based on a flawed heuristic: only parse tree nodes that are adjacent to the node where unification fails are considered. As demonstrated amply by our work, more distant nodes may contribute to the error. It is necessary to consider adjacency in a type-constraint graph rather than the parse tree. They provide no proof or even statement about how the set of candidate error locations relates to the site of the error. Moreover, the selection of a candidate location is done by the programmer using "his/her knowledge about the program." The work of Maruyama et al. uses a rudimentary slicing technique to filter out irrelevant information from the trace of a type inference.

2.1.5 Explanation and Visualization-based systems

Most of the effort in debugging type error messages in ML has focused on providing explanation-based and graphical front-ends for polymorphic type inference systems. Relatively early efforts include Beaven and Stansifer [8], Duggan and Bent [35], Duggan [34], and Soosaipillai [97]. In addition, the recent work of Yang and others [113, 114, 115, 116, 117, 118] and Chitil [18] focuses on explanation-based and visual front-ends to polymorphic type error debugging.

The main effort in these attempts has been to develop an interface where the results of all the type deductions (type bindings) are maintained carefully in the parse-tree of the program. The interaction is prompted by the programmer using a protocol consisting of three queries: `Typeof`, `Why`, and `How`. The `Typeof` query returns the type of an expression or reports a type error if the expression is ill-typed. Typically, the type mismatch occurs between an 'expected type' and the actually inferred type of a subexpression. At this point, the `Why` query asks the system why it expects that a certain subexpression should have a certain type. This essentially queries the system to display the 'top-level' rule of the type discipline that was invoked while attempting to compute the type of the subexpression in question. The query `How` navigates the type binding link pointers.

The principal drawback with the explanation-based approach is its lack of automation or semantic basis for classifying valid paths. In contrast, the type error reporting framework designed in this thesis provides the ability to *automatically* derive such an explanation in a formal way, by producing a proof of untypability.

Several researchers (Beaven and Stansifer [8], Gomard [40], Johnson [54], Maruyama [69]) argue that the problem of type error detection and correction is especially difficult because of polymorphic typing. Lee and Yi [65] propose a "top-down" version of the standard algorithm of Milner [75]. They argue that their algorithm reports type errors more eagerly than

Milner's algorithm. Bernstein and Stark [9] propose a type system that assigns monomorphic types to free variables in ML expressions. They show how this may form the basis of a practical system for debugging type errors. Yang et al. propose two incremental type inference algorithms [114, 118] for computing the source of type errors in the presence of polymorphism. A comparison of their type inference algorithms with those of Wand, Johnson and Walz, Lee and Yi, and Turner [102] can be found in Yang's thesis [114]. However, Yang's algorithms do not derive formal proofs of untypability. Besides, these algorithms, being built on top of unification, are unable to remove redundant inferences introduced by unification.

While polymorphism significantly complicates the typing discipline, the difficulty of reporting type errors is not primarily due to the demands of polymorphism. The fundamental difficulty in diagnosis of type errors is correctly formulating source-tracking in the unification algorithms at the core of type inference, a premise also underlying the early work in type error reporting [54, 108], and the more recent work of Gandhe et al. [38], and McAdam [71, 72].

2.1.6 Program Slicing

Program slicing as a debugging technique gained prominence due to Weiser [110, 111]. There are several ways in which a program slice has been defined in the literature, (see for example the survey chapter in Tip's thesis [100]) but they all consider a *program slice* to be a subset of the program's text and/or its execution profile (trace), i.e., a projection of the sequence of the runtime state of the program that can simulate a given behavior of the program. Program slicing has been effectively applied to derive dependency information in data, control and other flow relations in program analysis and compilation tools. The output of the diagnostic unification algorithm presented in this thesis may be considered as a program slice of the unification graph.

The work of van Deursen [103, 104], Bertot [10], Tip [100] and others has focused on a more formal approach to program slicing based on term rewrite systems. These works of research address the problem of 'origin tracking' in term rewrite systems. Among the interesting applications that their approach has been applied to consists of reporting error messages in languages supporting type checking, but not type inference [30, 31, 32].

2.2 Logic Programming and unification-based systems

Logic programming and Prolog-based systems use unification as their underlying computational engine. The problem of debugging logic programs consists of techniques to manage and organize information about logic variable bindings obtained as a result of successful unification, as well as undoing of bindings during unification failures. As the survey by Ducassé and Noyé [33] points out, a large number of logic programming environment tools are based on abstract interpretation, algorithmic debugging, and tracing, but there is a surprising absence of application of program slicing techniques.

Both logic programming and type inference reduce to unification of term equations. Failure analysis of unification in terms of identifying minimally unifiable subsets of term equations was carried out by Port [89]. Port's algorithm attempts to construct regular path expressions over the unification graph. Our work shows that the path expressions of relevance are a context-free set rather than a regular set.

Cox's [25] idea of maximally unifiable subsets and minimally non-unifiable subsets has also been employed as the basis for developing search strategies for breadth first resolution of logic programs. Chen et al. [17] propose an algorithm to construct these subsets in conjunction with the unification algorithm. Their algorithm works by explicitly maintaining the unification closure relation as an auxiliary graph. The vertices of this graph are elements of the unification closure relation of the original graph. Each inference is represented by edges from the antecedent(s) to the consequent. Attached to each node is the subset of equation used to derive the membership of a particular element in the unification closure.

Our extension to the unification algorithm keeps track of information that is more precise because it computes finer subsystems of the original set of term equations. Furthermore, the information maintained statically (as vertices) in the auxiliary graph constructed by the algorithm of Chen et al. can be generated dynamically in our extension to the unification algorithm, implying that the space requirements of our algorithm is likely more modest.

Gandhe et al. [38] apply Cox's idea of isolating maximally unifiable sets to the problem of correcting type errors in the Curry Hindley type system. Each maximally unifiable set then corresponds to a set of constraints on the syntax of the original term. Candidate corrections to the original untyped term are derived from maximally unifiable subsets by constructing structures, called "sharing graphs," originally used by Lamping to represent λ-calculus terms for optimal λ-calculus reduction [60]. Their main result is a one-to-one correspondence between constraints developed by Wand's Type Inference algorithm [109] and sharing graphs. McAdam [70] is another proposal for graph-based approaches to this problem. Our work shows that generated type constraints can be put in a one-to-one correspondence with flat syntax equations without requiring additional machinery for representation.

2.3 Unification Logics of Le Chenadec

Le Chenadec [62, 63, 64] introduces an equational logic for unification and notes the potential use of his logics for diagnosing type inference. The objects of his logic are well-founded terms and contexts. On the other hand, the logic P^U, introduced in Chapter 4, is a logic of connectivity of graph vertices. The soundness and completeness properties of P^U with respect to the unification graph are simpler to state and prove than those obtained from the logic LE_0 of Le Chenadec. (Compare, for example, Theorem 4.3.1 and Lemma 4.4.1

in Chapter 4 with Proposition 2.10 in [64].) All assumptions about the sharing structure are explicit in the logic P^U, whereas LE_0 depends on restricting subterm sharing only to strict variables ([64], page 151). The group theoretic framework proposed in this thesis for simplifying path expressions is a special case of the more general homotopy algebra semantics for unification deductions proposed by Le Chenadec. This specialization is all that is needed to reduce non-unifiability to the existence of context-free paths. The path-oriented view of proofs of membership in the unification closure allows us to characterize the problem of computing optimal unification diagnosis as a path minimization problem. The specialization to free groups also results in a simple algorithm for removing some, but not all, redundancies in non-unifiability proofs. A useful exercise would be to compare the quality of the proofs thus obtained with the variety of confluent rewrite systems introduced by Le Chenadec that operating directly on deductions. It is unlikely that his rewrite systems would yield shortest proofs or even minimal proofs. On the other hand, our framework affords the choice of computing non-optimal paths relatively cheaply (constant overhead to the unification algorithm), or computing optimal paths more expensively (in time polynomial in the size of the unification graph).

2.4 Diagnosis in Artificial Intelligence

Explanation-based diagnosis has always been an important issue in AI (Reiter [90], deKleer[29], Wick and Thompson [112], Genesereth [39]). Reiter, de Kleer and others [29, 90] develop a general characterization of system-description, behavior, conflict and diagnosis based on propositional logic. Reiter formulates broad criteria for model-based generation of diagnostic information based on the idea of "minimally conflicting sets." This notion of minimality is used in our framework for untypability in Chapter 5.

3

A review of term unification

The problem of term-unification is concerned with solving equations of terms over abstract algebras. The main components of the formal treatment of unification are the data structures of unification (terms and unification graphs), operations over these data structures (substitutions, solutions, unifiers, and most general unifiers), and relations overs these data structures (term-bisimulations, downward-closed equivalences and unification closures).

The formalism presented in this chapter addresses the unification of finite and circular terms, collectively known as *rational terms*. The problem of unification over rational terms was first studied by Huet [51]. Other studies of unification for infinite terms, for example Jaffar [53], Mukai [80], and Colmeraur [22], have been concerned more with developing efficient algorithms rather than building formalisms. Rational terms have been defined as infinite trees using metric spaces and complete partial orders (Courcelle [24]). In this chapter, they are modeled as non-wellfounded sets (Aczel [1]) obtained as solutions to flat systems of set equations (Barwise and Moss [6]).

The simplest variant of the unification problem, known as unification in the empty theory, consists of equations over inductive terms. Solutions, or unifiers of equations of wellfounded terms, may however result in unifiers whose range includes non-wellfounded terms.[1] When considering the unification problem for rational terms, all the main results, including conditions for the existence of unifiers and properties of most general unifiers, continue to hold. However, terms, equality, and substitutions need to be defined co-inductively rather than inductively, and this requires a reworking of the proofs of the basic results.

The origins of the unification problem can be traced to the 1930's in the work of Herbrand [45]. In the 1960's, Robinson coined the term "unification" and showed how it lay at the heart of resolution-based theorem proving [91]. Robinson defined the notion of a most general unifier and proposed an algorithm for computing mgu's. Since then, the properties of substitutions and unifiers have been studied extensively (Eder [36], Lassez et al. [61]). Furthermore, the many variants and generalizations of the unification problem (E-unification, higher-order unification, semi-unification etc.) and its diverse applications to areas of theorem proving, artificial intelligence, databases, type inference, and logic programming (Prolog) has since spawned a vast area of research. Surveys of unification,

[1]Enabling non-wellfounded solutions is done by removing the so-called "occurs-check" in unification implementations.

including its applications in other areas can be found in Knight [59], and Baader and Siekmann [3].

The formalism presented here emphasizes the relational basis of the unification problem and its solution, as in Le Chenadec [64], Baader and Siekmann [3], and Paterson and Wegman [84], rather than the transformational approach of Martelli and Montanari [68], Lassez et al. [61], and Jouannaud and Kirchner [55]. The relational approach casts unification in terms of connectivity properties of the graph data structures representing the unification problem. The connectivity properties are crucial to constructing a proof theory of unification and the integration of unification proofs into existing unification algorithms.

Section 1 presents the basic definitions, including that of Σ-graphs, the underlying data structure used to model terms. Section 2 defines rational terms as co-inductive types. Section 3 defines of the notion of substitution over rational terms. Section 4 lays out the unification problem and its solution, including unifiability, unification closure, and the computation of most general unifiers.

3.1 Basic Definitions

3.1.1 Relations

Given sets A and B, a (binary) relation R *over* $\langle A, B\rangle$ is a subset of $A \times B$. We write aRb to denote $(a, b) \in R$. When $A = B$, we say R is a relation *over* A. A relation R over a set A is *reflexive* if aRa for each $a \in A$. R is *symmetric* if for all $a, b \in A$, aRb implies bRa. R is *transitive* if for all $a, b, c \in A$, aRb and bRc implies aRc. R is an *equivalence* if R is reflexive, symmetric and transitive. For each $v \in A$, $[v]_R$ denotes the equivalence class of R that contains v. When the equivalence is clear from context, $[v]_R$ is abbreviated $[v]$. R is *non-wellfounded* if there is an infinite sequence $a_0, a_1, \ldots$ of elements of A such that $a_{i+1}Ra_i$ for each $0 \leq i$. The sequence $a_0, a_1, \ldots$ is called a *descending sequence for* R *starting at* a_0. If there is no such sequence, then R is *wellfounded*. R is *cyclic,* or *circular*, if there is a finite sequence $a_0, \ldots a_n$, such that $a_0 = a_n$ and $a_{i+1}Ra_i$, for each $0 \leq i \leq n-1$. Such a sequence is called a *cycle for* R *starting at* a_0. Clearly, if R is circular then it is non-wellfounded. For example, the relation $<$, denoting "less than" on natural numbers is wellfounded, but $>$ denoting "greater than" is non-wellfounded.

If R is a relation over a set A, then the *transitive closure of* R, denoted R^+, is the least transitive relation containing R. The *reflexive transitive closure of* R, denoted R^*, is the least reflexive, transitive relation containing R. It is easy to verify that a relation R on A is cyclic if and only if for some $a \in A$, aR^+a.

$g : A \longrightarrow B$ denotes a (total) function g from the set A to set B. If $S \subseteq A$, then $g[S]$, *the image of* g *on* S is the set $\{g(x) \mid x \in S\}$. If $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then $g \circ f : A \longrightarrow C$, also written gf, is the function defined as $g(f(x))$ for each $x \in A$. For each set A, I_A denotes the identity function on A defined as $I_A(x) = x$ for each $x \in A$.

3.1.2 Directed Graphs

A *directed graph* is a pair $G = \langle V, D\rangle$, where V is a set of *vertices* and $D \subseteq V \times V$ is a set of *directed edges*. The pair $(u, v) \in D$ is written $u \longrightarrow v$ and represents an edge *from* u *to* v. u and v are respectively the *source* and *destination* of the edge $u \longrightarrow v$, and are identified by the projection function *src* and *dest* respectively.

Given a directed graph $G = \langle V, D\rangle$, the *connectivity relations* $\xrightarrow{*}$ and $\xrightarrow{+}$ on V denote the reflexive transitive closure and transitive closure, respectively, of the directed edge relation D. We write $G \models u \longrightarrow v$ if $u \longrightarrow v \in D$. Similarly, we write $G \models u \xrightarrow{*} v$ and $G \models u \xrightarrow{+} v$.

If G is a directed graph containing vertices u and v, then a *path from* u *to* v *in* G is a finite, possibly empty sequence of directed edges $c_1 \ldots c_n$, $0 \leq n$, such that if $0 < n$, then for each $1 \leq i \leq n$, $c_i \in D$, and *dest*$(c_i) =$ *src*(c_{i+1}), and if $n = 0$, then $u = v$. If p is a path, then the *length of* p, denoted $|p|$, is n. p is *empty* if $|p| = 0$. If p is a path from u to v, then u and v are respectively the *source* and *destination of* p, and *src*$(p) = u$, *dest*$(p) = v$, and *endpoints*$(p) = \langle u, v\rangle$.

It is easy to see that given a graph G, $G \models u \xrightarrow{*} v$ if and only if there is a path from u to v in G, and $G \models u \xrightarrow{+} v$ if and only if there is a non-empty path from u to v in G.

3.1.3 Alphabet, sentences, signature

We use **N** to denote the set of natural numbers. An *alphabet* Σ is a set whose elements are called *symbols*. Σ^* is the set of all finite sequences $a_1 \ldots a_n$ such that $n \in \mathbf{N}$, and $a_i, 1 \leq n \in \Sigma$. The elements of Σ^* are called *sentences over* Σ. If $l = a_1 \ldots a_n$ and $l' = a'_1 \ldots a'_m$ are sentences, then the sentence $a_1 \ldots a_n a'_1 \ldots a'_m$ denotes the *concatenation* of l with l'. The empty sentence is denoted ϵ, and $l\epsilon = \epsilon l = l$ for every sentence l. Clearly, concatenation is associative. A Σ-*language* , or *language*, is any subset of Σ^*. The language Σ^+ denotes the set of non-empty sentences over Σ. Σ^0 denotes the set $\{\epsilon\}$.

A binary relation R over Σ^* is a *congruence* if for all sentences x, y, u, v over Σ, xRy implies $uxvRuyv$. If p, q, r are sentences over Σ such that $p = qr$, then q is a *prefix of* p and r is a *suffix of* p. A language $L \subseteq \Sigma^*$ is *prefix closed* (respectively *suffix closed*) if for each $p \in L$, every prefix (respectively suffix) of p is in L.

A *signature* is an alphabet Σ of *functor symbols* equipped with an arity function *arity* : $\Sigma \rightarrow \mathbf{N}$. A signature Σ is usually specified by enumerating its arity function. For instance,

$\{\mathsf{f} \mapsto 2, \mathsf{a} \mapsto 0\}$ denotes a signature with functor symbols f and a with arity 2 and 0 respectively. If $\mathit{arity}(f) = n$, we say f *is n-ary*. 0-ary functors are called *constants*. If Σ is a signature, the alphabet $\Sigma_{\mathbf{N}} = \{f_i | f \in \Sigma,\ 1 \leq i \leq \mathit{arity}(f)\}$ is called the set of *context symbols of* Σ. To avoid double subscripts, $\Sigma_{\mathbf{N}}$ is sometimes written $\Sigma.\mathbf{N}$ and the label f_i is written $f.i$. For example, if $\Sigma = \{\mathsf{f} \mapsto 2, \mathsf{g} \mapsto 1, \mathsf{a} \mapsto 0\}$, then $\Sigma_{\mathbf{N}} = \{\mathsf{f}_1, \mathsf{f}_2, \mathsf{g}_1\}$.

A *tuple* of size $n \in \mathbf{N}$ over a set A is a function $\{0, \ldots, n-1\} \longrightarrow A$. A tuple p of size n is written $(a_0, \ldots, a_{n-1})$, or $\langle a_0, \ldots, a_{n-1} \rangle$. The size of a tuple p is denoted $|p|$. The element $p(0)$ is called the *head* of p and denoted $\mathit{hd}(p)$. The *tail* of p, denoted $\mathit{tl}(p)$, is a tuple of size $|p| - 1$ defined as $\mathit{tl}(p)(i) = p(i+1)$, $0 \leq i < |p| - 1$.

If A is a set, and Σ a signature, $\Sigma(A)$ denotes the set

$$\{(f, a_1, \ldots, a_n) \mid f \in \Sigma, \mathit{arity}(f) = n, a_1, \ldots, a_n \in A\}$$

The tuple $(f, a_1, \ldots, a_n) \in \Sigma(A)$ is written $f(a_1, \ldots, a_n)$, and $f()$ is abbreviated f.

3.1.4 Labeled directed graphs, Σ-graphs

Directed graphs often have labels associated with vertices or edges or both, giving rise to labeled directed graphs. A *labeled directed graph* G is a tuple $\langle \Sigma_V, \Sigma_D, V, L, D \rangle$, where Σ_V and Σ_D are a pair of alphabets, respectively called the *vertex* and *edge labels sets of* G, $L : V \longrightarrow 2^{\Sigma_V}$ is a *vertex labeling function* and $D \subseteq V \times V \times (\Sigma_D \cup \{\epsilon\})$ is the set of *labeled edges of* G. The tuple $\langle u, v, \delta \rangle \in D$ is written $u \xrightarrow{\delta} v$. The source, destination and *label* of $u \xrightarrow{\delta} v$ are respectively u, v and δ, and are identified with the projection functions *src*, *dest* and l. Edges whose labels are in Σ_D are called *branch edges*. A vertex $u \in V$ is *strict* if $L(u) = \emptyset$. u is *homogeneous* if $L(u)$ is either empty or a singleton. G is *homogeneous* if every vertex in V is homogeneous. G is *deterministic* if for any two edges $w \xrightarrow{\delta} u$ and $w \xrightarrow{\delta} v$ in G, $u = v$. G is *guarded* if for all $u, v \in V$, $u \xrightarrow{\epsilon} v$ implies $u = v$. The *underlying directed graph of* G is the directed graph $G' = \langle V, D' \rangle$, where $D' = \{u \longrightarrow v | u \xrightarrow{\delta} v \in D$ for some $\delta\}$.

In a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ with vertices $u, v \in V$, a *labeled path* or *path from* u *to* v *over* G is a possibly empty sequence $c_1 \ldots c_n$, $0 \leq n$ of labeled directed edges $c_1 \ldots c_n$, $0 \leq n$, such that if $0 < n$, then for each $1 \leq i \leq n$, $c_i \in D$, and *dest*$(c_i) =$ *src*(c_{i+1}), and if $n = 0$, then $u = v$. If p is a labeled path, then the *label of* p, denoted $l(p)$, is the sentence over Σ_D obtained by concatenating the labels of each edge in that path. If p is empty, then $l(p)$ is defined to be the empty sentence ϵ. We assume that there is an empty path from every vertex to itself, and this empty path is also denoted ϵ. $G \models p : u \xrightarrow{l} v$ denotes that p is a path from u to v labeled l in G. $G \models u \xrightarrow{l} v$ denotes that there is a path from u to v labeled l in G.

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph, and let R be an equivalence relation on V. R is *downward-closed* if, for each uRu' and directed edges $u \xrightarrow{\delta} v \in D$ and $u' \xrightarrow{\delta} v' \in D$ with the same label δ, it is the case that vRv'. The *quotient* *graph* G/R *of* G is the labeled directed graph $\langle \Sigma_V, \Sigma_D, V_R, L_R, D_R \rangle$ where V_R is the set of equivalence classes of R, $L_R([u]_R) = \cup\{L(v) \mid v \in [u]_R\}$ for each equivalence class $[u]_R$ of R containing the vertex $u \in V$, and $[u]_R \xrightarrow{\delta} [v]_R \in D_R$ if and only if there are $u', v' \in V$ such that uRu', $u' \xrightarrow{\delta} v \in D$, and $v'Rv$.

We now consider a special subclass of labeled directed graphs used for modeling terms and equations between them. Let Σ be a signature. A Σ-*graph* is a labeled directed graph $G = \langle \Sigma, \Sigma_{\mathbf{N}}, V, L, D \rangle$ with the restriction that G is deterministic and homogeneous, and, if $f.i \in \Sigma_{\mathbf{N}}$ and $u \xrightarrow{f.i} v \in D$, then $f \in L(u)$. A Σ-graph $\langle \Sigma, \Sigma_{\mathbf{N}}, V, L, D \rangle$ is abbreviated $\langle V, L, D \rangle$. A Σ graph G is *complete* if for each vertex u in G, if $f \in L(u)$ and *arity*$(f) = n$, then there are vertices $u_1, \ldots u_n$ such that for each i, $u \xrightarrow{f.i} u_i$ is an edge in G.

3.2 Rational Terms

A term is an algebraic expression built from symbols from a signature and a set of variables. Common mathematical usage of the word "term" assumes that it is wellfounded, that is, it can be defined bottom-up, starting from the variables and constants, and building more complex terms using function symbols. More generally, however, terms, modeled as non-wellfounded sets [1], could contain themselves as proper subterms. Such terms, called *rational terms*, are useful for modeling circular data types like recursive environments, objects, and processes. A rational term is a term with a finite, and possibly circular subterm relation. In order to understand the relationship between wellfounded and rational terms, it is useful to define the set of Σ-terms in such a way as to include wellfounded, circular, and non-wellfounded terms.

If Σ is a signature and $\mathbf{V}$ a set of *strict variables* disjoint from Σ, the set $T^{\infty}(\Sigma, \mathbf{V})$ of Σ-*terms over* $\mathbf{V}$, or *terms,* is defined as the largest fixed-point of the domain equation

$$T(\Sigma, \mathbf{V}) = \mathbf{V} + \Sigma(T(\Sigma, \mathbf{V})) \tag{3.1}$$

Thus, a Σ-term is a strict variable, or a *compound term* $\tau = f(\tau_1, \ldots, \tau_n)$, $0 \leq n$, where $f \in \Sigma$, and $\tau_1, \ldots, \tau_n$ are all Σ-terms. f is called the *head* of τ. If $\tau = f(\tau_1, \ldots, \tau_n)$, then each τ_i, $1 \leq i \leq n$, is the *immediate subterm of* τ *at position* i. If τ is a strict variable, then the set of immediate subterms of τ is $\emptyset$. The *proper subterm* relation is the transitive closure of the immediate subterm relation. The *subterm* relation is the reflexive, transitive closure of the immediate subterm relation.

A Σ-term τ is *non-wellfounded* if there is a descending chain for the immediate subterm relation starting at τ. Otherwise, τ is *wellfounded*. τ is *circular* if there is a cycle for the immediate subterm relation starting at some subterm of τ. Otherwise, τ is *non-circular*. τ is *rational* if the set of its subterms is finite. Otherwise, it is *irrational*. Wellfounded terms are also called *inductive* terms. The set of wellfounded Σ-terms over $\mathbf{V}$, denoted $T^{*}(\Sigma, \mathbf{V})$, is the least fixed-point of Equation 3.1. In other words, wellfounded terms are the least set such that each variable in $\mathbf{V}$ is a term, and if $\tau_i, \ldots, \tau_n$, $1 \leq n$ are terms and $f \in \Sigma$ and *arity*$(f) = n$, then $f(\tau_i, \ldots, \tau_n)$ is a term. The set of *rational* Σ*-terms over* $\mathbf{V}$, denoted $T^+(\Sigma, \mathbf{V})$, is also a fixed-point of Equation 3.1. It therefore includes the set of wellfounded Σ-terms. The inclusion relation between the three fixed-points of Equation 3.1 is

$$T^*(\Sigma, \mathbf{V}) \subseteq T^+(\Sigma, \mathbf{V}) \subseteq T^\infty(\Sigma, \mathbf{V})$$

The function *vars* : $T(\Sigma, \mathbf{V}) \longrightarrow 2^{\mathbf{V}}$ maps a term to the set of all variables occurring in that term. Clearly, if τ is rational, *vars*(τ) is finite.

Example 3.2.1: Let $\Sigma = \{(f, 1)\}$ and $V = \{x\}$. Then $T^*(\Sigma, V) = \{x, f(x), f(f(x)), \ldots\}$, and $T^+(\Sigma, V) = T^\infty(\Sigma, V) = T^*(\Sigma, V) \cup \{f(f(f(...)))\}$. $\diamond$

Example 3.2.2: Let $\Sigma = \{(g, 1), (h, 1)\}$ and let $x \in V$. Then the terms $x, g(x), h(x)$ and $g(h(x))$ are all wellfounded. The term $g(g(g(...)))$ is circular, because it contains itself as its (only) proper subterm. Let τ be the term $H_1(H_2(H_3(...)))$, where H_i is g or h depending on whether the ith bit in the binary expansion of the number π is 0 or 1 respectively. Then τ is irrational. $\diamond$

For the rest of the discussion in this chapter, we assume the existence of a set $\mathbf{V}$ of strict variables. The phrase "Σ-term" will refer to an element in $T^+(\Sigma, \mathbf{V})$. Properties of Σ-terms and equations over these terms are the main objects of study in term unification.

3.2.1 Flat Systems

Wellfounded terms are represented as finite trees or finite directed acyclic graphs (DAG's). Rational terms are represented as finite graphs. Wellfounded compound rational terms are represented by a finite textual representation in the usual way: $\mathsf{f}(\tau_1, \ldots, \tau_n)$, for example.

Rational Σ-terms are identified as solutions of a finite system of equations. The equations consist of two disjoint sets of variables: the *parameters* of the system, which are a finite subset of $\mathbf{V}$, and the *indeterminates* of the system, which are also called *functor variables*, and which are variables to which values are assigned. Therefore, a system of equations is a function on indeterminates. Equations allow us to express circularity as well as sharing of subterms. A system usually represents several terms at once. For example, the terms $\tau = \mathsf{f}(\mathsf{g}(x), \tau)$ and $\mathsf{g}(x)$ are represented by the system of equations shown below.

$$\begin{aligned} w &= \mathsf{f}(w', w) \\ w' &= \mathsf{g}(x) \end{aligned}$$

The system of equations has one strict variable x and two functor variables w and w'. There is one equation per functor variable, and each equation is *flat*: subterms are nested one-level deep.

Notation: In this and the following chapters, strict variables will be denoted by x, y, z, t etc., functor variables by w, variables (strict or functor) by u, v, functor variables by f, g, h

etc., and terms by τ. In addition, subscripted and primed variants of these symbols will also be used.

Let Σ be an alphabet and V a set. Formally, a system of *flat Σ-term equations* , or *flat system over V*, for short, is a tuple $T = \langle W, X, b\rangle$, where $X \subset V$ is a finite set of *strict* variables, W is a finite set of *indeterminates* disjoint from V, and $b : W \to \Sigma(X \cup W)$ is a set of *flat equations*. $b(w) = f(u_1, \ldots, u_n)$ is informally written as $w = f(u_1, \ldots, u_n)$. The *label* of a functor variable w, denoted $L(w)$, is the functor symbol *hd*$(b(w))$.

A *pointed system of flat term equations*, or *pointed flat system*, is a pair $\langle v, T\rangle$, where $T = \langle W, X, b\rangle$ is a flat system, and $v \in X \cup W$. A pointed flat system provides a way to refer to a particular subterm of a term represented as a flat system.

Every term τ may be represented by a pointed flat system $\langle \tau, T\rangle$, where $T = \langle W, X, b\rangle$, W is the set of compound subterms of τ, X is the set of strict variables in τ. If $\tau', \tau_1, \ldots, \tau_n$ are all subterms of τ, and $\tau' = f(\tau_1, \ldots, \tau_n)$, then $b(\tau') = f(\tau_1, \ldots, \tau_n)$. Thus, for each compound subterm of τ, there is a flat equation in T.

A mapping s from W to Σ-terms is a *solution* of T if, for each $w = f(u_1, \ldots, u_n) \in b$:

$$s(w) = f(\hat{s}(u_1), \ldots, \hat{s}(u_n)) \tag{3.2}$$

where $\hat{s}(u)$ denotes u if $u \in X$, and $s(u)$ if $u \in W$.

The Solution Lemma formulation of the Anti-Foundation Axiom [6] asserts that every flat system has a unique solution. Furthermore, if a flat system is finite, each w in s is mapped to a term in $T^+(\Sigma, X)$. Thus, families of terms are identified with flat systems.

Example 3.2.3: The family of terms $\{\mathsf{f}(\mathsf{g}(x), z), \mathsf{f}(\mathsf{g}(y), z), \mathsf{g}(x), \mathsf{g}(y), t, x, y, z\}$, where t, x, y, z are strict variables is represented by the flat system $T = \{W, X, b\}$ where $W = \{w_1, w_2, w_3, w_4\}$, $X = \{t, x, y, z\}$, and b is

$$\begin{array}{rclcrcl} w_1 & = & \mathsf{f}(w_3, z) & \quad & w_2 & = & \mathsf{f}(w_4, z) \\ w_3 & = & \mathsf{g}(x) & & w_4 & = & \mathsf{g}(y) \end{array}$$

The mapping

$$\{w_1 \mapsto \mathsf{f}(\mathsf{g}(x), z),\ w_2 \mapsto \mathsf{f}(\mathsf{g}(y), z),\ w_3 \mapsto \mathsf{g}(x),\ w_4 \mapsto \mathsf{g}(y)\}$$

is a solution of T. $\diamond$

3.2.2 Term Graphs

If Σ is a signature, a Σ-*term graph*, or *term graph* is a Σ-graph that is guarded. A flat system $T = \langle W, X, b\rangle$ of Σ-terms over a set of variables V may be represented as a Σ-graph $G = \langle V, L, D\rangle$ in which $V = X \cup W$, $L(v) = \emptyset$ if $v \in X$ and $L(v) = \{\textit{hd}(b(v))\}$ if $v \in W$, and, for each vertex $w \in W$, if $1 \leq i \leq \textit{arity}(L(w))$ and $b(w)(i) = u$, then $w \xrightarrow{f.i} u \in D$. Clearly, G is a Σ-term graph in which the label of every edge is in $\Sigma_{\mathbf{N}}$.

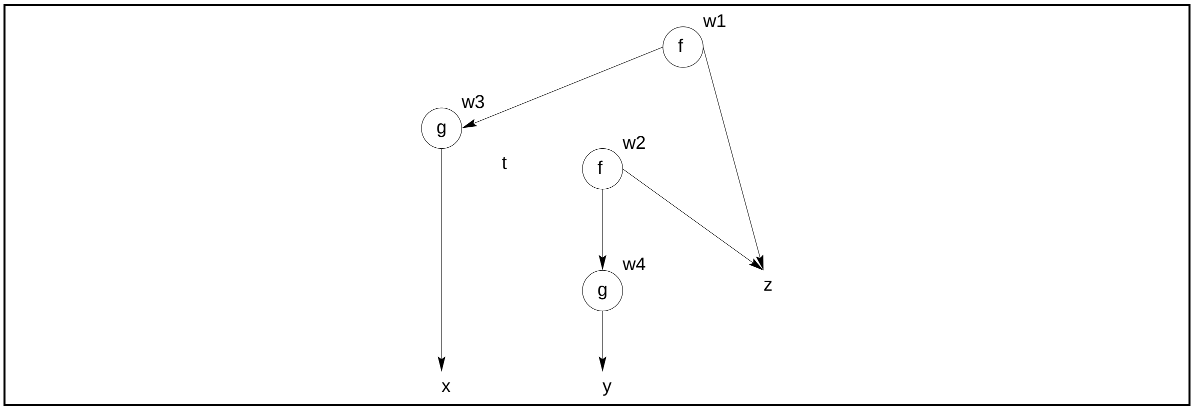


Figure 3.1: Term graph for flat system in Example 3.2.3. The vertex t is isolated.

Also, it is clear that $u \xrightarrow{f.i} v \in G$ if and only if $\hat{s}(u)$ is a compound term whose head is f and whose immediate subterm at position i is $\hat{s}(v)$. If $G \models u \xrightarrow{l} v$, then we say that the subterm $\hat{s}(v)$ *occurs in the context* l *within* $\hat{s}(u)$. If $l = f_1.i_1 f_2.i_2 \ldots f_n.i_n$, where $0 \leq n$, then the sentence $i_1 i_2 \ldots i_n$ is a *position* at which $\hat{s}(v)$ occurs in $\hat{s}(u)$. $\hat{s}(v)$ may occur at multiple contexts within $\hat{s}(u)$, and each context determines a position of the occurrence of $\hat{s}(v)$ in $\hat{s}(u)$.

Example 3.2.4: The term graph of the flat system in Example 3.2.3 is shown in Figure 3.1. Each functor node w with a vertex label f is represented by a circle containing the label f. If the arity of f is n, the directed edges $w \xrightarrow{f.i} u_i$ from w to u_i labeled $f.i$, $1 \leq i \leq n$, are usually pointing downwards and ordered left-to-right. To avoid clutter, the label $f.i$ on the edge is omitted because it can be inferred from the label on w and the relative position of the edge. Strict vertices have no labels and are represented without circles. $\diamond$

It is easy to verify that the Σ-graph representing a flat systems of equations is complete.

Σ-term graphs that are not complete can be thought of as modeling "incomplete Σ-terms" which may have some subterms missing. The missing subterms are often denoted by a hole $\Box$. For example, the term $\mathsf{f}(\Box, \mathsf{g}(\Box, \mathsf{a}))$ denotes an "incomplete Σ-term" with holes.

Flat systems and complete term graphs will be used interchangeably. Thus we write $T = \langle W, X, b \rangle$ to denote a flat system and the complete term graph of T. But we will sometimes work with Σ-term graphs that are incomplete.

3.2.3 Term-bisimulation

For wellfounded rational terms, equality can be defined using induction. In the presence of circularity, however, equality needs to be defined in terms of *observational equivalence*, a notion introduced in the study of automata theory and the algebra of processes [76, 83]. For rational terms, the observation of interest is the possibly infinite, fully-expanded printed representation of the term, where each functor variable is replaced by the right-hand side of its defining equation. For example, if we have two equations satisfying $\{\tau = f(\tau'), \tau' = f(\tau)\}$ both τ and τ' have the infinite printed representation $f(f(f(...)))$, and therefore should denote the same term. The Extensionality Principle for wellfounded terms states that two terms are equal if and only if they have the same outermost functor symbol and their corresponding subterms are equal. However, if we use this principle to prove $\tau = \tau'$, we are lead to the vacuous reasoning $\tau = \tau'$ if and only if $\tau = \tau'$.

Definition 3.2.1: Let Σ be a signature and V a set of variables. A *term-bisimulation* R is a relation over $T^\infty(\Sigma, V)$ such that for each $(\tau, \tau') \in R$, one of the following *term-bisimulation* condition holds:

1. τ and τ' are variables and $\tau = \tau'$, or

2. $\tau = f(\tau_1, \ldots, \tau_n)$ and $\tau' = f(\tau'_1, \ldots, \tau'_n)$, that is, the heads are the same, and for each $1 \leq i \leq n$, $\tau_i R \tau'_i$.

τ and τ' are *bisimilar* if $\tau R \tau'$ for some term-bisimulation R. $\square$

In the presence of rational terms, the Principle of Extensionality needs to be reformulated using the notion of term-bisimulations.

Extensionality Principle for rational terms: Two terms τ and τ' are equal if and only if they are bisimilar.

Proving the equality of non-wellfounded terms using the extensionality principle is referred to as proof by *co-induction*. The *co-inductive* proof for showing equality for rational terms τ and τ' consists of (1) constructing a relation R, (2) proving that R is a term-bisimulation, and (3) verifying that (τ, τ') belongs to R.

It is not hard to verify that $=$, the Equality relation over Σ-terms, is a term-bisimulation. In fact, it is the largest term bisimulation. Therefore, in co-inductive proofs, it is convenient to construct a relation R and verify that the relation $R \cup =$ is a term-bisimulation. Using this “stronger co-inductive” principle, after constructing a relation R, we verify that terms in each pair of R are either equal, or satisfy the term-bisimulation condition.

Example 3.2.5: Consider the terms $\tau_1 = f(\tau_1)$ and $\tau_2 = f(f(\tau_2))$. The relation $R = \{(\tau_1, \tau_2), (\tau_1, f(\tau_2))\}$ is a term-bisimulation. It is easy to verify that each pair (τ_1, τ_2) and $(\tau_1, f(\tau_2))$ satisfies the term-bisimulation conditions: τ_1 and τ_2 are both compound terms, their outermost functor symbols are identical, and their immediate subterms τ_1 and $f(\tau_2)$ are related by R. Similarly, the outermost functor symbols of τ_1 and $f(\tau_2)$ are identical, and their immediate subterms τ_1 and τ_2 are related by R. Hence, by co-induction, $\tau_1 = \tau_2$.

$\diamond$

A general introduction to bisimulations, including a justification for the extensionality principle for non-wellfounded sets, can be found in (Barwise and Moss [6]). A tutorial presentation of co-induction in a co-algebraic framework is discussed in (Jacobs and Rutten [52]).

3.3 Substitutions

Let Σ be a signature and V and V' be a sets of strict variables. A *substitution* s *over* V is a mapping defined over a subset of V. The subset of variables over which s is defined is called the *domain of* s and denoted *dom*(s). s maps each element of *dom*(s) to a Σ-term over V'. The operation *application of a substitution* s to Σ-terms over V, denoted *apply*$(s, \_)$ and abbreviated $\hat{s}$, is defined as the largest set satisfying the following conditions:

1. if $\tau \in$ *dom*(s), then $\hat{s}(\tau) = s(\tau)$.

2. if $\tau \in V -$ *dom*(s), then $\hat{s}(\tau) = \tau$.

3. if $\tau = f(\tau_1, \ldots, \tau_n)$, then $\hat{s}(\tau) = f(\hat{s}(\tau_1), \ldots, \hat{s}(\tau_n))$

The definition of $\hat{s}(\_\,)$ on the domain of rational terms is an example of definition by *co-recursion*. If the application operation were defined using induction (smallest set closed under the above conditions), then its domain would include only the wellfounded Σ-terms over V.

If s is a substitution over V and $\tau \in T^+(\Sigma, V)$, then $\hat{s}(\tau)$ is uniquely determined by the values of $\hat{s}$ on *vars*(τ). The set of variables occurring in s is denoted *vars*(s). The set of *independent variables in* s, denoted *ind*(s), is the set of variables x in *vars*(s) such that *apply*$(s, x) = x$. If V is a set of variables, and $X \subseteq V$, then any two substitutions s_1 and s_2 on V *agree on* X if $\hat{s_1}(x) = \hat{s_2}(x)$ for each $x \in X$. This is abbreviated $s_1 =_X s_2$. It is easy to see that $=_X$ is an equivalence relation. It is easy to show that if s_1 and s_2 agree on *vars*(τ), then $\hat{s_1}(\tau) = \hat{s_2}(\tau)$. Given a substitution s, and a set of variables $X \subseteq$ *dom*(s), the restriction $s|_X$ of s to the domain X denotes the substitution with domain X satisfying the condition $s|_X\,(x) = s(x)$ for each $x \in$ *dom*(s).

Given substitutions r over V' and s over V, their *composition*, written rs, is the substitution over $V \cup V'$ with domain *dom*$(s) \cup$ *dom*(r) satisfying the condition that for each $x \in$ *dom*$(s) \cup$*dom*(r), $(rs)(x) = \hat{r}(\hat{s}(x))$. Composition is associative, so $r(st) = (rs)t$.[2] A substitution is *idempotent* if $ss = s$. The *identity substitution*, denoted I is the substitution with an empty domain. It can be easily verified, that $\hat{I}(\tau) = \tau$, for any Σ-term τ, and $Ir = rI = r$, for any substitution r.

3.4 Unification

We move from systems of flat equations to systems with a more general form of equations. A Σ*-term equation* over a set of variables V, or *term equation* is an unordered pair $\tau \stackrel{?}{=} \tau'$ of rational Σ-terms over V. A *system of* Σ*-term equations* over V is a finite set of Σ-term equations over V. If $\mathcal{E}$ is a set of term equations over V, a substitution s over V is a *unifier of* $\mathcal{E}$ if $\hat{s}(\tau) = \hat{s}(\tau')$ for each $\tau \stackrel{?}{=} \tau' \in \mathcal{E}$. We say that s *unifies* $\mathcal{E}$. s is a *most general unifier (mgu) of* $\mathcal{E}$ if for every unifier s' of $\mathcal{E}$, $s' =_V s's$. While flat systems always have solutions, systems of term equations do not always have unifiers. $\mathcal{E}$ is *unifiable* if a unifier for it exists, otherwise it is *non-unifiable*. As an example, the term equation $\mathsf{f}(x, y) \stackrel{?}{=} \mathsf{g}(u, v)$ is not unifiable. *Term unification* is the problem of computing a unifier for a system of term equations.

[2]For the non-wellfounded setting, showing this property is non-trivial (see [6]).

Example 3.4.6: The following are all unifiers of the system consisting of the single equation $x \stackrel{?}{=} y$:

$$s_1 = \{x \mapsto \mathsf{a}, y \mapsto \mathsf{a}\}, \;\; s_2 = \{x \mapsto z, y \mapsto z\}, \;\; s_3 = \{x \mapsto y\}, \;\; s_4 = \{y \mapsto x\}$$

$\diamond$

The basic unification problem, known as unification in the *empty theory*, consists of solving equations over the free term algebra $T^*(\Sigma, \mathbf{V})$ of inductively generated terms. The unification problem in the empty theory can be generalized to semantic or *E-unification,* in which equations of terms over arbitrary algebras are considered. Other variants of the unification problem consist of unification of λ-calculus terms modulo β and η reduction (higher-order unification [88, 51, 96]), and unification in order-sorted algebras (order-sorted unification [74]). Instances of the unification problem in these settings may admit zero, one, finite, or infinite sets of unifiers. In several cases the computation of the unifier(s) is intractable or undecidable.

The rest of the chapter is devoted to presenting the main results of term unification: representation of term equations as unification graphs, conditions necessary and sufficient for a system of term equations to be unifiable, construction of most general unifiers and their equivalence modulo renaming. The results are centered around the computation of the quotient graph of the unification graph with respect to a relation called the unification closure. A system of term equations is unifiable if and only if the quotient graph is a term graph. If the system of equations is unifiable, then a unique most general unifier modulo renaming exists, and can be computed from the equivalence closure.

3.4.1 Unification graphs

Unification graphs are alternative representations of unification systems and are the main objects of study in the remainder of this chapter. The motivation for representing systems of unification equations as graphs is to express the problem of unifiability of a system of equations in terms of properties of the unification graph. For each term equation $\tau_1 \stackrel{?}{=} \tau_2$ of a system of term equations $\mathcal{E}$, the terms τ_1, τ_2 can be represented respectively as pointed term graphs $\langle u_1, T_1 \rangle$ and $\langle u_2, T_2 \rangle$. The equation between the two can then be represented as an undirected edge (u_1, u_2). Thus $\mathcal{E}$ can be represented as a flat system of equations T along with a set of unordered tuples (u, u'), where u and u' are vertices in T.

A Σ-*unification graph* is a pair $G = \langle T, E \rangle$, where $T = \langle W, X, b \rangle$ is a Σ-term graph and E is a symmetric relation over $W \cup X$. The elements of E are called *equational edges*. $\langle \langle W, X, b \rangle, E \rangle$ is abbreviated $\langle W, X, b, E \rangle$.

Conversely, if $G = \langle T, E \rangle$ is a Σ-unification graph, where T is a Σ-term graph with solution s, the *system of* Σ*-term equations represented by* G is defined as the system of term equations $\{\hat{s}(u) \stackrel{?}{=} \hat{s}(v) | (u, v) \in E\}$. A substitution σ is a *unifier of* G if it is a unifier

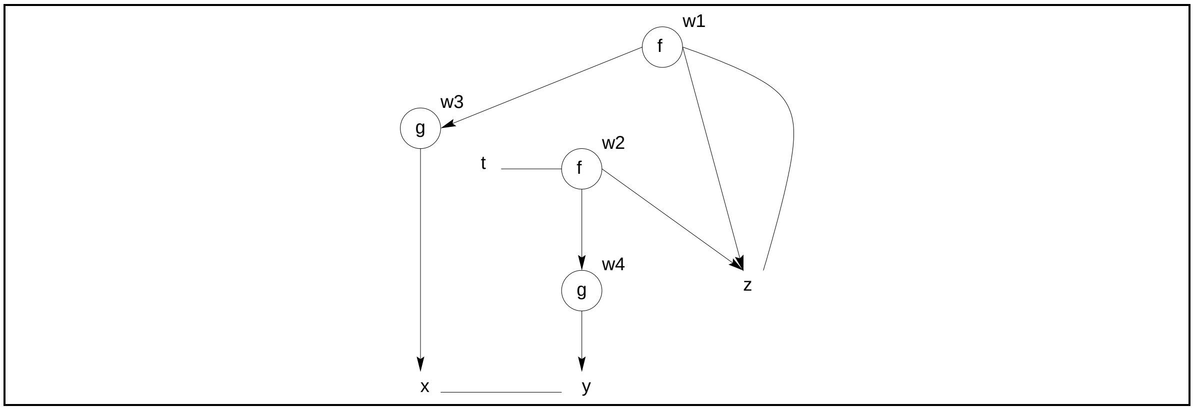


Figure 3.2: Unification graph $G = \langle T, E \rangle$ for Example 3.4.7. G is the term graph T augmented with elements of E, which are represented as undirected edges in G.

of the system of term equations represented by G.

Example 3.4.7: Let $\mathcal{E} = \{t \stackrel{?}{=} \mathsf{f}(\mathsf{g}(y), z), z \stackrel{?}{=} \mathsf{f}(\mathsf{g}(x), z), x \stackrel{?}{=} y\}$, be a system of equations, $T = \langle W, X, b \rangle$ be the term graph defined in Example 3.2.3 with solution s and $E = \{(t, w_2), (z, w_1), (x, y)\}$. The unification graph of $\mathcal{E}$ is the graph $G = \langle T, E \rangle$, shown in Figure 3.2. Elements of E are represented as undirected edges in G. Let τ_1, τ_2 be terms satisfying the equations $\tau_1 = \mathsf{f}(\tau_2, \tau_1)$ and $\tau_2 = \mathsf{g}(\tau_2)$. The following substitution σ' is a unifier of $\mathcal{E}$:

$$\{x \mapsto \tau_2, y \mapsto \tau_2, t \mapsto \tau_1, z \mapsto \tau_1\}$$

$\diamond$

3.4.2 Unifiability

The unifiability of a system of equations may characterized in terms of properties of a certain class of equivalence relations over vertices of the corresponding unification graph. We consider these properties next.

Let $G = \langle T, E\rangle$ be a unification graph. R is an *E-equivalence* if it is an equivalence relation that contains E.

Example 3.4.8: Let $G = \langle W, X, b, E\rangle$ be a unification graph. The equivalence relation obtained by grouping all the vertices $W \cup X$ into a single equivalence class is a downward-closed E-equivalence. The empty relation is not a downward-closed E-equivalence, unless G is empty. $\diamond$

An equivalence class $[u]$ of R is *strict* if $L[u]$ is empty and *non-strict* otherwise. $[u]$ is *homogeneous* if $L([u])$ has at most one element. R is *homogeneous on G* if each equivalence class of R is homogeneous. Quotient graphs of downward-closed homogeneous E-equivalences are term graphs.

Lemma 3.4.1 *(Term graphs from homogeneous downward-closed E-equivalences)*

Let G be a unification graph and R be an E-equivalence on G. Then R is homogeneous and downward-closed if and only if the quotient graph G/R is a term graph.

Proof The proof of the “if” part is straightforward. The quotient graph is a term graph and therefore homogeneous. This implies the homogeneity of R since the vertices of the quotient graph are the equivalence classes of R. Because G/R is deterministic, R is downward-closed.

To prove the converse, let W_R be the set of non-strict equivalence classes of R, and let X_R denote the set of strict equivalence classes. Let $b_R : W_R \longrightarrow \Sigma(W_R \cup X_R)$ be defined as $b_R([w]) = f([u_1], \ldots, [u_n])$ where $w = f(u_1, \ldots, u_n)$ in b. Since R is homogeneous, each non-strict equivalence class of R has a unique label. Since R is downward-closed, $[u] \xrightarrow{f.i} [v]$ and $p \xrightarrow{f.i} [v']$ implies that $[v] = [v']$. Therefore b_R is well-defined. Therefore, G/R is the term graph $\langle W_R, X_R, b_R \rangle$. $\dashv$

Let $G = \langle T, E \rangle$ be a unification graph where $T = \langle W, X, b \rangle$ with solution s. Let R be a homogeneous downward-closed E-equivalence on G with the quotient term graph $G/R = \langle W_R, X_R, b_R \rangle$. Let s_R be the solution of G/R. The *R-quotient substitution on G*, denoted σ_R, is the substitution $s_R[]_R : W \cup X \longrightarrow T^+(\Sigma, X_R)$.

Lemma 3.4.2 *(Restriction of R-quotient substitution)*

Let $T = \langle W, X, b \rangle$ be a term graph with solution s. Let $G = \langle T, E \rangle$ be a unification graph, and let R be a homogeneous downward-closed E-equivalence on G. Let σ_R be the R-quotient substitution on G. Then $\sigma_R =_{X \cup W} \sigma_R s =_{X \cup W} \sigma_R|_X\ s$.

Proof The proof that $\sigma_R(u) = \mathit{apply}(\sigma_R s, u)$ for $u \in X \cup W$ is by co-induction. The set

$$Q = \{\langle \sigma_R(u), \mathit{apply}(\sigma_R s, u)\rangle \mid u \in W \cup X\}$$

is a term bisimulation. In each of the following cases, the pair $\langle \sigma_R(u), \mathit{apply}(\sigma_R s, u)\rangle$ satisfies the term bisimulation conditions:

1. $u \in X$: both elements of the pair reduce to $\sigma_R(u)$.

2. $u \in W$: $\sigma_R(u) = \mathit{apply}(s_R, [u]_R)$. Let $b(u) = f(u_1, \ldots, u_n)$. Then $\sigma_R(u)$ reduces to $f(\sigma_R(u_1), \ldots, \sigma_R(u_n))$. $\mathit{apply}(\sigma_R s, u)$ reduces to $f(\mathit{apply}(\sigma_R s, u_1), \ldots, \mathit{apply}(\sigma_R s, u_n))$. But the pair $\langle \sigma_R(u_i), \mathit{apply}(\sigma_R s, u_i)\rangle \in Q$, for $1 \leq i \leq n$, implying that the pair $\langle \sigma_R(u), \mathit{apply}(\sigma_R s, u)\rangle$ satisfies the 2nd term-bisimulation condition.

To show that $\sigma_R s =_{X \cup W} \sigma_R|_X\ s$, let $u \in X \cup W$. $\mathit{apply}(\sigma_R|_X\ s, u) = \mathit{apply}(\sigma_R|_X, \hat{s}(u))$, and $\mathit{apply}(\sigma_R s, u) = \mathit{apply}(\sigma_R, \hat{s}(u))$. Since $\hat{s}(u)$ is a Σ-term over X, and σ_R and $\sigma_R|_X$ agree on X, the result follows. $\dashv$

We are now ready to state and prove the main result regarding unifiability: A unification graph G is unifiable if and only if there is a downward-closed E-equivalence R on its vertices such that the quotient graph G/R is a term graph. This result shows how unifiers and downward-closed E-equivalences may be derived from each other. This formulation of unifiability is well-known. For example, see Lemma 1 of Paterson and Wegman [84] for the case when the quotient graph is wellfounded. Courcelle [24] proves the result for the more general case when R is rational using cpo's and metric spaces. The proof shown here is based on co-induction.

The first part of the result states that if G is a unification graph and R a homogeneous E-equivalence on G, then the R-quotient substitution on G is a unifier of G.

Theorem 3.4.1 *(Unifier from homogeneous, downward-closed E-equivalence)*

Let $G = \langle T, E\rangle$ be a unification graph where $T = \langle W, X, b\rangle$ is a term graph over $\mathbf{V}$ whose solution is s. Let R be a homogeneous downward-closed E-equivalence on G, and let s_R be the solution of the term graph G/R. Let σ_R denote the R-quotient substitution $s_R[]_R$. Then $\sigma_R|_X$ is a unifier of G.

Proof Since $X \subseteq \mathbf{V}$, $\sigma_R|_X$ is a substitution over $\mathbf{V}$. We verify that for each $(u, v) \in E$, $\mathit{apply}(\sigma_R|_X\ s, u) = \mathit{apply}(\sigma_R|_X\ s, v)$

By Lemma 3.4.2 $\mathit{apply}(\sigma_R|_X\ s, u) = \sigma_R(u)$. Since R contains E, $[u]_R = [v]_R$, implying $\sigma_R(u) = \sigma_R(v)$, and from this the result follows. $\dashv$

Let $G = \langle T, E\rangle$ be a unification graph, where $T = \langle W, X, b\rangle$ is a term graph whose solution is s. Let σ be a unifier of G. Then the equivalence relation

$$R = \{(u, v) \mid u, v \in W \cup X, \mathit{apply}(\sigma s, u) = \mathit{apply}(\sigma s, v)\}$$

is called the *equivalence induced by* σ *on* G.

The proof of the converse of Theorem 3.4.1 is based on showing that equivalence induced by a unifier of a unification graph $G = \langle T, E \rangle$ is homogeneous and downward-closed.

Theorem 3.4.2 *(Homogeneous downward-closed* E*-equivalence from unifier)*

Let σ *be a unifier of a unification graph* $G = \langle T, E \rangle$*, where* $T = \langle W, X, b \rangle$*. Let* R *be the equivalence induced by* σ *on* G*. Then* R *is a homogeneous downward-closed* E*-equivalence.*

Proof R is an equivalence by construction. Since σ unifies G, R includes E. It is clear that R is homogeneous. To show that R is downward-closed, suppose $b(w) = f(u_1, \ldots, u_n)$ and $b(w') = f(v_1, \ldots, v_n)$, and wRw'. Then

$$\begin{aligned} \textit{apply}(\sigma s, w) &= f(\textit{apply}(\sigma s, u_1), \ldots \textit{apply}(\sigma s, u_n)) \\ \textit{apply}(\sigma s, w') &= f(\textit{apply}(\sigma s, v_1), \ldots \textit{apply}(\sigma s, v_n)) \end{aligned}$$

By the subterm condition of equality of rational terms, $\textit{apply}(\sigma s, u_i) = \textit{apply}(\sigma s, v_i)$, for for $1 \le i \le n$, proving R is downward-closed. $\dashv$

EQ	$\dfrac{}{u \sim_G v}$	$(u, v) \in E$
REF	$\dfrac{}{u \sim_G u}$	$u \in G$
SYM	$\dfrac{v \sim_G u}{u \sim_G v}$	
TRANS	$\dfrac{u \sim_G v' \qquad v' \sim_G v}{u \sim_G v}$	
DN	$\dfrac{w \sim_G w'}{u_i \sim_G v_i}$	$b(w) = f(u_1, \ldots, u_n)$ $b(w') = f(v_1, \ldots, v_n)$ $1 \leq i \leq n$

Figure 3.3: The Unification closure $\sim_G$ of a graph $G = \langle\langle W, X, b\rangle, E\rangle$.

3.4.3 Unification closure

A downward-closed E-equivalence of particular interest is the *unification closure* of G, denoted $\sim_G$, or $\sim$, which is the smallest downward-closed E-equivalence on G. In other words, $\sim_G$ is the least set closed under the rules in Figure 3.3.

The specialization of Theorem 3.4.1 and 3.4.2 for the case of unification closure is significant from the point of view of implementation because the unifiability of G is decided by examining the homogeneity of $\sim$.

Lemma 3.4.3 *(Unification closure and unifiers)*

Let G be a unification graph with a unifier σ. Let $u, v \in G$. Then $u \sim v$ implies $\hat{\sigma}(\hat{s}(u)) = \hat{\sigma}(\hat{s}(v))$.

Proof Since $\sim$ is the smallest downward-closed E-equivalence, it is contained in the equivalence induced by σ on G. ⊣

Corollary 3.4.1 *(Unification Closure)*

G is unifiable if and only if the unification closure $\sim$ of G is homogeneous.

Proof If $\sim$ is homogeneous, then by Theorem 3.4.1, the substitution derived from $\sim$ is a unifier. If G is unifiable by some unifier σ, then by Lemma 3.4.3, $\sim$ is contained in the equivalence R induced by σ on G. By Theorem 3.4.2, R is homogeneous. Therefore $\sim$ is homogeneous. ⊣

The following example illustrates the unifier derived by the unification closure, and the homogeneous downward-closed E-equivalence induced by that unifier over a unification graph:

Example 3.4.9: The set of equivalence classes of the unification closure $\sim$ of graph G of Figure 3.2 is

$$\{\{z, w_1\}, \{t, w_2\}, \{x, y\}, \{w_3\}, \{w_4\}\}$$

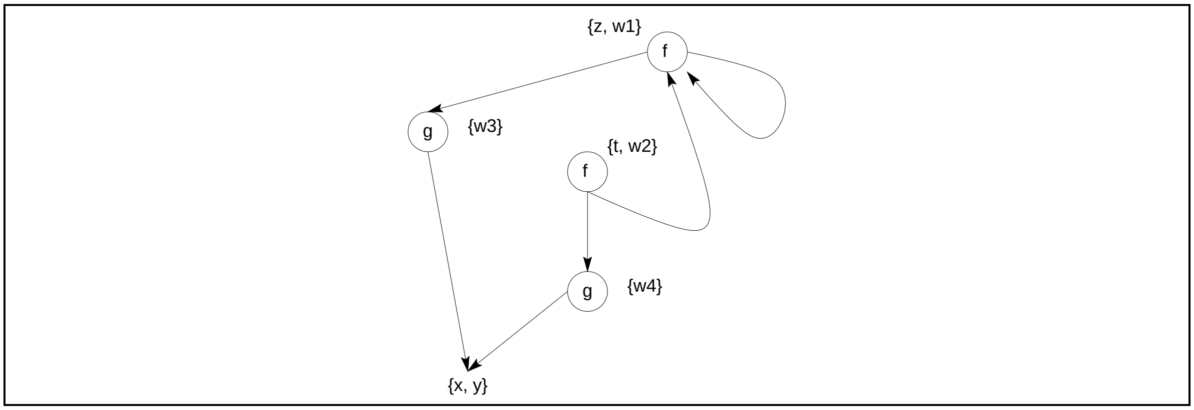


Figure 3.4: Quotient graph $G/\!\sim$ of the graph G in Figure 3.2

It is easy to verify that $\sim$ is homogeneous. The quotient term graph $G/\!\sim = \langle W_\sim, X_\sim, b_\sim \rangle$ is shown in Figure 3.4, with

$$W_\sim = \{\{z, w_1\}, \{t, w_2\}, \{w_3\}, \{w_4\}\}$$

the set of strict variables $X_\sim = \{\{x, y\}\}$, and $b_\sim$ equal to

$$\begin{aligned}
\{z, w_1\} &= \mathsf{f}(\{w_3\}, \{z, w_1\}) \\
\{t, w_2\} &= \mathsf{f}(\{w_4\}, \{z, w_1\}) \\
\{w_3\} &= \mathsf{g}(\{x, y\}) \\
\{w_4\} &= \mathsf{g}(\{x, y\})
\end{aligned}$$

Let τ be the term defined as $\tau = \mathsf{f}(\mathsf{g}(\{x, y\}), \tau)$. It is easy to verify that the solution $s_\sim$ of

$G/\sim$ is the mapping

$$\begin{aligned}
s_\sim(\{x,y\}) &= \{x,y\} \\
s_\sim(\{z,w_1\}) &= \tau \\
s_\sim(\{t,w_2\}) &= \tau \\
s_\sim(\{w_3\}) &= \mathsf{g}(\{x,y\}) \\
s_\sim(\{w_3\}) &= \mathsf{g}(\{x,y\})
\end{aligned}$$

The $\sim$-quotient unifier $\sigma_\sim = s_\sim[]_\sim$ of G is equal to

$$\{x \mapsto \{x,y\}, y \mapsto \{x,y\}, z \mapsto \tau, t \mapsto \tau\}$$

It is easy to verify that the $\sim$-quotient unifier is indeed a unifier of G. The equivalence classes of the relation R induced by $\sigma_\sim$ on G are

$$\{\{x,y\}, \{t,z,w_1,w_2\}, \{w_3,w_4\}\}$$

It is easily verifiable that R is a homogeneous, downward-closed E-equivalence. The quotient term graph G/R is shown in Figure 3.5. $\diamond$

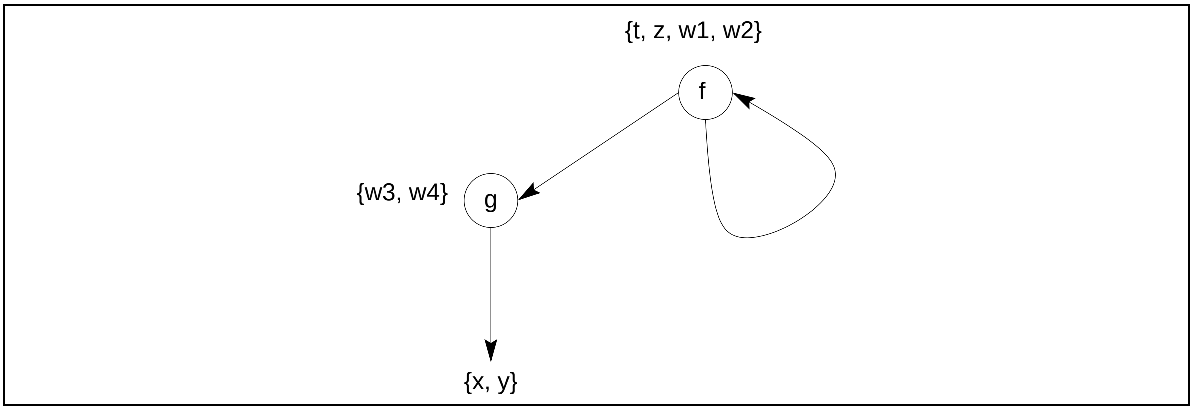


Figure 3.5: Quotient graph G/R of the graph G of Figure 3.2, where R is the equivalence induced by the unifier $\sigma_\sim$ on G computed in Example 3.4.9.

3.4.4 Construction of Most General Unifiers

When a unification graph G is non-unifiable, the non-unifiability of G can be traced to the non-homogeneity of the unification closure $\sim$ of G. On the other hand, when G is unifiable, $\sim$ can be used to construct a most general unifier of G.

Let R be an equivalence relation on the vertices of a unification graph G. A *representative function on* R is a function that maps each strict equivalence class of R to a member of that equivalence class.

Lemma 3.4.4 *(Unification closure and most general unifier)*

Let $G = \langle W, X, b, E \rangle$ *be a unification graph, where* $X \subset \mathbf{V}$ *is the set of strict variables in* G. *Let the unification closure of* $\sim$ *be homogeneous. Let* $G/\!\sim\, = \langle W_\sim, X_\sim, b_\sim \rangle$ *be the quotient term graph with a solution* $s_\sim$. *Let* $\sigma_\sim = s_\sim[]_\sim$ *be the* $\sim$*-quotient substitution on* G. *Let* $r : X_\sim \longrightarrow X$ *be a representative function on the strict equivalence classes* $X_\sim$ *of*

$\sim$. *Then* $(r\sigma_\sim)\,|_X$ *is an mgu of* G.

Proof

We first verify that $(r\sigma_\sim)\,|_X$ is a unifier of G. Note that its domain X is a subset of $\mathbf{V}$. Let $u \in W \cup X$. Since *vars*$(\hat{s}(u)) \subseteq X$, and $r\sigma_\sim =_X r\sigma_\sim|_X$, *apply*$((r\sigma_\sim)\,|_X\; s, u) =$ *apply*$(r\sigma_\sim s, u) =$ *apply*$(r, \sigma_\sim(u))$, the last equality due to Lemma 3.4.2. From this, and because $u \sim v$ implies $\sigma_\sim(u) = \sigma_\sim(v)$, it follows that $(r\sigma_\sim)\,|_X$ is a unifier of G.

In order to show that $(r\sigma_\sim)\,|_X$ is an mgu of G, let σ' be a unifier of G. We show *apply*$(\sigma', x) =$ *apply*$(\sigma'(r\sigma_\sim)\,|_X, x)$ for each $x \in \mathbf{V}$. There are two cases:

1. $x \in \mathbf{V} - X$: Then *apply*$(\sigma'(r\sigma_\sim)\,|_X, x) = \sigma'(x)$ and the result follows.

2. $x \in X$: Let $Q = \{\langle$*apply*$(\sigma' s, u),$ *apply*$(\sigma' r\sigma_\sim, u)\rangle | u \in X \cup W\}$. Since $\hat{s}(x) = x$, and $(r\sigma_\sim)\,|_X\;(x) = r\sigma_\sim(x)$ when $x \in X$, it follows that $\langle\hat{\sigma'}(x),$ *apply*$(\sigma'(r\sigma_\sim)\,|_X, x\rangle \in Q$. In order to show $\hat{\sigma}'(x) =$ *apply*$(\sigma'(r\sigma_\sim)\,|_X, x)$, we show that Q is a a term-bisimulation. There are two cases:

   (a) $[u]_\sim$ is strict: then $u \in X$, and $s_\sim[u]_\sim = [u]_\sim$. Hence $rs_\sim[]_\sim(u) = r([u]_\sim)$, which is equal to $s(r([u]_\sim))$, since $r([u]_\sim) \in [u]_\sim$ and $[u]_\sim$ is strict. But $r([u]_\sim) \sim u$, hence by Lemma 3.4.3, *apply*$(\sigma' s, u)) =$ *apply*$(\sigma' s, r([u]_\sim))$ and the result follows.

   (b) $[u]_\sim$ is non-strict: Let $u \sim w$ for some w such that $b(w) = f(u_1, \ldots u_n)$. By Lemma 3.4.3, *apply*$(\sigma' s, u) =$ *apply*$(\sigma' s, w)$, which reduces to

$$f(\textit{apply}(\sigma' s, u_1), \ldots, \textit{apply}(\sigma' s, u_n))$$

Also, $\sigma_\sim(u) = s_\sim([u]_\sim)$, since $[u]_\sim$ is a non-strict vertex in the quotient term graph $G/\!\sim$. $[u]_\sim = f([u_1]_\sim, \ldots, [u_n]_\sim)$, therefore $\textit{apply}(\sigma' r\sigma_\sim, u)$ reduces to

$$f(\textit{apply}(\sigma' r\sigma_\sim, u_1), \ldots, \textit{apply}(\sigma' r\sigma_\sim, u_n))$$

Thus, $\langle \textit{apply}(\sigma' s, u), \textit{apply}(\sigma' r\sigma_\sim, u)\rangle$ satisfies the 2nd term-bisimulation condition because $\langle \textit{apply}(\sigma' s, u_i), \textit{apply}(\sigma' r\sigma_\sim, u_i)\rangle \in Q$, $1 \leq i \leq n$.

$\dashv$

Hence, an mgu of a unifiable unification graph G can be obtained by composing a representative function with the $\sim$-quotient substitution of G. The next lemma shows that *every* mgu of G is a composition of a representative function and the $\sim$-quotient substitution of G. Thus, while the $\sim$-quotient substitution is a unifier but not an mgu of G, each mgu of G may be built from G by composing different representative functions with it.

Lemma 3.4.5 *(Equivalence of mgu's modulo renaming)*

Let $G = \langle T, E\rangle$ be a unifiable unification graph where $T = \langle W, X, b\rangle$ is a term graph with solution s. Let $s_\sim$ be the solution of the quotient term graph $G/\!\sim$, and let $\sigma_\sim = s_\sim[]_\sim$ be the $\sim$-quotient substitution of G. Let σ' be an mgu of G. Let $r : X_\sim \longrightarrow X$ be defined as $r([u]) = \hat{\sigma'}(u)$. Then $r([x]) \in [x]$ and $\sigma' = (r\sigma_\sim)\,|_X$.

Proof It is easy to verify that r is a representative function for $\sim$. By Lemma 3.4.4, $(r\sigma_\sim)\,|_X$ is an mgu of G. Since the domain of $(r\sigma_\sim)\,|_X$ is X, it follows that *dom*$(\sigma') \subseteq X$.

We verify that *apply*$(\sigma', u) =$ *apply*$((r\sigma_\sim)\,|_X, u)$ for each $x \in \mathbf{V}$. There are two cases:

1. $u \in \mathbf{V} - X$: both $\hat{\sigma'}(u)$ and *apply*$((r\sigma_\sim)\,|_X, u)$ reduce to u.

2. $u \in X$: Let $Q = \{\langle$*apply*$(\sigma' s, u),$ *apply*$(r\sigma_\sim, u)\rangle | u \in X \cup W\}$. Since $\hat{s}(u) = u$, and $\sigma_\sim\,|_X\;(u) = \sigma(u)$ when $u \in X$, $\{\langle$*apply*$(\sigma', u),$ *apply*$(r\sigma_\sim, u)\rangle | u \in X\} \subseteq Q$. It is easy to verify that Q is a is a term-bisimulation.

$\dashv$

Thus, an mgu of a unifiable unification graph is unique modulo renaming, with the understanding that the renaming refers to the renaming substitution from the representatives of strict equivalence classes used in one mgu to those of the other mgu. This is illustrated in the next example.

Example 3.4.10: Let $\mathcal{E} = \{u \stackrel{?}{=} \mathsf{f}(x, y), y \stackrel{?}{=} z, x \stackrel{?}{=} v\}$ be a system of equations whose unification graph is $G = \langle W, X, b, E\rangle$ where $W = \{w\}$, $X = \{x, y, z, u, v\}$, $b = \{w = \mathsf{f}(x, y)\}$, and $E = \{(u, w), (y, z), (x, v)\}$. Then the set of equivalence classes of the unification closure $\sim$ of G is $\{\{w, u\}, \{x, v\}, \{y, z\}\}$. The quotient graph is $\langle W_\sim, X_\sim, b_\sim\rangle$, where $W_\sim = \{w, u\}$, $X_\sim = \{\{x, v\}, \{y, z\}\}$. The $\sim$-quotient unifier $\sigma_\sim$ is

$$\sigma_\sim = \{x \mapsto \{x, v\}, v \mapsto \{x, v\}, y \mapsto \{y, z\}, z \mapsto \{y, z\}, u \mapsto \mathsf{f}(\{x, v\}, \{y, z\})\}$$

Let $r_1, r_2 : X_\sim \longrightarrow X$ be two representative functions, and let $\sigma_1 = (r_1\sigma_\sim)\mid_X$, and $\sigma_2 = (r_2\sigma_\sim)\mid_X$ be two mgu's:

$$
\begin{aligned}
r_1 &= \{\{x, v\} \mapsto x, \{y, z\} \mapsto y\} \\
r_2 &= \{\{x, v\} \mapsto v, \{y, z\} \mapsto z\} \\
\sigma_1 &= \{x \mapsto x, v \mapsto x, y \mapsto y, z \mapsto y, u \mapsto \mathsf{f}(x, y)\} \\
\sigma_2 &= \{x \mapsto v, v \mapsto v, y \mapsto z, z \mapsto z, u \mapsto \mathsf{f}(v, z)\}
\end{aligned}
$$

Here *ind*$(\sigma_1) = \{x, y\}$, and *ind*$(\sigma_2) = \{v, z\}$. Note that $r_i([u]) = \sigma_i(u), u \in X$, for $i \in \{1, 2\}$. The renaming function $\rho_{12} = \{x \mapsto v, y \mapsto z\}$ maps the range of r_1 to the range of r_2, and the renaming function $\rho_{21} = \{x \mapsto v, y \mapsto z\}$ maps the range of r_2 to the range of r_1. ρ_{12} and ρ_{21} are inverses of each other, and $\sigma_2 = \rho_{12}\sigma_1$, and $\sigma_1 = \rho_{21}\sigma_2$. This shows that σ_1 and σ_2 are equal modulo renaming. $\diamond$

If E is a system of equations with mgu s, the set of *independent variables of E under s* is defined as $\{t \in \textit{vars}(E) \mid s(t) = t\}$. It is clear that if s is an mgu of a system of equations E, $s(t) = t$ for all variables $t \notin \textit{vars}(E)$. For this reason, we may assume that if s is an mgu of E, *vars*$(s) \subseteq$ *vars*(E).

3.5 The unification algorithm

This section presents a simple implementation of the unification algorithm for well-founded terms. The algorithm takes a pair of pointed term graphs which are directed acyclic graphs, and determines if the terms represented by them are unifiable.

The first formal account of the term unification problem and the discovery of the unification algorithm is generally credited to J. A. Robinson [91]. The original algorithm by Robinson, based on a representation of terms by means of linear sequence of symbols was exponential in time and space complexity. The space complexity was later improved by using a tabular data structure Robinson [92]. The idea of representing terms as graphs with structure sharing, proposed by Boyer and Moore [11], greatly improved the efficiency of representing terms. Algorithms with quadratic time complexity were proposed by Venturini-Zilli [105], and Corbin and Bidoit [23]. Almost linear time algorithms were proposed by G. Huet [51], L.D. Baxter [7]). Finally, linear time unification algorithms were proposed by Paterson and Wegman [84] and Martelli and Montanari [67].

The unification algorithm is presented as the procedure *unify* and three auxiliary procedures *union*, *find* and *occurs?* shown in Figures 3.6 through 3.8.

The unification graph is represented as a data structure consisting of a set of variables (vertices) linked together by pointers. Each variable has a *type*, which is either strict or functor. Each functor variable v has a field $v.L$ denoting the functor label of v. If the arity

```
procedure unify(v1, v2) =
   let r1 = find(v1) and r2 = find(v2)
      in  if r1 = r2 then return
       else case r1.type, r2.type
              strict, strict: union(r1, r2)
              functor, strict: unify(v2, v1)
              strict, functor: let ans = occurs?(r2, r1)
                          in case ans
                             no: union(r1, r2)
                             yes: fail(CYCLE)
              functor,functor:
                 if r1.L ≠ r2.L then fail(CLASH)
                 else
                    union(r1, r2);
                    for i = 1 to arity(r1.L) do
                       unify(r1.child(i), r2.child(i))
```

Figure 3.6: Unification algorithm: procedure *unify*

```
procedure union(r1, r2) = r1.binding := r2

procedure find(v) =
   if unbound?(v) then return v
   else let v' = v.binding
          in let r = find(v') in return r
```

Figure 3.7: Unification algorithm: procedures *union* and *find*

```
procedure occurs?(v1, v2) =
    let r1 = find(v1) and r2 = find(v2)
        in if r1 = r2 then return yes
              else case r1.type
                     strict: return no
                     functor:
                        for i = 1 to arity(r1.L) do
                             let ans = occurs?(r1.child(i), r2)
                                 in case ans
                                      no: continue
                                      else: return yes
                        return no
```

Figure 3.8: Unification algorithm: procedure *occurs?*

of $v.L$ is n, then the field $v.\mathit{child}(i)$, $1 \leq i \leq n$, denotes the vertex at the ith position in the sequence of children of v. Additionally, for each variable, the field $v.\mathit{binding}(v)$ denotes a pointer that is either nil, in which case *unbound?*(v) is true, or points to another vertex. Strict vertices are shared and functor vertices may be shared. Every vertex defines a unique term consisting of the child vertices that can be reached from that vertex.

The input to the procedure *unify* is a pair of vertices v_1, v_2. The procedure *unify* either succeeds, or fails, in which case the terms τ_1 and τ_2 denoted by the vertices v_1 and v_2 respectively do not unify. The algorithm works by modifying the *binding* field of variables to incrementally construct an equivalence relation $\sim$ on the vertices of the unification graph G defined by the term equation $\tau_1 \stackrel{?}{=} \tau_2$. When the algorithm terminates with success, the relation $\sim$ is equal to the $\sim_G$, the unification closure of G. Each equivalence class of $\sim$

is maintained as a tree of vertices linked by the *binding* pointer whose root is representative vertex of the equivalence class. The *binding* pointer is manipulated by the following auxiliary procedures:

1. The procedure *find*(v) iteratively follows the *binding* pointer and returns the representative vertex of the equivalence class of $\sim$ containing v. The *binding* pointer of the representative vertex is assumed to be nil. The representative of a non-strict equivalence class (one containing at least one functor vertex) is always a functor vertex.

2. The procedure *union*(u, v) sets the *binding* pointer of u to be equal to v, where u and v are representative of different equivalence classes, thereby merging the two equivalence classes into one whose representative is v.

3. The procedure *occurs?*(u, v) returns yes or no depending on whether the term denoted by the equivalence class containing v is a subterm of the term denoted by the equivalence class containing u in the quotient graph $G/\sim$.

The *occurs?* predicate is also known as the *occurs check* and is used by the *unify* procedure to determine if the unification closure is cyclic. In the "standard" version of the unification algorithm, which is shown in Figure 3.6 and which computes acyclic mgu's of term equations consisting of wellfounded terms, *unify* fails with CYCLE if the occurs check returns a yes. For unification of rational terms, where the terms or the unification closure may be cyclic, the occurs check can simply be omitted from the *unify* procedure. Often, unification algorithms omit the occurs check from the *unify* procedure and perform the test in a separate phase.

The call *unify*(v_1, v_2) first computes the representatives r_1 and r_2 of the equivalence classes containing the vertices v_1 and v_2 respectively (line 2). If the representatives are identical, then u and v are already unified, and there is nothing to do. If the two are distinct, then if both equivalence classes are strict, then the two are merged (line 5). Line 6 reverses the arguments in order to ensure that the representative r_2 of the merged equivalence class is a functor variable if the merged equivalence class is non-strict. Line 7 tests if the term denoted by r_1 occurs in the term denoted by r_2 in the quotient graph $G/\sim$. If the occurs-check is false, the equivalence classes are merged (line 9), otherwise the procedure *unify* fails, indicating that the unification closure is cyclic. If both representatives are functor nodes (line 11), and their labels do not match, *unify* fails signaling a CLASH (line 12). Otherwise, the two equivalence classes are merged (line 14) and the the corresponding children of the two functor nodes are unified (lines 15, 16).

The algorithm presented here operates on term graphs and uses linear space but exponential time in size of its input. However, this naïve implementation lends itself to ease of exposition, and later on, to easy modification. The source of exponential behavior is the occurs-check function. With better data structures, however, occurs-check can be easily implementated in linear time, bringing the complexity of the algorithm down to quadratic. By moving the occurs-check "offline,", that is, out of *unify*, and implementing *find* and *union* using a technique called path compression, based on Tarjan's union-find algorithm [99], the complexity of the algorithm can be brought down to almost linear time. Finally using Paterson and Wegman's algorithm [84], the complexity of unification for wellfounded terms can be brought down to linear time.

The unification algorithm maintains the following invariants:

Lemma 3.5.1 *(Unification invariants)*

Let G be a unification graph of a term equation $\tau_1 \overset{?}{=} \tau_2$ between the terms τ_1 and τ_2 represented by the term graphs rooted at vertices v_1 and v_2. Let the unification closure of G be denoted $\sim$. Then the following invariants are maintained by the unification algorithm:

1. *For each call unify(u, v), $G \models u \sim v$.*
2. *For each call union(r_1, r_2), $G \models r_1 \sim r_2$.*
3. *If find$(v) = r$, then $G \models v \sim r$.*
4. *If occurs?$(u, v) = \mathsf{yes}$, then $G/\!\sim \models [u]_\sim \xrightarrow{*} [v]_\sim$.*

Proof (Sketch)

The invariant (1) is true for the "top-level" call. For subsequent calls, the invariant is true because of the equivalence and downward-closure of $\sim$. Similarly, for each call to *union*(r_1, r_2) inside a call to *unify*(v_1, v_2), $r_1 =$ *find*(v_1) and *find*$(r_2) = v_2$. By the induction hypothesis, $r_1 \sim v_1$ and $r_2 \sim v_2$ and $v_1 \sim v_2$. Hence the result follows. To show (3) the call is true the first time when *unbound?*(v) is true. For each subsequent call, the result is true by transitivity. The proof of (4) follows from (3) and the inductive hypothesis, when *find*$(u) \neq$ *find*(v). $\dashv$

Non-unifiability is characterized by the following special case:

Corollary 3.5.1 *Let G be a unification graph of a term equation $\tau_1 \stackrel{?}{=} \tau_2$ between the terms τ_1 and τ_2 represented by the term graphs rooted at vertices v_1 and v_2. Let the unification closure of G be denoted $\sim$. Then*

1. *If unify$(v_1, v_2) =$ fail$(CYCLE)$, then $G/{\sim} \models [u] \xrightarrow{+} [u]$ for some $u \in G$.*
2. *If unify $=$ fail$(CLASH)$, then $G/{\sim} \models [w] = [w']$ for some w, w' such that $L(w) \neq L(w')$.*

The above result captures the conditions for non-unifiability of a unification graph in terms of properties of the quotient of the unification graph. The operational details of deriving such a condition for non-unifiability is the theme of the next chapter.

3.6 Summary

This chapter formalized the problem of unification of rational terms. Rational terms were defined as co-inductive objects, and extensionality was defined via term-bisimulations.

The notion of unification of a system of term equations, unifiers, and most general unifier were defined. Unifiability reduces to computing the homogeneity of the unification closure of a unification graph. A simple unification algorithm that determines the unifiability of a pair of term equations was presented.

4

Path expression logics for unifiability

The last chapter showed how the unifiability of a system of equations is determined by the homogeneity and acyclicity of the quotient graph $G/\sim$ of the graph G representing the term equations. However, it left open the question of source-tracking: how to express unifiability directly in terms of connectivity relation on the source graph G, rather than the quotient graph.

This chapter uses the logic of path deductions for expressing connectivity relations on a labeled directed graph. Using the “formulas as types and proofs as programs” notion, also known as the Curry-Howard isomorphism (Howard [49]), paths can be viewed as encoding proofs of connectivity in a graph. Using this interpretation, the type of a path is its endpoints. The set of paths is thus a typed subset of the elements of the algebra of the untyped free monoid generated by the edges of a directed labeled graph. Each path also has an associated semantics, which is the label of the path. Source-tracking of unification is based on considering bidirectional paths over a labeled directed graphs G: that is paths in the graph $G \cup G^{-1}$, where G^{-1} is obtained by reversing the direction of edges in G. In other words, edges can be traversed either positively (from source to destination) or negatively, from destination to source. Unification closure may be viewed as a connectivity relation by interpreting the downward-closure rule as building a bidirectional path connecting vertices through their parents. Source-tracking unification involves computing paths witnessing connectivity in a labeled directed graph G, where positive and negative traversal is permitted, and relating them to paths witnessing connectivity in the quotient graph $G/\!\sim$, where only positive traversal is permitted.

Connectivity in the original graph does not guarantee connectivity in the quotient graph. That is determined by properties of the label semantics of the path witnessing the connectivity in original graph. Roughly speaking, an empty path in the quotient graph corresponds to a path in the original graph whose positive and negative traversals cancel out, while a non-empty path in the quotient graph corresponds to a path in the original graph that is left with a net positive traversal. The subset of paths in the original graph that can be thus related to paths in the quotient graph are called *unification paths*. If the labels on branch edges and their inverses are interpreted as closed and open parentheses, respectively, then the labels of unification paths are sentences in the context-free language of balanced parentheses (the semi-Dyck languages) and their suffixes.

Various classes of paths are expressed as "well-typed terms" drawn from a free algebra of terms over the edges of a labeled directed graph. The logic Π, introduced in Section 4.2, computes paths in a directed labeled graph. Π^2, computes bidirectional paths, while Π^U computes unification paths. In the logic P^U, path terms are drawn from the term algebra whose signature is that of a group. Unification path expressions may be simplified (normalized) to unification paths.

The logics P^U and Π^U may be viewed as an operational semantics of unification. The construction of P^U deductions is easily integrated into the unification algorithm with the addition of a proof parameter. The unification algorithm maintains the invariant that at each call of *unify*(u, v, p), p is the proof of the membership of the pair (u, v) in the unification closure over a graph G containing vertices u and v. The construction of these proofs incurs only a constant overhead per unification call.

The path expression logics P^U and Π^U are equivalent to the logic *LE*$_0$ of Le Chenadec [64]. The systems Π^U, P^U and *LE*$_0$ are all sound and complete with respect to the connectivity in the quotient graph. However, P^U and Π^U are logics about connectivity between vertices in a graph, rather than equality between terms. They make the connection with formal language path problems more explicit and suggest a more direct route to implementation with existing unification algorithms.

An important measure of the quality of source-tracking information is the absence of irrelevant details in that information. Simplification of source-tracking information is implemented by a convergent rewrite system operating on path expressions and paths. Subject reduction guarantees that simplification preserves the endpoint type and label semantics of the path.

Section 4.1 reviews the basic results about Σ algebras and Semi-Dyck sets, the basis for encoding unification proofs. Section 4.2 introduces the logics Π and Π^2. Section 4.3 presents the logic Π^U for unification paths. Section 4.4 introduces the logic P^U of unification path expressions. Section 4.5 shows how the computation of P^U proofs can be integrated into the unification algorithm.

4.1 Preliminaries

4.1.1 Σ-Algebras

In this section, we recapitulate basic results of the algebraic theory of data types: Σ-algebras, homomorphisms, initial, free and quotient algebras. The notation used here has been (partly) borrowed from (Baader and Nipkow [2]).

Given a signature Σ, a Σ-*algebra* $\mathcal{A}$ is a pair $\langle A, a \rangle$, where A is a set, and $a : \Sigma(A) \longrightarrow A$ is a function from $\Sigma(A)$ to A. The set A is called the *carrier of* $\mathcal{A}$. Thus, for each $f \in \Sigma$, if *arity*$(f) = n$, $a(f(a_1, \ldots, a_n))$ is an element of A. We can also think of $a(f)$ as a function from A^n to A. $a(f)$ is then called the *interpretation of* f *in* $\mathcal{A}$. Much of the discussion in the rest of the chapter relies on the *monoid* signature $\Sigma_{\mathbf{Mon}} = \{(\epsilon, 0), (\circ, 2)\}$, and *group* signature $\Sigma_{\mathbf{Gr}} = \Sigma_{\mathbf{Mon}} \cup \{(.)^{-1}, 1)\}$.

Let Σ be a signature. If $h : A \longrightarrow B$ is a function, then $\Sigma(h) : \Sigma(A) \longrightarrow \Sigma(B)$ denotes the function that maps each element $f(a_1, \ldots, a_n)$ of $\Sigma(A)$ to the element $f(h(a_1), \ldots, h(a_n))$. Given Σ-algebras $\mathcal{A} = \langle A, a\rangle$ and $\mathcal{B} = \langle B, b\rangle$, a function $h : A \longrightarrow B$ is a *homomorphism from $\mathcal{A}$ to $\mathcal{B}$* if $ha = b\Sigma(h)$. If Σ is a signature and V is a set of variables, $\mathcal{T}(\Sigma, V)$, the *Σ-term algebra generated by V* is the Σ-algebra whose carrier is $T^*(\Sigma, V)$, and for each $f \in \Sigma$ of arity n, $a(f, t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. Σ-term algebras are important because if $\mathcal{T}(\Sigma, V)$ is a Σ-term algebra and $\mathcal{B} = \langle B, b\rangle$ is any Σ-algebra, then for each function $h : V \longrightarrow B$, there is a unique homomorphism from $\mathcal{T}(\Sigma, V)$ to $\mathcal{B}$, called the *homomorphic extension of* h. The homomorphic extension of h is often indicated by h itself, but sometimes it is denoted $\hat{h}$ in order to distinguish it from h.

If $\mathcal{A} = \langle A, a\rangle$ is a Σ-algebra, an equivalence relation R on A is a *congruence*, if for each $f \in \Sigma$ such that *arity*$(f) = n$, and for each $a_i \; R \; a_i'$, $1 \leq i \leq n$, $f(a_1 \ldots, a_n) \; R \; f(a_1' \ldots, a_n')$. The quotient algebra $\mathcal{A}/R$ is the Σ-algebra $\langle A/R, a/R\rangle$ whose carrier set A/R is the set of equivalence classes $[u]_R = \{v \in A \mid u \; R \; v\}$ of R, and for each $f \in \Sigma$ with *arity*$(f) = n$, and for each $[a_i] \in A/R$, $1 \leq i \leq n$, $a/R(f([a_1]_R, \ldots, [a_n]_R))$ is defined to be $[a(f(a_1, \ldots, a_n))]_R$.

Given a signature Σ and a denumerable set V of variables, a Σ-identity is a pair $s \approx t$, where s, t are elements of $T^*(\Sigma, V)$. The set of Σ-identities of interest in this chapter are the *monoid identities* $\mathbf{Mon} = \{p \circ (q \circ r) \approx (p \circ q) \circ r, \epsilon \circ p \approx p, p \circ \epsilon \approx p\}$, and the *group* *identities* $\mathbf{Gr} = \{p \circ (q \circ r) \approx, (p \circ q) \circ r, p^{-1} \circ p \approx \epsilon, p \circ \epsilon \approx p\}$. A Σ-identity $s \approx t$ *holds in a* Σ*-algebra* $\mathcal{A}$, written $A \models s \approx t$, if for every homomorphism $h : \mathcal{T}(\Sigma, V) \longrightarrow \mathcal{A}$, $h(s) = h(t)$. A Σ-algebra $\mathcal{A}$ is a *model* for a set of Σ-identities E, written $A \models E$, if for each $s \approx t \in E$, $A \models s \approx t$. A *monoid* is a $\Sigma_{\mathbf{Mon}}$-algebra that models the set of identities $\mathbf{Mon}$. A *group* is a $\Sigma_{\mathbf{Gr}}$-algebra that models the set of identities $\mathbf{Gr}$. A pair $s \approx t$ is a *semantic consequence of* E, written $E \models s \approx t$, if for each $\mathcal{A}$ such that $\mathcal{A} \models E$, $\mathcal{A} \models s \approx t$. The set of all semantic consequences of E is called the *equational theory of* E and is denoted $\approx_E$. By Birkhoff's theorem, $\approx_E$ is equal to the least equivalence relation over $T^*(\Sigma, V)$ containing E that is closed under substitutions and congruences.

If X is a set, then $\mathcal{T}(\Sigma_{Mon}, X)/\approx_{\mathbf{Mon}}$, abbreviated $\mathbf{Mon}(X)$, is called the *free monoid generated by* X. It is easy to verify that $\mathbf{Mon}(X)$ is a monoid. Similarly, $\mathcal{T}(\Sigma_{Gr}, X)/\approx_{\mathbf{Gr}}$, abbreviated $\mathbf{Gr}(X)$, is the *free group generated by* X. It is easy to see that the carrier of $\mathbf{Mon}(X)$ is isomorphic to the set X^* of sentences over X.

4.1.2 Semi-Dyck and Dyck languages

The class of languages identified by balanced parentheses, known as the Semi-Dyck sets, are important from both practical and theoretical standpoint in computer science. Because of its uniformity and ease of parsing, notation based on nested, balanced parentheses has proved ideal for representing textual data and also programs.

Semi-Dyck sets also play an important role in the theory of formal languages, where they are known to characterize the full class of context-free languages [19] from which the class of recursively enumerable sets may be characterized (see for example, Savitch [94]).

The main result of this chapter is concerned with showing how unification proofs can be encoded as paths whose labels are sentences of a Semi-Dyck set. In this section, we briefly review the definitions of Semi-Dyck and Dyck sets and related languages. We discuss their grammars, cancellative properties, and their connection with groups. Semi-Dyck and Dyck languages are treated in detail by Harrison [42].

Let $\Sigma = \{b_1, \ldots, b_n\}$ be an alphabet. Then the alphabet $\Sigma^{-1} = \{b_i^{-1} \mid b_i \in \Sigma\}$, assumed disjoint from Σ, consists of matching symbols such that for each $b_i \in \Sigma$, there is a unique matching symbol $b_i^{-1} \in \Sigma^{-1}$. The symbols in Σ are called *closed* Σ*-parenthesis symbols*, and those in Σ^{-1} are called *open* Σ*-parenthesis symbols*. $\mathbf{D}(\Sigma)$ denotes the set $\{b^{-1}b \approx \epsilon \,|b \in \Sigma\}$ consisting of one-way cancellative identities. $\mathbf{D}'(\Sigma)$ denotes the set $\{bb^{-1} \approx \epsilon, b^{-1}b \approx \epsilon, b \in \Sigma\}$ consisting of left and right cancellative identities. When Σ is clear from the context, we abbreviate these sets of identities as $\mathbf{D}$ and $\mathbf{D}'$, respectively. Let $\approx_{\mathbf{D}}$ and $\approx_{\mathbf{D}'}$ denote the smallest congruences over $(\Sigma \cup \Sigma^{-1})^*$ containing the identities $\mathbf{D}$ and $\mathbf{D}'$, respectively. Intuitively, $u \approx_{\mathbf{D}} v$ if u may be obtained from v via applications of the identities in $\mathbf{D}$ (or vice versa), and likewise for $u \approx_{\mathbf{D}'} v$. The equational systems $\mathbf{D}$ and $\mathbf{D}'$ may be turned into rewrite systems by orienting each of the identities $b^{-1}b \approx \epsilon$ and $bb^{-1} \approx \epsilon$ as $b^{-1}b \longrightarrow \epsilon$ and $bb^{-1} \longrightarrow \epsilon$, respectively. Thus, the rewrite system $\mathbf{D}(\Sigma)$

consists of the rules

$$\{b^{-1}b \longrightarrow \epsilon \mid b \in \Sigma\}$$

and the rewrite system $\mathbf{D}'(\boldsymbol{\Sigma})$ is equal to $\mathbf{D}(\Sigma) \cup \{bb^{-1} \longrightarrow \epsilon \mid b \in \Sigma\}$. It is easy to show that the rewrite systems $\mathbf{D}$ and $\mathbf{D}'$ are convergent (terminating and confluent) and complete with respect to the corresponding equational systems $\mathbf{D}$ and $\mathbf{D}'$. We denote by $\mu_{\mathbf{D}(\Sigma)}(x)$ and $\mu_{\mathbf{D}'(\Sigma)}(x)$, the (unique) normal forms under $\mathbf{D}(\Sigma)$ and $\mathbf{D}'(\Sigma)$ rewriting, respectively, of a sentence $x \in (\Sigma \cup \Sigma^{-1})^*$. For example, if $\Sigma = \{]\}$, and $\Sigma^{-1} = \{[\}$, then $\mu_{\mathbf{D}(\Sigma)}(][) =][$, but $\mu_{\mathbf{D}'(\Sigma)}(][) = \epsilon$. The normal form of a sentence p may be computed in a single left-to-right pass of p in time $O(|p|)$.

Given $L \subseteq (\Sigma \cup \Sigma^{-1})^*$, let

$$\begin{aligned} D(\Sigma, L) &\stackrel{\text{def}}{=} \{l \in (\Sigma \cup \Sigma^{-1})^* \mid \mu(l) \in L\} \\ D'(\Sigma, L) &\stackrel{\text{def}}{=} \{l \in (\Sigma \cup \Sigma^{-1})^* \mid \mu'(l) \in L\} \end{aligned}$$

We are primarily interested in the languages $D(\Sigma, L)$, when L is $\Sigma^0 = \{\epsilon\}$, Σ^+, or Σ^*. These are abbreviated $D^0(\Sigma)$, $D^+(\Sigma)$, and $D^*(\Sigma)$, respectively. When Σ is clear from context, these are further abbreviated to D^0, D^* and D^+. The language $D^0(\Sigma)$ is known as the *semi-Dyck set over* Σ, and the language $D'(\Sigma, \Sigma^0)$, abbreviated D'^0 is known as the *Dyck set over* Σ.

D^0 and D^+ are mutually disjoint, and $D^* = D^0 \cup D^+$. It can be seen that D^* is the set of all suffixes of D^0 sentences, and is suffix-closed. (See Harrison [42], page 314, for the symmetric case of prefixes of D^0 sentences.) Thus D^+ may be informally characterized as the set of "unbalanced" suffixes of sentences of balanced parentheses. The languages D^0, D^+, and D^* are generated using the following context-free grammar:

$$\begin{array}{lcll} D^0 & ::= & \epsilon \mid D^0 b^{-1} D^0 b D^0 & b \in \Sigma \\ D^+ & ::= & D^* b D^* & b \in \Sigma \\ D^* & ::= & D^0 \mid D^+ & \end{array}$$

Example 4.1.11: If $\Sigma = \{),]\}$ and $\Sigma^{-1} = \{(,[\}$, then the set of $\mathbf{D}(\Sigma)$ consists of the identities $(\,) = \epsilon$ and $[\,] = \epsilon$. The sentences '([ ])', and '(( )[ ])' are in D^0. '( ) [ ]]', '( )]' are in D^+ and they are both reduce to $]$ by the rewrite system $\mathbf{D}(\Sigma)$. The sentence '( ])' is not in D^*. $\diamond$

The next lemma formalizes the intuition that for every D^* sentence l, every open parenthesis occurring in l is matched, and every closed parenthesis is either matched, or unmatched. In the latter case, l is in D^+.

Lemma 4.1.1 *(Decomposition)*

Let Σ be an alphabet. If $l \in D^*(\Sigma)$, then

1. *If $l = p\delta^{-1}q$ for some $\delta \in \Sigma$, then $q = x\delta y$, where $x \in D^0$*

2. *If $l = p\delta q$, for some $\delta \in \Sigma$, then*

   (a) $p = x\delta^{-1}y$, *where* $y \in D^0$, *or*

   (b) $p, q \in D^*$, *implying* $l \in D^+$

Proof By induction on the derivation of D^* sentences. $\dashv$

We now consider another example of using cancellative identities to reduce D^* sentences.

Example 4.1.12: Consider the sentence l equal to $(_1)_2(_3)_4$ consisting of left and right parentheses symbols indexed for identification. If $\Sigma = \{(\}$, and $\Sigma^{-1} = \{)\}$, then l is in $D^0(\Sigma)$. l can be reduced to ϵ using the one-sided cancellative rule $() \longrightarrow \epsilon$ with $(_1$ and $)_2$ cancelling each other, and similarly $(_3$ and $)_4$. However, l may be reduced to ϵ using two-sided cancellative rules $\{() \longrightarrow \epsilon,)(\longrightarrow \epsilon\}$ as well: $)_2$ and $(_3$ cancel each other, leaving the sentence $(_1)_4$ which reduces to ϵ using the rule $() \longrightarrow \epsilon$. $\diamond$

The above example hints that D^*-sentences are closed under two-sided cancellation. This means that when reducing D^*-sentences, two-sided cancellative identities may be employed, rather than one-sided cancellative identities.

Lemma 4.1.2 *(Subject reduction of D^*-sentences under $\mathbf{D}'$ rewriting)*

Let Σ be an alphabet. If $l \in D^*(\Sigma)$ and $l \longrightarrow_{\mathbf{D}'} l'$, then $l \approx_{\mathbf{D}} l'$.

Proof By a case analysis on the rules of $\mathbf{D}'$. Let $\delta \in \Sigma$.

1. $\delta^{-1}\delta \longrightarrow \epsilon$: $l = r\delta^{-1}\delta s$ and $l' = rs$ for some $r, s \in (\Sigma \cup \Sigma^{-1})^*$. Since the redex $\delta^{-1}\delta$ is being reduced to ϵ, and $\delta^{-1}\delta \approx \epsilon \in \mathbf{D}(\Sigma)$, it follows that $l \approx_{\mathbf{D}(\Sigma)} l'$.

2. $\delta\delta^{-1} \longrightarrow \epsilon$: $l = r\delta\delta^{-1}s$ and $l' = rs$, for some $\delta \in \Sigma$ and $r, s \in (\Sigma \cup \Sigma^{-1})^*$. By Lemma 4.1.1, $s = s_1\delta s_2$, where $s_1 \in D^0$. Again, since δ occurs in l, there are two cases:

   (a) $r = r_1\delta^{-1}r_2$, where $r_2 \in D^0$: Therefore,

$$
\begin{array}{lcll}
l & = & r\delta\delta^{-1}s & \text{Given} \\
 & = & r_1\delta^{-1}r_2\delta\delta^{-1}s_1\delta s_2 & \text{Lemma 4.1.1} \\
 & \xrightarrow{*}_{\mathbf{D}} & r_1\epsilon\epsilon s_2 & \text{Since } r_2, s_1 \in D^0 \\
 & \xrightarrow{*}_{\mathbf{D}} & r_1 s_2 & \text{and} \\
l' & = & rs & \text{Given} \\
 & = & r_1\delta^{-1}r_2 s_1\delta s_2 & \text{Since } r = r_1\delta^{-1}r_2,\ s = s_1\delta s_2 \\
 & \xrightarrow{*}_{\mathbf{D}} & r_1\epsilon s_2 & \text{Since } r_2, s_1 \in D^0 \\
 & \xrightarrow{*}_{\mathbf{D}} & r_1 s_2 &
\end{array}
$$

   Hence, we have, $l \approx_{\mathbf{D}(\Sigma)} r_1 s_2 \approx_{\mathbf{D}(\Sigma)} l'$,

   (b) $r \in D^*(\Sigma)$ and $\delta s \in D^*$. Therefore, $l = r\delta\delta^{-1}s_1\delta s_2$. Since $s_1 \in D^0$, $l \xrightarrow{*}_{\mathbf{D}} r\delta s_2$. Also, $l' = rs_1\delta s_2$. Since $s_1 \in D^0$, $l' \xrightarrow{*}_{\mathbf{D}} r\delta s_2$. Hence both l and l'

reduce to $r\delta s_2$, implying $l \approx_{\mathbf{D}(\Sigma)} r\delta s_2 \approx_{\mathbf{D}} l'$.

$\dashv$

Dyck sets are closely related to groups. The quotient monoid $(\Sigma \cup \Sigma^{-1})^*/ \approx_{\mathbf{D'}\Sigma}$ has $D'^0(\Sigma)$ as its identity element. By defining a unary inverse operation $[w]^{-1} = [w']$, where $ww' \approx_{\mathbf{D'}(\Sigma)} \epsilon$, the quotient monoid can be made into a group. This group is isomorphic to the free group generated by Σ. The isomorphism f maps each class $[w]$ of $\approx_{\mathbf{D'}}$ containing w to the class of the free group containing w with the property that $[w] \subseteq f([w])$.

It is natural to define an *inverse* operation *inv* directly on sentences as follows: *inv*$(\epsilon) = \epsilon$, *inv*$(c) = c^{-1}$, for $c \in \Sigma$, *inv*$(c^{-1}) = c$, for $c \in \Sigma^{-1}$, and for each p, q that are sentences over $\Sigma \cup \Sigma^{-1}$, *inv*$(pq) =$ *inv*(q)*inv*(p). The free monoid $(\Sigma \cup \Sigma^{-1})^*$, along with *inv*, is a $\Sigma_{\mathbf{Gr}}$-algebra. The inverse operation can be modeled by two-sided cancellation: $q =$ *inv*(p) implies $qp \approx_{\mathbf{D'}} \epsilon$.

4.1.3 Σ-graph representation of unification graphs

In the last chapter, we assumed unification graphs were represented by a union of term graphs and undirected equational edges. In order to represent a unification graph as a Σ-graph, we simply orient each equational edge in an arbitrary direction and assign the label ϵ to it. Formally, a Σ-term unification graph $G = \langle X, W, b, E\rangle$ is represented as a Σ-graph $\langle V, L, D\rangle$ in which $V = X \cup W$, $L(x) = \emptyset$ if $x \in X$, and $L(w) = \{$*hd*$(b(w))\}$, and

$D = \{w \xrightarrow{f.i} v \mid f \in L(w) \text{ and } b(w)(i) = u\} \cup \{u \xrightarrow{\epsilon} v \mid (u, v) \in E\}$. Therefore, the Σ-graph $\langle V, L, D\rangle$ of G is constructed by augmenting the Σ-term graph $\langle X, W, b\rangle$ with a set of directed equational edges: for each pair of vertices in $(u, v) \in E$, there is an edge $u \xrightarrow{\epsilon} v$. The notation $c : u \xrightarrow{\delta} v \in D$ is shorthand used for the statement "the edge $u \xrightarrow{\delta} v$, abbreviated c, is in D."

Thus, labeled directed graphs generalize both term graphs and unification graphs. Unification closure can be defined directly on labeled directed graphs. Let $G = \langle \Sigma_V, \Sigma_D, V, L, D\rangle$ be a labeled directed graph. The set of *ϵ-edges of G* is the set $\{(u, v) \in D \mid u \xrightarrow{\epsilon} v \in D\}$. The *unification closure of a labeled directed graph* is the smallest downward-closed equivalence relation on the vertices V containing the ϵ-edges of G.

Example 4.1.13: The unification graph of Example 3.4.7 is represented by the Σ-graph $G = \langle V, L, D\rangle$, where $\Sigma = \langle \mathsf{f} \mapsto 2, \mathsf{g} \mapsto 1\rangle$. The graph is shown in Figure 4.1. The equational edges a_1, a_2 and a_3 are labeled ϵ. The branch edges b_1 and b_4 are labeled $\mathsf{f}.1$, while the branch edges b_3 and b_6 are labeled $\mathsf{f}.2$. The branch edges are b_2 and b_5 are labeled $\mathsf{g}.1$. ◇

Directed equational edges are distinguished by thin, transparent arrowheads and are implicitly labeled ϵ, while branch edges are identified with thicker, solid arrowheads.

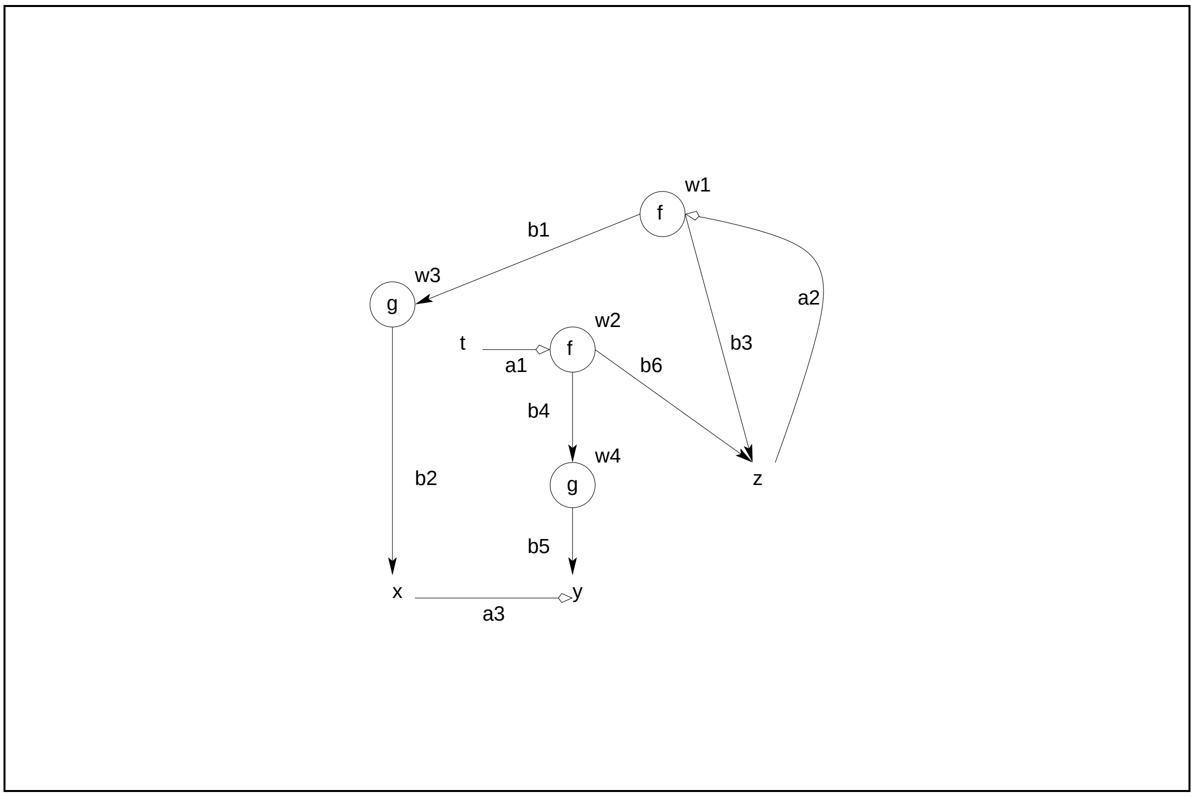


Figure 4.1: Unification graph G of Example 3.4.7 represented as a directed, labeled graph

4.2 Path Logics

In this section, we identify various types of connectivity relations on the vertices of labeled directed graphs. These relations are expressed as a "type system" of path terms. The first of these logics, called Π, identifies labeled paths over a directed graph G. The second, Π^2, identifies bidirectional paths over a directed graph, which are paths over $G \cup G^{-1}$, where G^{-1}, the inverse of G is obtained by reversing the orientation of the edges of G. The third, Π^U, identifies *unification paths*, which are labeled paths over G that encode proofs of membership in the unification closure of a unification graph.

4.2.1 Paths over labeled directed graphs

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph. The set of labeled paths in G may be identified as those sentences over D obtained from deductions in the logic of a type system $\Pi(G)$ shown in Figure 4.2. The rules of Π are directly derived from the inductive definition of a path. Judgements in $\Pi(G)$ are of the form $\vdash p : u \xrightarrow{l} v$. Here, G is a labeled directed graph, p is a sentence over D, u and v are vertices in G, and l is a sentence over Σ_D. We write $\vdash_{\Pi(G)} p : u \xrightarrow{l} v$ to denote that there is a deduction $\mathcal{D}$ in the logic Π whose final step is the judgement $p : u \xrightarrow{l} v$. The logic Π is sound and complete with respect to labeled paths in a labeled directed graph:

$$\text{INIT} \qquad \frac{}{u \xrightarrow{\delta} v : u \xrightarrow{\delta} v} \qquad\qquad u \xrightarrow{\delta} v \in D$$

$$\text{REF} \qquad \frac{}{\epsilon : u \xrightarrow{\epsilon} u}$$

$$\text{TRANS} \qquad \frac{p : u \xrightarrow{l} v' \qquad q : v' \xrightarrow{l'} v}{pq : u \xrightarrow{ll'} v}$$

Figure 4.2: The logic $\Pi(G)$ of paths over G where G is the labeled directed graph $\langle \Sigma_V, \Sigma_D, V, L, D \rangle$

Lemma 4.2.1 *If G is a labeled directed graph, then $\vdash_{\Pi(G)} p : u \xrightarrow{l'} v$ if and only if $G \models p : u \xrightarrow{*} v$ and $l(p) = l'$.*

Proof Straightforward induction on the definition of labeled paths and Π deductions. $\dashv$

Note that because an edge in a labeled directed graph may be labeled with ϵ, non-empty paths may have empty labels. The following properties are all straightforward to prove by induction on the length of paths:

Lemma 4.2.2 *(Labeled Path decomposition)*

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph, and let p be a path in G from u to v labeled l. Then

1. $|l| \leq |p|$.

2. *If* $l_1 l_2 = l$*, there is a vertex* v' *paths* q *and* r *in* G*, such that* $G \models q : u \xrightarrow{*} v'$*,* $l(q) = l_1$*,* $G \models r : v' \xrightarrow{*} v$*,* $l(r) = l_2$ *and* $p = qr$*.*

3. *If* $l = x\delta y$*, where* $\delta \in \Sigma_D$*, then there are vertices* u' *and* v' *in* V *and an edge* c *from* u' *to* v' *labeled* δ*, and paths* q *and* r *such that* $G \models q : u \xrightarrow{*} u'$*,* $l(q) = x$*,* $G \models r : v' \xrightarrow{*} v$*,* $l(q) = y$*, and* $p = qcr$*.*

Proof By induction on p. $\dashv$

4.2.2 Bidirectional Paths

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph. The *inverse of* G, G^{-1}, is the labeled directed graph $\langle \Sigma_V, \Sigma_D^{-1}, V, L, D^{-1} \rangle$, where $D^{-1} = \{v \xrightarrow{\delta^{-1}} u \mid u \xrightarrow{\delta} v \in D\}$. We define $\epsilon^{-1} = \epsilon$. Thus, G^{-1} is obtained from G by reversing the direction of each edge in G, changing each edge label δ to δ^{-1}, and leaving the label ϵ unchanged. $G \cup G^{-1}$ is the labeled directed graph $\langle \Sigma_V, \Sigma_D \cup \Sigma_D^{-1}, L, V, D \cup D^{-1} \rangle$. The set of *bidirectional labeled paths over* G is the set of labeled paths over $G \cup G^{-1}$. The set of bidirectional paths are identified by the deductions in the logic Π^2 of Figure 4.3.

It is easy to see that if $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ is a labeled directed graph, then p is a bidirectional path in G from u to v if and only if *inv*(p) is a bidirectional path in G from v to u. Furthermore $l(\textit{inv}(p)) = \textit{inv}(l(p))$.

INIT	$\dfrac{}{u \xrightarrow{\delta} v : u \xrightarrow{\delta} v}$	$u \xrightarrow{\delta} v \in G$
REF	$\dfrac{}{\epsilon : u \xrightarrow{\epsilon} u}$	$u \in G$
SYM	$\dfrac{}{(u \xrightarrow{\delta} v)^{-1} : v \xrightarrow{\delta^{-1}} u}$	$u \xrightarrow{\delta} v \in G$
TRANS	$\dfrac{p : u \xrightarrow{l} v' \qquad q : v' \xrightarrow{l'} v}{pq : u \xrightarrow{ll'} v}$	

Figure 4.3: The logic $\Pi^2(G)$ of bidirectional paths over G where G is the labeled directed graph $\langle \Sigma_V, \Sigma_D, V, L, D \rangle$

Lemma 4.2.3 *Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph.*

1. *(Π^2 soundness and completeness with respect to Π)*

   $\vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$ *if and only if* $\vdash_{\Pi(G \cup G^{-1})} p : u \xrightarrow{l} v$.

2. *(Inverse)* $\vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$ *if and only if* $\vdash_{\Pi^2(G)}$ *inv*$(p) : v \xrightarrow{l'} u$*, where* $l' =$ *inv*(l).

3. *(Subject Reduction)* $G \vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$ *and* $p \longrightarrow_{\mathbf{D'}(D)} q$ *implies* $\vdash_{\Pi^2(G)} q : u \xrightarrow{l'} v$ *for some* l *such that* $l \longrightarrow_{\mathbf{D'}(\Sigma_D)} l'$.

Proof The proof of (1) is by straightforward induction on Π deductions for the if direction, and induction on Π^2 deductions for the only if direction. The proofs of (2) and (3) are by induction on Π^2 deductions. $\dashv$

Among the set of bidirectional paths over a labeled directed graph, we identify a subset of paths known as unification paths. Formally, if $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ is a labeled directed graph, then the set of *unification paths in G from u to v* is the set of all those bidirectional paths in G from u to v whose label is in $D^*(\Sigma_D)$.

Example 4.2.14: The unification graph G of Figure 4.1 is a labeled directed graph whose set of edge labels is equal to $\Sigma_{\mathbf{N}}$, where $\Sigma = \{\mathsf{f} \mapsto 2, \mathsf{g} \mapsto 1\}$. G^{-1} is obtained by inverting the orientation of each edge of G. $b_1^{-1} a_2^{-1}$ is a bidirectional path from w_3 to z. It is not a unification path because its label f_1^{-1} is not in $D^*(\Sigma_{\mathbf{N}})$. The path $b_5 a_3^{-1} b_2^{-1}$ labeled $\mathsf{g}_1 \mathsf{g}_1^{-1}$ from w_4 to w_3 is also not a unification path. However, the path $a_1 b_4$ labeled g_1 from t to w_4 is a unification path.

The quotient graph of the unification closure is the labeled directed graph of Figure 3.4.

$\diamond$

We are now ready to state the first part of one of the main results of this chapter: Existence of a unification path from a vertex u to v in a labeled directed graph guarantees connectivity from u to v in the quotient graph $G/\!\sim$ of G with respect to the unification closure $\sim$. Conversely, for every path from u to v in $G/\!\sim$, there is a unification path from u to v in G. These results, stated below in Lemma 4.2.4 and Lemma 4.2.6, adequately characterize connectivity in the $\sim$-quotient graph $G/\!\sim$ in terms of connectivity in G. This characterization forms the basis for deriving source-tracking for unification because we can track paths in the quotient graph in terms of their "source" paths in the original graph, which are unification paths.

Lemma 4.2.4 *(Soundness of unification paths)*

Let G be a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, L, V, D\rangle$ whose unification closure is $\sim$. Then $\vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$, and $l \in D^*(\Sigma_D)$ implies $G/\sim \models [u]_\sim \xrightarrow{l'} [v]_\sim$, where $l' = \mu_{\mathbf{D}(\Sigma_D)}(l)$.

Proof By induction on the derivation of l in the grammar $\mathbf{D}^*(\Sigma_D)$. Let $[u]$ denote the equivalence class under $\sim$ containing u. For each case, we construct a deduction $\vdash_{\Pi(G/\sim)} p' : [u] \xrightarrow{l'} [v]$.

1. $l = \epsilon$: We need to show $\vdash_{\Pi(G/\sim)} p' : [u]_\sim \xrightarrow{\epsilon} [v]_\sim$. The proof is by induction on Π^2 sentences. There are four cases depending on the rule used to construct p. In each case, we show $[u] = [v]$, from which, using $\mathrm{REF}_{\Pi(G/\sim)}$, we can construct the Π deduction $\vdash_{\Pi(G/\sim)} \epsilon : [u] \xrightarrow{\epsilon} [v]$.

   (a) $\mathrm{INIT}_{\Pi^2(G)}$: $\vdash_{\Pi^2(G)} p : u \xrightarrow{\epsilon} v$. This implies $u \xrightarrow{\epsilon} v$ is an edge in G. Hence, by the definition of $\sim$, $[u] = [v]$.

   (b) $\mathrm{REF}_{\Pi^2(G)}$: This implies $u = v$ and therefore $[u] = [v]$.

   (c) $\mathrm{SYM}_{\Pi^2(G)}$: $p : u \xrightarrow{\epsilon} v$. Then $v \xrightarrow{\epsilon} u \in D$ and therefore by the definition of $\sim$, $[u] = [v]$.

(d) TRANS$_{\Pi^2}$: $p : u \xrightarrow{\epsilon} v$. Then $\vdash_{\Pi^2(G)} r : u \xrightarrow{l_1} v'$ and $\vdash_{\Pi^2(G)} s : v' \xrightarrow{l_2} v$ such that $p = qr$ and $\epsilon = l_1 l_2$. l_1 and l_2 are equal to ϵ. By induction it follows that there are deductions $\vdash_{\Pi(G/\sim)} r' : [u] \xrightarrow{\epsilon} [v']$ and $\vdash_{\Pi(G/\sim)} s' : [v'] \xrightarrow{\epsilon} [v]$. This implies $[u] = [v'] = [v]$.

2. $l = x\delta^{-1}y\delta z$, where $\delta \in \Sigma_D$, and $x, y, z \in D^0(\Sigma_D)$. Using Lemma 4.2.2, it is easy to show that there are vertices u_1, u_2, u_3, u_4, edges c_1, c_2, and paths p_1,,p_3, and p_5, such that $p = p_1 c_1^{-1} p_3 c_4 p_5$, and

$$\vdash_{\Pi^2(G)} p_1 : u \xrightarrow{x} u_1$$

$$c_1 : u_2 \xrightarrow{\delta} u_1 \in D$$

$$\vdash_{\Pi^2(G)} p_3 : u_2 \xrightarrow{y} u_3$$

$$c_4 : u_3 \xrightarrow{\delta} u_4 \in D$$

$$\vdash_{\Pi^2(G)} p_5 : u_4 \xrightarrow{z} v$$

Applying the induction hypothesis, $[u] = [u_1]$, $[u_2] = [u_3]$, and $[u_4] = [v]$. Also, by the downward closure (DN) of $\sim$, $[u_1] = [u_4]$, implying that $[u] = [v]$. From this, it follows that $G\sim \models [u] \xrightarrow{\epsilon} [v]$.

3. $l = x\delta y$, where $\delta \in \Sigma_D$, and $x, y \in D^*(\Sigma_D)$. This proof is similar to the previous case, and is omitted.

$\dashv$

The next lemma shows that membership in the unification closure $\sim$ of a labeled directed graph G is witnessed by a unification path labeled by a semi-Dyck sentence over the edge labels of G.

Lemma 4.2.5 *(Completeness of semi-Dyck-labeled unification paths)*

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph with unification closure $\sim$. If $G \models u \sim v$, then there is a deduction $\vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$ such that $l \in D^0(\Sigma_D)$.

Proof By induction on $\sim$. There are five cases. In each case, we construct a proof such that $l \in D^0(\Sigma_D)$.

1. REF$_\sim$: $u = v$, $\vdash_{\Pi^2(G)} \epsilon : u \xrightarrow{\epsilon} u$ by the REF$_{\Pi^2}$ rule, and $\epsilon \in D^0(\Sigma_D)$.

2. INIT$_\sim$: Let $u \xrightarrow{\epsilon} v \in D$, then $\vdash_{\Pi^2(G)} u \xrightarrow{\epsilon} v : u \xrightarrow{\epsilon} v$ by the INIT$_{\Pi^2}$ rule, and $l(u \xrightarrow{\epsilon} v) = \epsilon$.

3. SYM$_\sim$: Let $G \models v \sim u$. By the induction hypothesis, $\vdash_{\Pi^2(G)} q : v \xrightarrow{l} u$ for some $l \in D^0(\Sigma_D)$. *inv*$(l) = l$ since $l \in D^0(\Sigma_D)$. By Lemma 4.2.3 (2), $\vdash_{\Pi^2(G)}$ *inv*$(q) : u \xrightarrow{l} v$.

4. TRANS$_\sim$: Let $u \sim v'$, $v' \sim v$, for some $v' \in V$. By the induction hypothesis, $\vdash_{\Pi^2(G)} q : u \xrightarrow{l'} v'$ and $\vdash_{\Pi^2(G)} r : v' \xrightarrow{l'} v$ for some bidirectional paths q, r in G such that $l, l' \in D^0(\Sigma_D)$. Then, by an application of the TRANS$_{\Pi^2}$ rule, $\vdash_{\Pi^2(G)} qr : u \xrightarrow{ll'} v$. Clearly, $ll' \in D^0(\Sigma_D)$.

5. DN$_\sim$: $u \sim v$ because there are vertices u', v' in V, and edges $u' \xrightarrow{\delta} u$ (abbreviated c) and $v' \xrightarrow{\delta} v$ (abbreviated c') in D such that $u' \sim v'$. Then

$\vdash_{\Pi^2(G)} c : u' \xrightarrow{\delta} u$	By INIT$_{\Pi^2}$	1
$\vdash_{\Pi^2(G)} c^{-1} : u \xrightarrow{\delta^{-1}} u'$	By SYM$_{\Pi^2}$	2
$\vdash_{\Pi^2(G)} q : u' \xrightarrow{l} v'$	By the ind. hyp.	3
$\vdash_{\Pi^2(G)} c^{-1}q : u \xrightarrow{\delta^{-1}l} v'$	By 2, 3 and TRANS$_{\Pi^2}$	4
$\vdash_{\Pi^2(G)} c' : v' \xrightarrow{\delta} v$	By INIT$_{\Pi^2}$	5
$\vdash_{\Pi^2(G)} c^{-1}qc' : u \xrightarrow{\delta^{-1}l\delta} v$	By 4, 5 and TRANS$_{\Pi^2}$	6

By the induction hypothesis in Step 3, $l \in D^0(\Sigma_D)$. Hence $\delta^{-1}l\delta \in D^0(\Sigma_D)$.

$\dashv$

Unification paths are complete with respect to connectivity in the $\sim$-quotient graph of a directed labeled graph.

Lemma 4.2.6 *(Unification path completeness)*

Let G be a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, L, V, D \rangle$. Let $\sim$ denote the unification closure of G. If $G/\!\sim\, \models [u]_\sim \xrightarrow{l'} [v]_\sim$, then there is a deduction $\vdash_{\Pi^2(G)} p : u \xrightarrow{l} v$ such that $l \approx_{\mathbf{D}} l'$.

Proof The proof is by induction on $\vdash_{\Pi(G/\sim)} [u] \xrightarrow{l'} [v]$. By Lemma 4.2.1, it is sufficient if we construct a deduction $\vdash_{\Pi(G/\sim)} [u] \xrightarrow{l'} [v]$ for each case:

1. REF$_\Pi$: $\vdash_{\Pi(G/\sim)} [u] \xrightarrow{l'} [v]$: Then $l' = \epsilon$, $[u] = [v]$. The result follows from Lemma 4.2.5.

2. INIT$_\Pi$: $\vdash_{\Pi(G/\sim)} c : [u] \xrightarrow{\delta} [v]$: Then $[u] \xrightarrow{\delta} [v]$ is an edge in $G/\sim$ and $\delta \in \Sigma_D$. Then, by the definition of a quotient graph, $u' \xrightarrow{\delta} v' \in D$ (abbreviated c) for some $u' \in [u]$ and $v' \in [v]$. Therefore

$\vdash_{\Pi^2(G)} q : u \xrightarrow{l} u'$	By Lemma 4.2.5, since $[u] = [u']$	1
$\vdash_{\Pi^2(G)} c : u' \xrightarrow{\delta} v'$	By INIT$_{\Pi^2}$, since $u' \xrightarrow{\delta} v' \in D$	2
$\vdash_{\Pi^2(G)} qc : u \xrightarrow{l\delta} v'$	From 1 and 2 using TRANS$_{\Pi^2}$	3
$\vdash_{\Pi^2(G)} r : v' \xrightarrow{m} v$	By Lemma 4.2.5, since $[v] = [v']$	4
$\vdash_{\Pi^2(G)} qcr : u \xrightarrow{l\delta m} v'$	From 3 and 4 using TRANS$_{\Pi^2}$	5

In steps (1) and (4), $l, m \in D^0(\Sigma_D)$ by the induction hypothesis.

Hence $l\delta m \approx_{\mathbf{D}(\Sigma_D)} \delta$.

3. TRANS$_\Pi$: Straightforward application of inductive hypothesis. Omitted.

$\dashv$

4.2.3 Computation of shortest unification paths

We have seen how unification source-tracking information may be encoded as unification paths. One measure of the succinctness of this information is the length of the unification paths encoding this information. Since unification paths are paths over a graph whose labels are constrained by the context-free grammar $D^*(\Sigma)$ for an alphabet Σ, shortest unification paths may be computed using the dynamic-programming-based context-free-grammar shortest paths algorithms of Barrett et al. [5]. If G is a directed labeled graph and $\mathcal{L}$ is a context-free language, then the context-free-grammar shortest path problem is the problem of computing the shortest path from the set of all paths p in G between a given source and destination vertex such that the label of p is a sentence in $\mathcal{L}$. If $\mathcal{L}$ is specified by a context-free grammar in Chomsky Normal Form (in which each production is of the form $A \to BC$ or $A \to a$ for non-terminals A, B and C, and terminal a), then the algorithm of Barrett et al. has time complexity $O(|V|^5|N|^2|R|)$ where V is the vertices in G , N is the set of non-terminals, and R the set of productions of the grammar.[1] The efficiency may be improved to $O(|V|^3|N||R|)$ using Fibonnaci heaps.

For unification paths over a directed labeled graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$, the grammar $D^*(\Sigma_D)$ is of size $O(\Sigma_D)$ and may be transformed into a grammar in Chomsky Normal Form whose set of non-terminals and productions is each of size $O(\Sigma_D)$. Thus, shortest

[1] The algorithm is unaffected if ϵ productions of the form $A \to \epsilon$ are included in the grammar [66].

unification paths can be computed in $O(|V|^5|\Sigma_D|^3)$ time, or $O(|V|^3|\Sigma_D|^2)$ time using Fibonnaci heaps.

4.3 Logic of unification paths

Unification paths may be constructed as "typed" path terms using the rules Π^U specified in Figure 4.4. Judgements in Π^U are of the form $p : u \xrightarrow{l} v$, where G is a labeled directed graph $\langle \Sigma_V, \Sigma_D, V, L, D\rangle$, p is a sentence over $D \cup D^{-1}$, u, v are vertices in V, and $l \in \Sigma_D^*$. The judgement $p : u \xrightarrow{l} v$ asserts that p is a unification path in G whose label reduces to l under $\mathbf{D}(\Sigma_D)$ rewriting.

Theorem 4.3.1 *(Soundness and Completeness of Π^U with respect to unification closure)*

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D\rangle$ be a labeled directed graph. Let u, v be vertices in G. Then

1. *(Soundness) If $\vdash_{\Pi^U(G)} p : u \xrightarrow{l} v$, then $\vdash_{\Pi^2(G)} p : u \xrightarrow{l'} v$, $l \in \Sigma_D^*$, and $l = \mu_{\mathbf{D}(\Sigma_D)}(l')$.*

2. *(Completeness) If $\vdash_{\Pi^2(G)} p : u \xrightarrow{l'} v$ such that $l' \in D^*(\Sigma_D)$, then $\vdash_{\Pi^U(G)} p : u \xrightarrow{l} v$, such that $l \in \Sigma_D^*$, and $l = \mu_{\mathbf{D}(\Sigma_D)}(l')$.*

Proof Soundness is by induction on Π^U deductions. Completeness is by induction on the derivation of l' using the rules of the grammar $D^*(\Sigma_D)$. $\dashv$

Paths computed by Π^U deductions are closed under $\mathbf{D}'$ rewriting.

INIT	$\dfrac{}{c : u \xrightarrow{\delta} v}$	$c : u \xrightarrow{\delta} v \in G$
REF	$\dfrac{}{\epsilon : u \xrightarrow{\epsilon} u}$	$u \in G$
SYM	$\dfrac{}{a^{-1} : u \xrightarrow{\epsilon} v}$	$a : v \xrightarrow{\epsilon} u \in G$
TRANS	$\dfrac{p : u \xrightarrow{l} v' \qquad q : v' \xrightarrow{l'} v}{pq : u \xrightarrow{ll'} v}$	
DN	$\dfrac{p : w \xrightarrow{\epsilon} w'}{c^{-1}pc' : u \xrightarrow{\epsilon} v}$	$c : w \xrightarrow{\delta} u \in G$ $c' : w' \xrightarrow{\delta} v \in G$

Figure 4.4: The logic $\Pi^U(G)$ of unification paths over G where G is the labeled directed graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$

Lemma 4.3.1 *(Π^U subject reduction)*

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ *be a labeled directed graph. Then* $\vdash_{\Pi^U(G)} p : u \xrightarrow{l} v$ *and* $p \longrightarrow_{\mathbf{D}'(D)} q$ *implies* $\vdash_{\Pi^U(G)} q : u \xrightarrow{l} v$.

Proof By Lemma 4.3.1 (Soundness), $\vdash_{\Pi^2(G)} p : u \xrightarrow{m} v$, where $m = l(p)$, and $l = \mu_{\mathbf{D}(\Sigma_D)}(m)$. Since $p \xrightarrow{*}_{\mathbf{D}'(D)} q$, by Lemma 4.2.3 (3), $\vdash_{\Pi^2(G)} q : u \xrightarrow{n} v$, $n = l(q)$, and $m \longrightarrow_{\mathbf{D}'(\Sigma_D)} n$. Since $m \in D^*(\Sigma_D)$, by Lemma 4.1.2, $m \approx_{\mathbf{D}(\Sigma_D)} n$. Therefore, $l = \mu_{\mathbf{D}(\Sigma_D)}(m)$. By Lemma 4.3.1 (Completeness), $\vdash_{\Pi^U(G)} q : u \xrightarrow{l} v$. ⊣

Example 4.3.15: In the graph G of Figure 4.1, $\vdash_{\Pi^U(G)} b_5 a_3^{-1} : w_4 \xrightarrow{g_1} x$. $\vdash_{\Pi^2(G)} b_4^{-1} b_6 : w_4 \xrightarrow{f_1 f_2} z$, but there is no Π^U deduction such that $\vdash_{\Pi^U(G)} b_4^{-1} b_6 : w_4 \xrightarrow{f_1 f_2} z$. Also, $\vdash_{\Pi^U(G)} b_5^{-1} b_5 : y \xrightarrow{\epsilon} y$, and $l(b_5^{-1} b_5) = g_1^{-1} g_1$. $\vdash_{\Pi^2(G)} b_5 b_5^{-1} : w_4 \xrightarrow{g_1 g_1^{-1}} w_4$ but there is no corresponding deduction in Π^U for the path $b_5 b_5^{-1}$. In other words, $b_5 b_5^{-1}$ is not a unification path even though its normal form (under 2-sided cancellation) yields the unification path ϵ. ◇

There is a close resemblance of this logic to the inductive definition of unification closure $\sim$ in Figure 3.3. The only apparent limitation is that SYM_{Π^U} in Figure 4.4 is restricted to leaves of deductions. This restriction is easily removed by replacing SYM_{Π^U} rule by the following "derived" rule:

$$\text{SYM}' \quad \frac{q : v \xrightarrow{\epsilon} u}{\mathit{inv}(q) : u \xrightarrow{\epsilon} v}$$

This replacement is justified by the following lemma:

Lemma 4.3.2 $\vdash_{\Pi^U(G)} q : u \xrightarrow{\epsilon} v$ *implies* $\vdash_{\Pi^U(G)}$ *inv*$(q) : v \xrightarrow{\epsilon} u$.

Proof By induction on the deduction $\vdash_{\Pi^U(G)} q : u \xrightarrow{\epsilon} v$. $\dashv$

Based on this observation, we introduce the inverse operation directly as part of the syntax of terms to obtain the logic P^U, discussed in the next section.

4.4 Logic of Unification Path Expressions

We consider the logic of *unification path expressions* over a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ shown in Figure 4.5 whose terms are drawn from the term algebra $\mathcal{T}(\Sigma_{\mathbf{Gr}}, D)$, where D is the set of labeled directed edges G. Judgements are of the form $p : u \xrightarrow{l} v$ where G is a labeled directed graph $\langle \Sigma_V, \Sigma_D, V, L, D \rangle$, $p \in T^*(\Sigma_{Gr}, D)$, and $l \in \Sigma_D^*$. $\vdash_{P^U(G)} p : u \xrightarrow{l} v$ denotes that there is a deduction in the logic P^U whose last step is $p : u \xrightarrow{l} v$. Since the free monoid $(D \cup D^{-1})^*$ extended with the inverse function *inv* is a $\Sigma_{\mathbf{Gr}}$-algebra, the function *flatten* from D to $(D \cup D^{-1})^*$ defined by

$$\mathit{flatten}(c) = c$$

uniquely extends to the homomorphism *flatten* $: T^*(\Sigma_{\mathbf{Gr}}, D) \longrightarrow (D \cup D^{-1})^*$. We abbreviate *flatten*(p) by $\overline{p}$. Thus, $\overline{\epsilon} = \epsilon$, $\overline{p^{-1}} =$ *inv*$(\overline{p})$, and $\overline{pq} = \overline{p}\ \overline{q}$. The *flatten* function helps

INIT	$\dfrac{}{c : u \xrightarrow{\delta} v}$	$c : u \xrightarrow{\delta} v \in D$
REF	$\dfrac{}{\epsilon : u \xrightarrow{\epsilon} u}$	$u \in V$
SYM	$\dfrac{p : u \xrightarrow{\epsilon} v}{p^{-1} : u \xrightarrow{\epsilon} v}$	
TRANS	$\dfrac{p : u \xrightarrow{l} v' \qquad q : v' \xrightarrow{l'} v}{pq : u \xrightarrow{ll'} v}$	
DN	$\dfrac{p : w \xrightarrow{\epsilon} w'}{c^{-1}(pc') : u \xrightarrow{\epsilon} v}$	$c : w \xrightarrow{\delta} u \in D$ $c' : w' \xrightarrow{\delta} v \in D$

Figure 4.5: The logic $P^U(G)$ of unification path expressions over a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$

us switch between proofs in Π^U and P^U:

Lemma 4.4.1 *(Soundness and completeness of P^U deductions)*

Let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ be a labeled directed graph.

1. *If $\vdash_{\Pi^U(G)} p : u \xrightarrow{l} v$, then $\vdash_{P^U(G)} p : u \xrightarrow{l} v$.*

2. *If $\vdash_{P^U(G)} p : u \xrightarrow{l} v$, then $\vdash_{\Pi^U(G)} \overline{p} : u \xrightarrow{l} v$.*

Proof The proof of (1) is by induction on Π^U deductions. A Π^U deduction may be converted to a P^U deduction by replacing each SYM_{Π^U} inference $a^{-1} : u \xrightarrow{\epsilon} v$ with the following inference in P^U.

$$\cfrac{\cfrac{c : u \xrightarrow{\epsilon} v \in D}{c : u \xrightarrow{\epsilon} v}\ \text{INIT}}{(c)^{-1} : v \xrightarrow{\epsilon} u}\ \text{SYM}$$

The proof of (2) is by induction on P^U deductions. Intuitively, each P^U subdeduction

$$\frac{q : v \xrightarrow{\epsilon} u}{(q)^{-1} : u \xrightarrow{\epsilon} v}\ \text{SYM}$$

may be replaced by the "derived" inference

$$\frac{q : v \xrightarrow{\epsilon} u}{\mathit{inv}(q) : u \xrightarrow{\epsilon} v}\ \text{SYM}'$$

which in turn can be replaced with a deduction in Π^U. $\dashv$

A unification path expression over a labeled directed graph $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ is said to have zero (0) or positive ($+$) parity if $G \vdash_{P^U} p : u \xrightarrow{l} v$ and $l = \epsilon$ or $l \in \Sigma_D^+$, respectively. If $x \in \{0, +\}$, we write $P^x_{uv}(G)$ to denote the set of all unification path expressions in G from u to v with parity x. We write $P^*_{uv}(G)$ to denote the set of all unification path expressions in G from u to v.

Associativity	$(pq)r$	$=$	$p(qr)$	$=$	pqr
1	$(pq)^{-1}$	$\longrightarrow$	$q^{-1}p^{-1}$		
2	$(p^{-1})^{-1}$	$\longrightarrow$	p		
3	ϵ^{-1}	$\longrightarrow$	ϵ		
4	$p\epsilon$	$\longrightarrow$	p		
5	ϵp	$\longrightarrow$	p		
6	pp^{-1}	$\longrightarrow$	ϵ		
7	$p^{-1}p$	$\longrightarrow$	ϵ		

Figure 4.6: Equational rewrite system R/A for free groups, where A consists of the Equational rule of Associativity and R consists of the remaining rules (Peterson and Stickel, 1981).

4.4.1 Normalization

Simplification of unification path expressions is achieved by using the rewrite system R/A of Peterson and Stickel [85] shown in Figure 4.6.

Theorem 4.4.1 *(Peterson-Stickel, 1981)*

The equational system R/A is convergent and complete with respect to the equational theory of free groups.

Rewriting under R/A may be used to simplify unification path expressions. The normal form of a term p under R/A rewriting is denoted $\mu_{\mathbf{Gr}}(p)$. The normal form $\mu_{\mathbf{Gr}}(p)$ may be computed by first flattening p to $\overline{p}$ and then reducing $\overline{p}$ to its normal form by applying the two-sided cancellation rules:

Lemma 4.4.2 *(Decomposition of normal form computation)*

If D is any set and $p \in T^*(\Sigma_{\mathbf{Gr}}, D)$, then $\mu_{\mathbf{Gr}}(p) = \mu_{\mathbf{D}'(D)}(\overline{p})$.

Proof Since $\overline{p} \approx_{\mathbf{Gr}} p$, both $\overline{p}$ and p have the same normal form under R/A. Now, $\overline{p}$ is flattened, which means every symbol in p is either c^{-1} or c, where $c \in D$. Hence, the only redexes in $\overline{p}$ are of the form $cc^{-1} \longrightarrow_{R/A} \epsilon$ and $c^{-1}c \longrightarrow_{R/A} \epsilon$. These redexes are exactly those that can be reduced using $\mathbf{D}'(D)$. ⊣

From this decomposition of normal form computation, it is straightforward to show that the "type" of a unification path expression is preserved by its normal form.

Lemma 4.4.3 *(P^U Weak subject reduction)*

Let G be a labeled directed graph and let $\vdash_{P^U(G)} p : u \xrightarrow{l} v$. If $p' = \mu_{\mathbf{Gr}}(p)$, then $\vdash_{P^U(G)} p' : u \xrightarrow{l} v$.

Proof By Lemma 4.4.1, if $\vdash_{P^U(G)} p : u \xrightarrow{l} v$, then $\vdash_{\Pi^U(G)} \overline{p} : u \xrightarrow{l} v$. By Lemma 4.4.2, $p' = \mu_{\mathbf{D}'(D)}(\overline{p})$ and by Lemma 4.3.1, $\vdash_{\Pi^U(G)} p' : u \xrightarrow{l} v$. ⊣

Unification path expressions are not closed under one-step R/A rewriting, as illustrated by the following example:

Example 4.4.16: (One-step rewriting in R/A)

Let G be a labeled directed graph consisting of the edges

$$\{a : w \xrightarrow{\epsilon} w', b_1 : w \xrightarrow{\mathsf{f.1}} u, b_2 : w' \xrightarrow{\mathsf{f.1}} v\}$$

Let $p = (b_1^{-1}(ab_2))^{-1}$. Clearly $\vdash_{P^U(G)} p : v \xrightarrow{\epsilon} u$. If $q = (ab_2)^{-1}(b_1^{-1})^{-1}$, then $p \longrightarrow_{R/A} q$, but $\not\vdash_{P^U(G)} q : v \xrightarrow{\epsilon} u$. However, $\vdash_{P^U(G)} b_2^{-1}a^{-1}b_1 : v \xrightarrow{\epsilon} u$, where $b_2^{-1}a^{-1}b_1$ is the normal form of p and q under R/A rewriting. $\diamond$

4.5 Unification Algorithm with source-tracking

Construction of P^U-deductions is easily integrated into the unification algorithm of Chapter 3 to obtain a unification algorithm that generates a proof of membership in the unification closure of a unifiable unification graph. The unification algorithm with source-tracking is shown in Figures 4.7 through Figures 4.9. The binding field of each variable and the return value of *find* is a tuple carrying a pointer to another variable and an extra element that is a unification path expression. The procedures *unify* and *union* also carry an extra parameter that is a unification path expression. The procedure *occurs?*(u, v) return either no or $\mathsf{yes}(x, p)$, where x is either 0 or $+$ and $p \in P^x_{uv}$. In the event of a cycle, the algorithm returns a unification path expression of positive parity from a vertex to itself. In the event of a clash, the algorithm returns a unification path expression of zero parity between vertices of different labels.

```
procedure unify(v1, v2, m) =
    let ⟨r1, p1⟩ = find(v1) and ⟨r2, p2⟩ = find(v2)
        in if r1 = r2 then return
            else case r1.type, r2.type
                strict, strict: union(r1, r2, (p1)^-1 m p2)
                functor, strict: unify(v2, v1, (m)^-1)
                strict, functor: let ans = occurs?(r2, r1)
                                     in case ans
                                         no: union(r1, r2, (p1)^-1 m p2)
                                         yes(_, q): fail(CYCLE, (p1)^-1 m p2 q)
                functor, functor:
                    if r1.L ≠ r2.L then fail(CLASH, (p1)^-1 m p2)
                    else
                        union(r1, r2, (p1)^-1 m p2);
                        for i = 1 to arity(r1.L) do
                            unify(r1.child(i), r2.child(i), (b1)^-1 (p1)^-1 m p2 b2)
                                where b1 = edge(r1, r1.child(i))
                                    and b2 = edge(r2, r2.child(i))
```

Figure 4.7: Unification algorithm with source-tracking: procedure *unify*

```
procedure union(r1, r2, p) = r1.binding := ⟨r2, p⟩

procedure find(v) =
    if unbound?(v) then return ⟨v, ε⟩
    else let ⟨v', p⟩ = v.binding
        in let ⟨r, q⟩ = find(v') in return ⟨r, pq⟩
```

Figure 4.8: Unification algorithm with source-tracking: procedures *union* and *find*

```
procedure occurs?(v1, v2) =
    let ⟨r1, p1⟩ = find(v1) and ⟨r2, p2⟩ = find(v2)
        in if r1 = r2 then return yes(0, p1(p2)^-1)
          else case r1.type
               strict: return no
               functor:
                    for i = 1 to arity(r1.L) do
                        let ans = occurs?(r1.child(i), r2)
                            in case ans
                                no: continue
                                yes(_, q): return yes(+, p1 b q(p2)^-1)
                                           where b = edge(r1, r1.child(i))
                    return no
```

Figure 4.9: Unification algorithm with source-tracking: procedure *occurs?*

If $m : \tau_1 \stackrel{?}{=} \tau_2$ is a term equation, its unification graph G contains term trees rooted at vertices u_1 and u_2, representing the terms τ_1 and τ_2 respectively, and an equational edge from v_1 to v_2 labeled m. The first call to unify is *unify*(u_1, u_2, m). The unification algorithm maintains the invariant that for every call *unify*(v_1, v_2, p), $p \in P^0_{v_1 v_2}$. Also, if *find*$(v_1) = \langle r_1, p_1 \rangle$, then $p_1 \in P^0_{v_1 r_1}$. Similarly, if *find*$(v_2) = \langle r_2, p_2 \rangle$, then $p_2 \in P^0_{v_2 r_2}$. The call *union*(r_1, r_2, p) maintains the invariant that $p \in P^0_{r_1 r_2}$. If the call *unify*(v_1, v_2, p) results in the call *union*$(r_1, r_2, (p_1)^{-1} p p_2)$, binding r_1 to r_2, where *find*$(v_1) = \langle r_1, p_1 \rangle$ and *find*$(v_2) = \langle r_2, p_2 \rangle$, then the vertex r_2 and the path expression $(p_1)^{-1} p p_2$ are stored as the binding information r_1.*binding*. The path $(p_1)^{-1} p p_2$ implicitly carries the information that v_1 is connected to v_2 with the path p. To recover the information that v_1 is connected to v_2 by p, we simply compute the path expression following the *binding* pointers from v_1 to v_2, which is done in three steps:

1. compute the path expression from v_1 to r_1 using *find*(v_1), which is p_1.

2. compute the path from r_1 to r_2 using *find*(r_1), which is $(p_1)^{-1}pp_2$.

3. compute the path from r_2 to v_2, which is $(p_2)^{-1}$.

The concatenation of these three path expressions, $p_1(p_1)^{-1}pp_2(p_2)^{-1}$, simplifies to p using the rewrite rules R/A. All paths expressions manipulated by the algorithm are unification path expressions. The yes answer returned by *occurs?* is now constructive, and is of one of the following forms:

1. $\mathsf{yes}(0, q)$, implying that $G/\!\sim\, \models [u]_\sim = [v]_\sim$, and this is witnessed by the unification path expression $q \in P^0_{uv}$.

2. $\mathsf{yes}(+, q)$ implying that $G/\!\sim\, \models [u]_\sim \xrightarrow{+} [v]_\sim$ and this is witnessed by the unification path expression $q \in P^+_{uv}$.

Lemma 4.5.1 *(Invariants for Unification algorithm with source-tracking)*

Let G be a unification graph of a term equation $\tau_1 \overset{?}{=} \tau_2$ between the terms τ_1 and τ_2 represented by the term graphs rooted at vertices v_1 and v_2. Then the following invariants are maintained by the unification algorithm:

1. *For each call unify*(u, v, p), $\vdash_{P^U(G)} p : u \xrightarrow{\epsilon} v$.

2. *For each call union*(u, v, p), $\vdash_{P^U(G)} p : u \xrightarrow{\epsilon} v$.

3. *If find*$(u) = \langle v, p \rangle$, *then* $\vdash_{P^U(G)} p : u \xrightarrow{\epsilon} v$.

4. *If occurs?*$(u, v) = \mathsf{yes}(x, p)$, *then* $x \in \{0, +\}$, $\vdash_{P^U(G)} p : u \xrightarrow{l} v$ *and* $l \in \Sigma_{\mathbf{N}}^{x}$, *where* $\Sigma_{\mathbf{N}}$ *is the set of context labels in* G.

Proof The proof follows from a straightforward inspection of the path expressions constructed at each stage of the algorithm. ⊣

The invariants show that at each stage, the unification algorithm with source tracking not only constructs the unification closure $\sim$ of a unification graph G, but also computes witnesses of membership in the relation $\sim$. These witnesses are unification path expressions. When *unify* fails, the algorithm presents a witness of the non-unifiability:

Corollary 4.5.1 *Let* Σ *be a signature, and let* G *be a unification graph of a* Σ*-term equation* $\tau_1 \stackrel{?}{=} \tau_2$ *between the terms* τ_1 *and* τ_2 *represented by the term graphs rooted at vertices* v_1 *and* v_2*. Then*

1. *If unify* $=$ *fail*$(CYCLE, q)$, *then* $\vdash_{P^U(G)} q : u \xrightarrow{l} u$ *for some vertex* u *in* G *and label* $l \in \Sigma_{\mathbf{N}}^{+}$.

2. *If unify* $=$ *fail*$(CLASH, q)$, *then* $\vdash_{P^U(G)} q : w \xrightarrow{\epsilon} w'$ *for some functor vertices* w, w' *in* G *such that* $L(w_1) \neq L(w_2)$.

The cost of constructing unification path expressions at each point in the algorithm is constant time per call to *unify*, *find*, *union* and *occurs?*, assuming structure sharing. Thus, overall, the addition of source-tracking to the unification algorithm increases the runtime of the unification algorithm by only a constant factor.

The process of normalization is orthogonal to the building of path expressions. It may be performed once the unification path has been computed. It is easily seen that normalization using R/A takes time proportional to the size of the term being normalized. Although this normal form is not always of minimum size, its computation is considerably less expensive than the computation of the shortest unification path.

4.6 Summary

In this chapter we identified the set of unification paths that capture membership in the unification closure of a unification graph. We introduced the deduction systems Π^U and P^U to compute these unification paths. Proofs in P^U may be simplified by rewriting in R/A. An extension of the unification algorithm that incorporates source-tracking by computing deductions in P^U was presented.

5

Error reconstruction in Curry-Hindley Type Inference

This chapter develops a framework for applying the results of unification source-tracking to the problem of diagnosing untypability in the Curry-Hindley [47] type system, abbreviated CH. The Curry-Hindley system defines a set of type rules for the lambda-calculus [4, 21], and forms the basis for type systems of modern programming languages.

The problem of error reconstruction for CH type inference is simple, but fundamental to the study of type inference in programs. It is the obvious basis for studying type error information in statically-typed languages with type inference.

The framework we use is based on defining a domain of syntactic constraints and type constraints. The rules of the type system determine a *generating function* that maps syntactic constraints to type constraints.

Our framework may be used at two levels of program granularity. In the first, the program is represented as a guarded flat system of *syntax equations* over locations (vertices of the program syntax tree). Syntactic constraints (equations) are mapped by a generating function to *type equations*, which are term equations whose variables range over program locations. Source-tracking information is obtained as an inverse of the generating function mapping syntax equations to type equations.

In the second approach, the domain of program locations is augmented with the special location $\square$ denoting a "hole" or an irrelevant location. Terms built using this extended domain are called *weakenings*. Weakenings are incomplete terms obtained by replacing zero or more variables in a flat term by $\square$. Equations over incomplete terms provide a finer-grained division of the program syntax graph and the unification graph.

There is a natural partial order over incomplete terms. This partial order induces a partial order over the domain of weakenings of sets of type equations. The partial order on weakenings of systems of type equations in turn corresponds to a simulation ordering between the unification graphs of the systems of type constraints.

It is well-known that CH-typability is reducible to unification of terms encoding types (see Hindley [46], Wand [109], or Cardelli [13]). The reduction of type inference to term unification may be viewed as a process in which syntactic constraints *generate* type constraints that are then solved by unification. This leads to the notions of *typability* and *untypability* of a syntactic constraint set. When programs are ill-typed, untypable syntactic constraint sets are identified, rather than subexpressions. The ordering of syntactic constraint sets leads to the notion of *minimally* untypable sets of syntactic constraints.

The approach outlined in this chapter relies on the properties of the constraint generation function and not on the solving of those constraints. Thus, the framework is independent of any specific type inference algorithm or data-structures. This is in contrast to the algorithm-based techniques that collect source information, such as Wand's algorithm [109], techniques that "explain" source information by "debugging," such as Duggan and Bent [35] and Beaven and Stansifer [8], or "tracing" techniques specific to a particular inference algorithm, such as Maruyama [69].

The form of the syntax and type constraints generated for CH are such that for every non-trivial type constraint, it is possible to associate a syntactic constraint called its *source*. This leads to a a *source function* that maps each type constraints back to its source.

There are several reasons why we focus on the study of the type inference problem for the Curry-Hindley type system: first, the problem of type inference provides a useful framework to which the results developed in the previous chapter can be applied. Second, computation of type information is more interesting for type inference, where types of variables are not declared and type information is completely inferred, than when variable types are declared. Lastly, the Curry-Hindley type system has been very well studied as the core of typed programming languages.

5.1 Type Systems

Some syntactically correct programs exhibit runtime behavior that includes application of an operation to values not supported by the operation, like adding a number to a function value. A *type system* is a set of rules that associates expressions in a programming language with syntactic entities called *types*. The type system usually decides what type, if any, an expression can have. Type systems are used to identify a subset of programs that make sense syntactically, but not semantically, and prevent their execution. A program is executed only after it is verified to be *well-typed* according to the rules of the type system. Type systems are usually part of the specification of a programming language.

A *typing judgement* or *typing* is an ordered triple $A \Rightarrow e : \tau$ consisting of a set of hypotheses, A, an expression, e, and a type, τ. It expresses the judgement that "under the assumptions specified in A, the expression e has the type τ." A subset of these typing judgements are identified as *well-typings*. The set of well-typings is usually defined by derivability in the logic of the type system. Let X denote a type system. We write $\vdash_X A \Rightarrow e : \tau$ to denote a well-typing derivable in the logic of the type rules of X. An expression e is *well-typed in* X if there are A and τ such that $\vdash_X A \Rightarrow e : \tau$. In this case, we say that the expression e *has the type* τ *under the assumptions* A *in the type system* X. Otherwise, e is said to be *ill-typed* or *untypable in* X. An expression that is well-typed in one type system could be untypable in a different type system. *Type inference* or *type reconstruction* is the process of taking an expression, and sometimes, a set of type assumptions, and constructing a well-typing for that expression if it is typable, and reporting that the expression is untypable otherwise.

Type systems are often specified using induction over program expressions. A well-typing for an expression then guarantees a well-typing for all its subexpressions. The problem of computing a well-typing for an expression e may be reduced to solving a system of constraints derived from the structure of e using the rules of the type system. The variables in these constraints, called *type variables*, are place-holders for the types of the constituent subexpressions of e. We state this principle as follows:

Proposition 5.1.1 *e is typable under a type system X if and only if the system of type constraints generated under the typing rules of X for e are solvable.*

The structure of the system of constraints is of course dependent on the type system X. Usually type constraints are encoded either as equations or inequations between types, and these constraints are solved by term-unification and its many variants and extensions. Instances of Proposition 5.1.1 are the reduction of CH-typability to unification of systems of equalities between type terms (see for example, Hindley [46], Cardelli [13], Milner [75], Wand [109]), the reduction of the Damas-Milner type system [28] for parametric polymorphism to unification, and the reduction of Milner-Mycroft typability [81] for polymorphic recursion to semi-unification (Henglein [44], Kfoury et al. [58]).

When the system of type constraints generated for an expression e has a solution in the form of a substitution s, often called a *a substitution*, the type of e is just the value of $s(t)$, where t is the type variable associated with e. On the other hand, when the system of type constraints is unsolvable, the system of constraints and the expression e must be analyzed to identify substructures of the expression e that generate subsystems of constraints that are unsolvable.

The origins of type systems can be traced to the early twentieth century in the efforts to provide a logical formalization of mathematics. Bertrand Russell introduced his theory of "types" as a way of circumventing paradoxes in set theory and the formalization of mathematics [93]. The simply-typed lambda calculus was introduced by Alonzo Church [21]. The connection between unification and simply-typed lambda-calculus was shown by Hindley [46, 47, 48]. A formal type system and type inference algorithms were central to the design of the language ML(Milner et al. [77]), which was originally designed as a meta language for theorem proving (Milner et al. [75]). More recently, type systems have been used extensively for modeling safety, security, mobility, and correctness of software systems. Survey articles on types include Cardelli and Wegner [15], Mitchell [78] and Cardelli [14]. Recent texts devoted to the study of type systems in programming languages include Schmidt [95] and Pierce [87].

5.2 Preliminaries: Partial Orders

The quality of source-tracking information can be characterized by imposing an ordering on the syntactic and type constraints derived from source-tracking. In this section, we briefly state the definitions and facts related to partial orders that we require.

Given a set P and a binary relation $\sqsubseteq_P$ on P, the pair $(P, \sqsubseteq_P)$ is a *pre-order* if $\sqsubseteq_P$ is reflexive and transitive. $(P, \sqsubseteq_P)$ is a *partial order* or *poset*, if $\sqsubseteq_P$ is reflexive, transitive, and antisymmetric. The relation $x \sqsubset_P y$ is defined as "$x \sqsubseteq_P y$ and $y \not\sqsubseteq_P x$." x and y are *comparable under* $\sqsubseteq_P$ if $x \sqsubseteq_P y$ or $y \sqsubseteq_P x$.

If $(P, \sqsubseteq_P)$ is a partial order, then for each subset Q of P, $\sqsubseteq_P$ induces a partial order $\langle Q, \sqsubseteq_Q \rangle$, where for each $x, y \in Q$, $x \sqsubseteq_Q y$ if and only if $x \sqsubseteq_P y$. Q is an *antichain* if $x \sqsubseteq_Q y$ implies $x = y$. If $Q \subseteq P$, then, the *down-set of* Q, denoted $Q\!\downarrow$ is $\{y \in P \mid \exists x \in Q : y \sqsubseteq_P x\}$. If T is any set and $\perp$ is a special symbol not in T, then $T_\perp$ denotes the *flat poset* $\langle T \cup \{\perp\}, \sqsubseteq \rangle$, where $x \sqsubseteq y$ if and only if $x = \perp$ or $x = y$.

The partial order $(P, \sqsubseteq_P)$ induces an ordering on functions mapping into P. If A is any set and f and g are functions from A to P, then $g \sqsubseteq_P f$ if and only if for each $a \in A$, $g(a) \sqsubseteq_P f(a)$.

If $\sqsubseteq_P$ is a partial order, then the *lower power-domain ordering induced by* $\sqsubseteq_P$[1] is the preorder $\langle \mathbf{2}^P, \sqsubseteq \rangle$ where $S' \sqsubseteq S$ if and only if $\forall s' \in S'. \exists s \in S : s' \sqsubseteq_P s$. It is easy to

[1]The lower power-domain ordering is also known as the *Hoare ordering*.

verify that $S' \sqsubseteq S$ if and only if $S'\!\downarrow\subseteq S\!\downarrow$. Given a set $Q \subseteq P$, we are often interested in the restriction of $\sqsubseteq$ to $\{S \subseteq P | S \sqsubseteq Q\}$. If Q is finite, then every non-empty set $S \sqsubseteq Q$ may be uniquely written as $\bigcup_{i=1}^{n} \{\{x_i\}\!\downarrow\}$ where $\{x_1, \ldots, x_n\}$ is an antichain. We call this the *normal form* of S. The ordering $\sqsubseteq$ becomes a partial order if sets with identical normal forms are considered equal. It is easy to verify that if $S, S' \subseteq P$, and $S' \sqsubset S$, where $\sqsubseteq$ is the lower power-domain ordering induced by $\sqsubseteq_P$, then there is an $s \in S$ such that for all $s' \in S'$, if s and s' are comparable, then $s' \sqsubset_P s$.

Let $g : A \longrightarrow B$. If $\eta \in B$ and $s \in A$, then s is a *source for* η *under* g if $\eta = g(s)$. A *source function for* g is a function $h : B \longrightarrow A$ such that $g \circ h = I_B$[2]. If h is a source function for $g : A \longrightarrow B$, then for each $\eta \in B$, $h(\eta)$ is a source for η under g. A source function for g exists if and only if g is onto. Also, if g is one-to-one and onto, then the inverse g^{-1} of g is the unique source function for g.

A function $g : A \longrightarrow B$, where $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are partial orders, is *monotone* if for each $a, a' \in A$, $a \sqsubseteq_A a'$ implies $g(a) \sqsubseteq_B g(a')$. g is *strictly monotone* if $a \sqsubset_A a'$ implies $g(a) \sqsubset_B g(a')$. If $\eta \in B$, then $s \in A$ is an *imprecise source for* η *under* g if $\eta \sqsubseteq_B g(s)$.

[2] h is more generally known as a *section* of g.

5.2.1 Σ-Graph simulations

The notion of simulation of errors in unification and type inference is captured by relations on graphs called graph simulations. The idea of simulation is tied to the notion of *observable behavior*. Informally, the observations on a graph consist of the labels on vertices and directed edges of the graph. A graph simulation from a graph G' to G shows how the observable behavior of G' may be captured by the observable behavior G.

Let $G = \langle V, L, D\rangle$ and $G' = \langle V', L', D'\rangle$ be Σ-graphs. A relation $R \subseteq V' \times V$ is a *simulation from G' to G* if for each $(u', u) \in R$

1. $L(u) = L'(u)$
2. If $L'(u') = \emptyset$, then $u = u'$
3. If $u' \xrightarrow{\delta} v'$, then there is a $v \in V$ such that $u \xrightarrow{\delta} v$ and $v'Rv$.

R is a *total simulation from G' to G* if for each $v' \in V'$, there is a $v \in V$ such that $v'Rv$.

G simulates G', or *G' is simulated by G*, written $G' \sqsubseteq G$ if there is a total simulation from G' to G. The relation $\sqsubseteq$ is a pre-order.

Example 5.2.17: Let $\Sigma = \{\mathsf{f} \mapsto 2\}$, let $G = \langle V, L, D\rangle$ and $G' = \langle V', L', D'\rangle$ be Σ-graphs, where

1. $V' = \{w'_1, w'_2, n_1\}$

2. $L' = \{n_1 \mapsto \emptyset, w'_1 \mapsto \{\mathsf{f}\}, w'_2 \mapsto \{\mathsf{f}\}\}$,

3. $D' = \{w'_1 \xrightarrow{1} n_1, w'_1 \xrightarrow{2} n_1, w'_2 \xrightarrow{1} n_1\}$.

and let $G = \langle \Sigma_V, \Sigma_D, V, L, D \rangle$ where

1. $V = \{w_1, n_1\}$

2. $L = \{n_1 \mapsto \emptyset, w_1 \mapsto \{\mathsf{f}\}\}$,

3. $D = \{w'_1 \xrightarrow{1} n_1, w'_1 \xrightarrow{2} n_1\}$.

Clearly $G \sqsubseteq G'$. Also $G' \sqsubseteq G$ because of the total simulation $\{(n_1, n_1), (w'_1, w_1), (w'_2, w_1)\}$ from G' to G. $\diamond$

5.3 The CH type system: syntax and type rules

The main object of study in this chapter is the programming language Λ of Λ-*expressions*. Let $\mathbf{V}$ be a denumerable set of *program variables*. The language Λ of Λ-*expressions* is defined by the context free grammar

$$e := \lambda x.\ e \mid x \mid @e\ e$$

where x ranges over $\mathbf{V}$. The constructors λ and $@$ are used for function construction and application, respectively. λ is known as a *binding construct* and in the expression $\lambda\ x.\ e$, the occurrence of x is called a *binding occurrence*.

The set of *types* $\tau \in \Psi$ is the free term-algebra over the signature $\Sigma_\Psi = \{(\rightarrow, 2)\}$ and a denumerable set *N* of *type variables*. Types may be described by the context-free grammar

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

in which t ranges over type variables and τ ranges over types.

5.3.1 Abstract Syntax graphs representations of closed Λ-expressions

Usually Λ-expressions are represented as *abstract syntax trees* whose internal nodes are labeled either by λ or $@$, and whose leaves are labeled by variables. It is useful, however, to consider term graph representations of Λ-expressions in which variable reference nodes point directly to the binding variable occurrence in the case of λ-bound variables. Such terms graphs, called *abstract syntax graphs* (ASG's), can be generated in a straightforward manner from abstract syntax trees. This is best illustrated by an example.

Example 5.3.18: Let e be the Λ-expression

$$\lambda_0\ x_1.\ (@_2\ (@_3\ x_5\ x_6)\ x_4)$$

The subscripts refer to locations in the term graph. The ASG of e, shown in Figure 5.1, is the pointed term graph $\langle n_0, T_e \rangle$, where $T_e = \langle W_e, X_e, b_e \rangle$ is a term graph whose set of functor vertices W_e is equal to $\{n_0, n_2, n_3, n_4, n_5, n_6\}$, the set of strict vertices X_e is $\{n_1\}$, and the function b_e is

$$\begin{array}{rclcrcl} n_0 & = & \lambda(n_1, n_2) & \quad & n_2 & = & @(n_3, n_4) \\ n_3 & = & @(n_5, n_6) & & n_4 & = & \lambda\mathsf{var}(n_1) \\ n_5 & = & \lambda\mathsf{var}(n_1) & & n_6 & = & \lambda\mathsf{var}(n_1) \end{array}$$

$\diamond$

5.3.2 CH Type Rules

The CH type system, defined in Figure 5.2, relates expressions in Λ with types. The set of CH-well-typings is defined as the least set closed under the rules in Figure 5.2. In these rules, A is a finite function from V to types, called a *type environment*. Type environments maintain hypotheses about types of variables possibly occurring free in an expression.

In our formulation of these rules, subscripts are used to identify parts of an expression. It is useful to think of each vertex i in the parse tree of e as having an attribute that is the type τ_i of that vertex. Each row in Figure 5.2 contains a rule name, a typing rule, and a *side condition* that encodes a constraint between types of parts of the expression. Side conditions express constraints on types that need to hold in addition to the antecedents of a type rule in order to be able to infer the consequent of the type rule. Type constraints

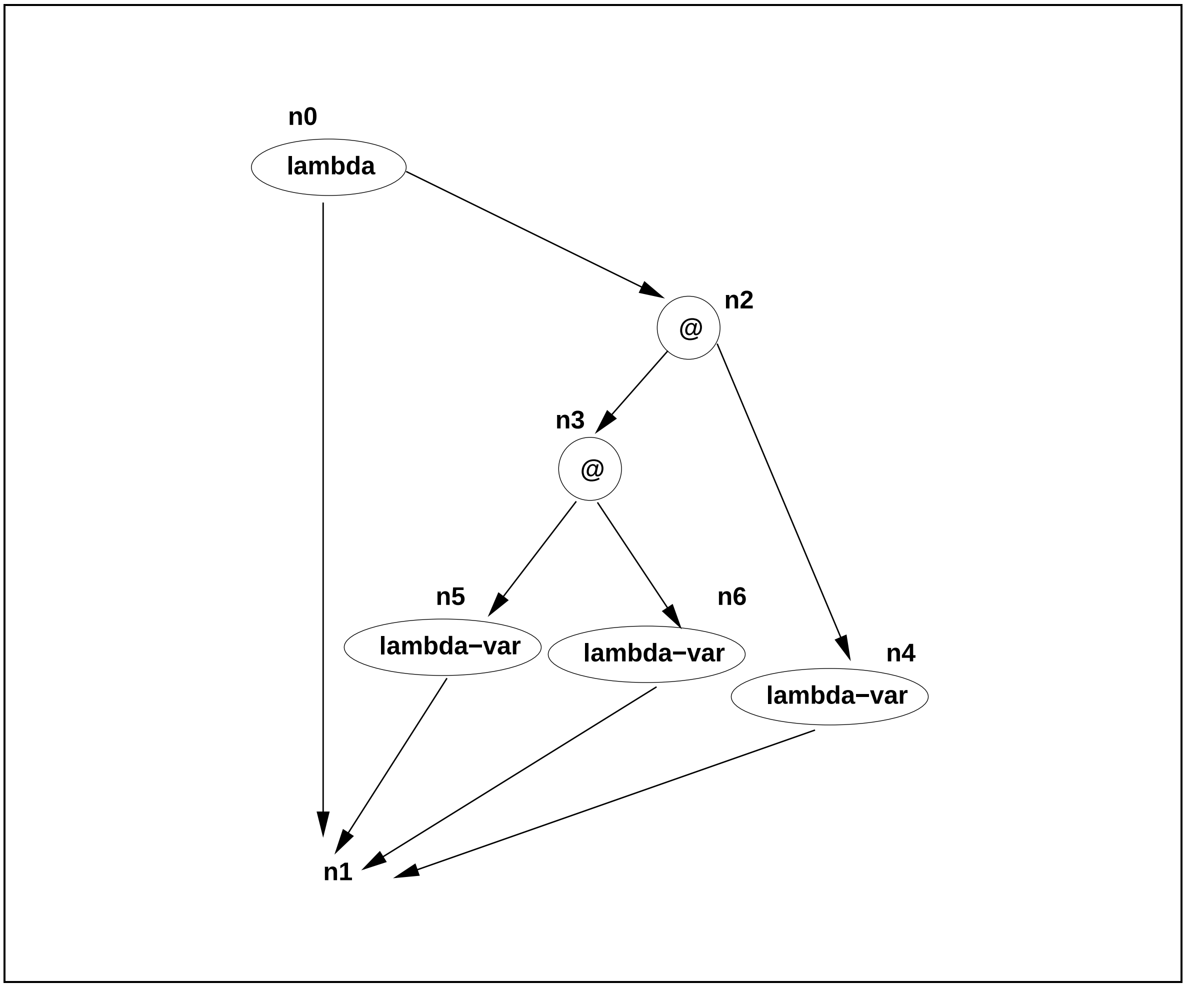


Figure 5.1: ASG for expression in Example 5.3.18

are expressed as *type equations*, which are term equations between pairs of type terms. For the consequent of the rule to be a well-typing, the side condition must be satisfiable. Note that the type system CH is *syntax-directed*: given an expression e, there is exactly one rule in CH with an instance whose consequent contains e. The set of type equations generated as side conditions by applying the corresponding type rules of Figure 5.2 to the subexpressions of e is called the *canonical set of type equations of* e *under CH*, and is denoted E_e.

A *principal type* of a closed expression e is a type of e from which all other types of e may be derived by applying a substitution. The Curry-Hindley system admits principal types. Hence, for any closed expression e, if e is typable in CH, then there is a type τ such that $\vdash_{CH} \emptyset \Rightarrow e : \tau$, and for any other τ' such that $\vdash_{CH} \emptyset \Rightarrow e : \tau'$, there is a substitution σ such that $\tau' = \sigma\tau$. The type τ is called the *principal type of* e. Principal types are unique modulo renaming of variables. For example, $\vdash_{CH} \emptyset \Rightarrow \lambda\, x.\, x : t \rightarrow t$, and and $t{\rightarrow}t$ is the principal type of $\lambda\, x.\, x$ (modulo renaming). Thus, $\vdash_{CH} \emptyset \Rightarrow \lambda\, x.\, x : \tau \rightarrow \tau$, where $\tau = (t_1 \rightarrow t_1)$, and the type $\tau \rightarrow \tau$ can be obtained from the type $t{\rightarrow}t$ by the substitution $t \mapsto \tau$.

The problem of type inference was formulated and solved by Curry [26] and Hindley [47] in the context of combinatory logic. The problem was independently discovered and solved in a programming language context by Morris [79] and Milner [75]. Hindley and Milner show how the principal type of a typable Λ-expression e is the mgu of the set of type equations generated by e. A simpler proof of Hindley's result was presented by Wand [109].

In the rest of the chapter, however, we will be concerned not with the question of computation of principal types, but with the problem of diagnosing the situation when an expression has no type at all. In this context, the reduction of type inference in CH to unification may be stated as:

Theorem 5.3.1 *If e is a Λ-expression, then e is well-typed under CH if and only if the canonical set of type equations of e is unifiable with an acyclic unifier.*

For the remainder of this chapter, we identify unifiability with the existence of acyclic unifiers. Thus, we say that a system of equations E is *unifiable* if E is unifiable with an acyclic unifier. Otherwise, E is *non-unifiable*. E is *minimally non-unifiable under subset ordering* if E is non-unifiable and for every $E' \subset E$, E' is unifiable.

5.4 Subset-based generating function for CH

By specifying a Λ-expression e in terms of a set of syntax equations, it is possible to construct a generating function *gen*$_e$ from the set of syntax equations of e to a set of type equations. Source-tracking is then expressed as a source function for *gen*$_e$.

VAR	$\dfrac{}{A[x_j : \tau_j] \Rightarrow x_i : \tau_i}$	$\tau_i = \tau_j$
LAMBDA	$\dfrac{A[x_j : \tau_j] \Rightarrow e_k : \tau_k}{A \Rightarrow \lambda_i\, x_j.\, e_k : \tau_i}$	$\tau_i = \tau_j \rightarrow \tau_k$
APP	$\dfrac{A \Rightarrow e_j : \tau_j \qquad A \Rightarrow e_k : \tau_k}{A \Rightarrow @_i\, e_j\, e_k : \tau_i}$	$\tau_j = \tau_k \rightarrow \tau_i$

Figure 5.2: The CH type inference system

Let e be a closed Λ-expression whose ASG is $T_e = \langle W_e, X_e, b_e \rangle$. The generation of type equations under the CH type system is expressed using the generating function $\mathit{gen}_e : b_e \longrightarrow E_e$ defined by the schema in Figure 5.3, where E_e is the canonical set of type equations of e under CH. By construction, gen_e is one-to-one and onto. Therefore, each type equation η in E_e has a unique source under gen_e. Furthermore, the inverse $\mathit{gen}_e^{-1} : E_e \longrightarrow b_e$ is the unique source function for gen_e.

Example 5.4.19: Continuing Example 5.3.18, where e is the expression

$$\lambda_0\; x_1.\; (@_2\; (@_3\; x_5\; x_6)\; x_4)$$

$$\begin{array}{rcl} \textit{gen}_e(n_i = \lambda(n_j, n_k)) & = & n_i \overset{?}{=} n_j \rightarrow n_k \\ \textit{gen}_e(n_i = @(n_j, n_k)) & = & n_j \overset{?}{=} n_k \rightarrow n_i \\ \textit{gen}_e(n_i = \lambda\mathsf{var}(n_j)) & = & n_i \overset{?}{=} n_j \end{array}$$

Figure 5.3: Schemas for the generating function $\textit{gen}_e$ mapping equations in b_e to type equations in the canonical system E_e of type equations of e in the Curry-Hindley type system , where $T_e = \langle W_e, X_e, b_e \rangle$ is the ASG of the closed Λ-expression e.

the function $\textit{gen}_e$ is equal to:

$$\begin{array}{rclcrcl} n_0 & = & \lambda(n_1, n_2) & \mapsto & n_0 & \overset{?}{=} & n_1 \rightarrow n_2 \\ n_2 & = & @(n_3, n_4) & \mapsto & n_3 & \overset{?}{=} & n_4 \rightarrow n_2 \\ n_3 & = & @(n_5, n_6) & \mapsto & n_5 & \overset{?}{=} & n_6 \rightarrow n_3 \\ n_4 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_4 & \overset{?}{=} & n_1 \\ n_5 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_5 & \overset{?}{=} & n_1 \\ n_6 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_6 & \overset{?}{=} & n_1 \end{array}$$

Figure 5.4 shows the unification graph representing E_e, along with labels and orientations for all edges in the graph. We let E_e denote both the system of type equations and the unification graph representing them.

◇

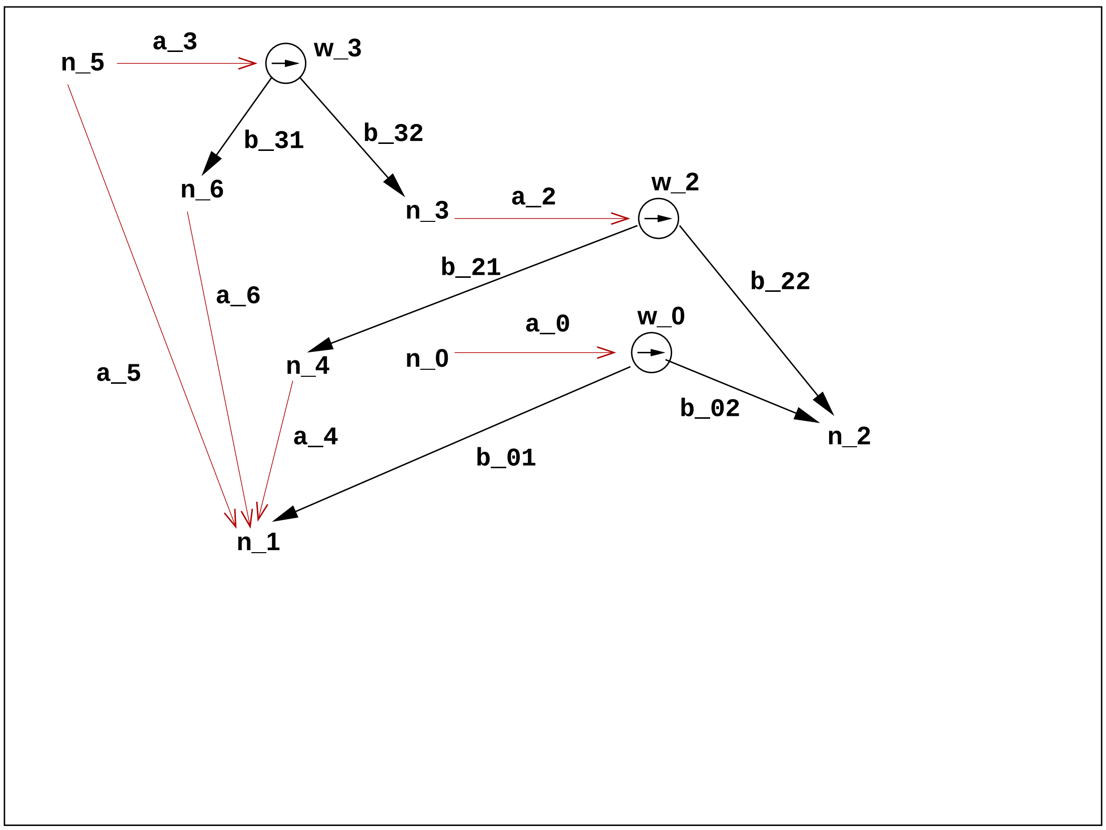


Figure 5.4: Unification graph with oriented and labeled edges representing type equations generated for the expression of Example 5.3.18

5.4.1 Untypability of subsets of syntax equations

Let e be a closed Λ-expression whose ASG is the term graph $\langle W_e, X_e, b_e\rangle$. If e is untypable, then the untypability of e may be traced to the non-unifiability of some subset $E \subseteq E_e$ of type equations, where E_e is the canonical set of type equations generated for e. Since each type equation in E_e is generated from a syntax equation in b_e, the source of untypability of an expression e can be traced to subsets of syntax equations of e.

Definition 5.4.2: Let e be an untypable Λ-expression whose term graph is $\langle W_e, X_e, b_e\rangle$ and gen_e be the generating function for e. The subset of syntax equations $B \subseteq b_e$ is *untypable* if and only if $\mathit{gen}_e[B]$ is non-unifiable. B is *minimally untypable* if and only if B is untypable and for all $B' \subset B$, $\mathit{gen}_e[B']$ is unifiable. □

The inverse function gen_e^{-1} immediately yields minimally untypable syntax equation sets from minimally unifiable type equation sets:

Lemma 5.4.1 *(Minimally untypable sets of syntax equations)*

Let e be a decorated expression whose canonical set of type equations under CH is E_e. If $E \subseteq E_e$ is minimally non-unifiable, then $\mathit{gen}_e^{-1}[E]$ is minimally untypable.

Proof Follows from the property that gen_e and is one-to-one and onto. ⊣

The following example illustrates the extraction of untypable sets of syntax equations from non-unifiable sets of type equations.

Example 5.4.20: The expression e

$$\lambda_0\, x_1.\, (@_2\, (@_3\, x_5\, x_6)\, x_4)$$

of Example 5.4.19 is untypable because the unification graph representing the set of type equations E_e generated by e is non-unifiable (since only non-cyclic unifiers are allowed).

Two sets E_1 and E_2 of minimally non-unifiable subsets of E_e can be identified. The subset $E_1 \subseteq E_e$ consists of the equations:

$$n_5 \stackrel{?}{=} n_6 {\rightarrow} n_3 \qquad n_5 \stackrel{?}{=} n_1 \qquad n_6 \stackrel{?}{=} n_1$$

and its unification graph is shown in Figure 5.5(a). The set E_1 is generated by the set of syntax equations B_1:

$$\begin{array}{lllllll} n_3 & = & @(n_5, n_6) & \mapsto & n_5 & \stackrel{?}{=} & n_6 {\rightarrow} n_3 \\ n_5 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_5 & \stackrel{?}{=} & n_1 \\ n_6 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_6 & \stackrel{?}{=} & n_1 \end{array}$$

shown in Figure 5.5(b).

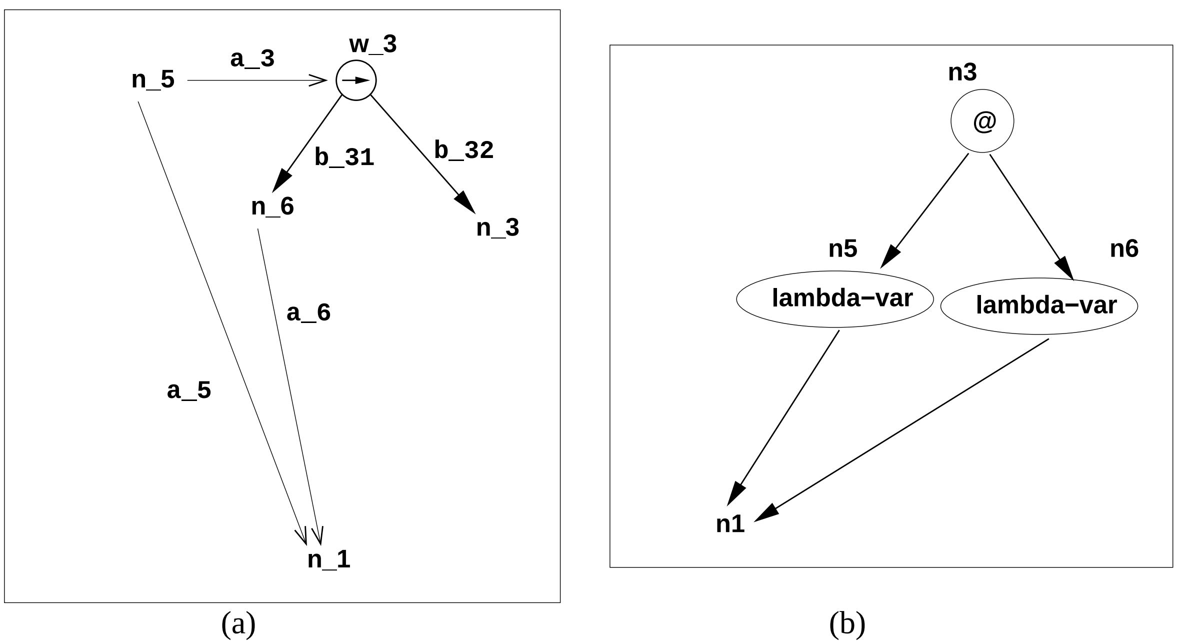


Figure 5.5: (a) the unification graph of the set of type equations E_1 of Example 5.4.20 and (b) syntax equations B_1 generating E_1.

The subset E_2 is

$$\begin{array}{rclcrcl} n_3 & \stackrel{?}{=} & n_4 \to n_2 & \qquad & n_5 & \stackrel{?}{=} & n_6 \to n_3 \\ n_4 & \stackrel{?}{=} & n_1 & & n_5 & \stackrel{?}{=} & n_1 \end{array}$$

and is shown in Figure 5.6(a). The set E_2 is generated by the set of syntax equations B_2:

$$\begin{array}{rclcrcl} n_2 & = & @(n_3, n_4) & \mapsto & n_3 & \stackrel{?}{=} & n_4 \to n_2 \\ n_3 & = & @(n_5, n_6) & \mapsto & n_5 & \stackrel{?}{=} & n_6 \to n_3 \\ n_4 & = & \lambda \mathsf{var}(n_1) & \mapsto & n_4 & \stackrel{?}{=} & n_1 \\ n_5 & = & \lambda \mathsf{var}(n_1) & \mapsto & n_5 & \stackrel{?}{=} & n_1 \end{array}$$

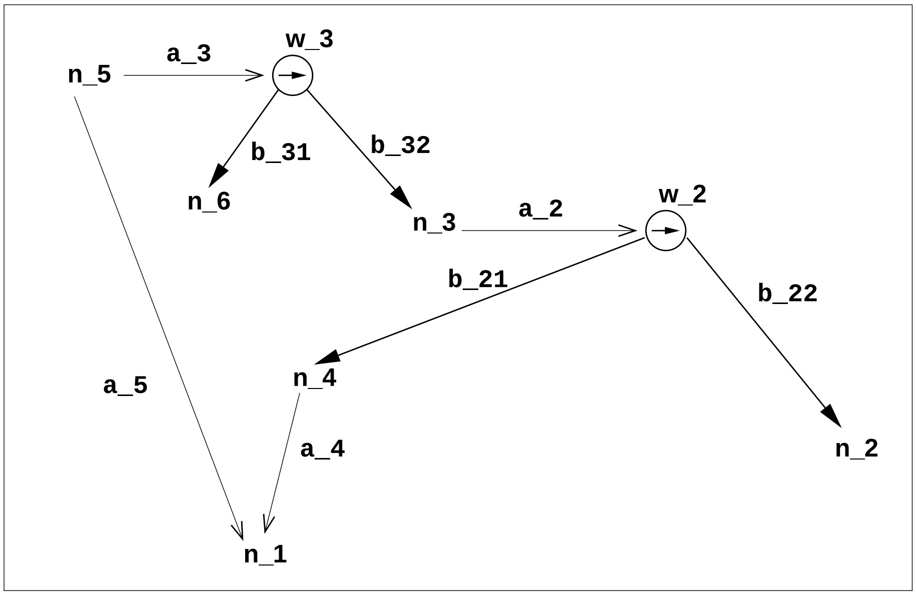
shown in Figure 5.6(b).

The syntax equation sets B_1 and B_2 are both minimally untypable.

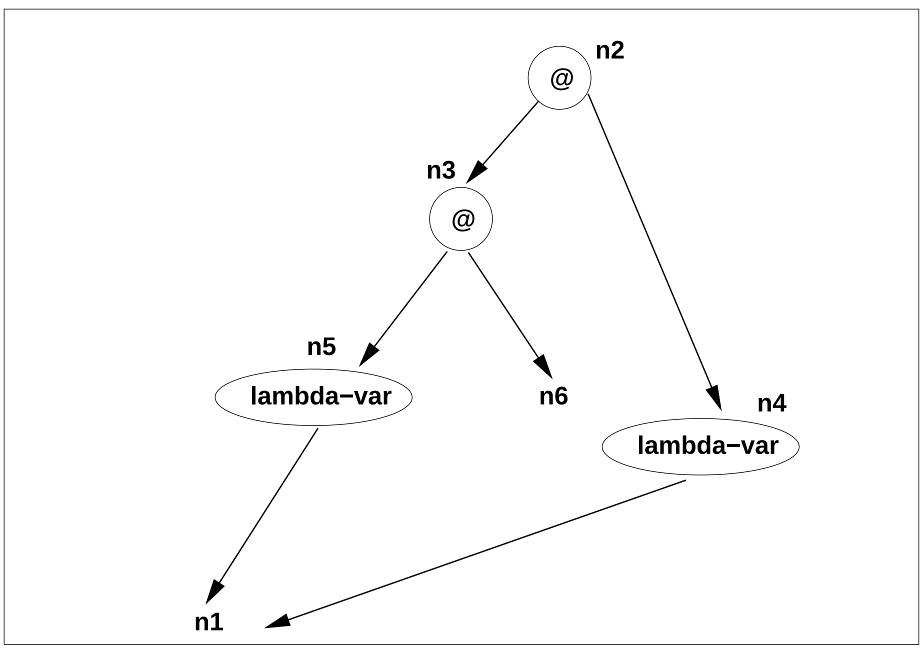
$\diamond$

5.5 Weakenings of type equations

The last section introduced the function *gen*$_e$ that expressed the generation of type equations from syntax equations of a Λ-expression e. This formulation of syntactic and type constraints allowed for information that is redundant in determining the non-typability of



(a)



(b)

Figure 5.6: (a) the unification graph of the set of type equations E_2 of Example 5.4.20, and (b) the ASG of the syntax equations B_2 generating E_2.

an expression. For example, not all variables in the set of type equations E_1 of Example 5.4.20 contribute to the cyclicity of the unifier of E_1. The source of the cyclicity is the cycle involving the location variables n_5, n_6 and n_1. The location n_3 does not contribute to this cycle, and should therefore not be part of the syntactic constraints generating the cycle. Absence of locations from constraints is modeled by *weakenings*, obtained by replacing zero or more locations with a $\square$. The symbol $\square$ may be thought of as a "hole," each occurrence of which representing an distinct anonymous location.

Recall from Chapter 3 that if N is a set of variables, and Σ is a signature, the set of incomplete Σ-terms over N is the set of Σ-terms over $N \cup \square$, where $\square$ denotes a special variable not occuring in N. The set $N \cup \square$ may be considered a flat poset, with the ordering $x \sqsubseteq y$ if and only if $x = y$ or $x = \square$. The set of *weak terms* is the set of wellfounded Σ-terms over $N \cup \square$.

The set of *weak term equations* is the set of term equations over weak terms. The flat poset $\sqsubseteq$ induces a partial order $\sqsubseteq$ on weak terms and weak term equations:

1. $x \sqsubseteq y$, if $x, y \in N \cup \square$ and $x = y$ or $x = \square$.

2. $f(\tau_1, \ldots, \tau_n) \sqsubseteq f(\tau_1', \ldots, \tau_n')$, if $\tau_i, \tau_i' \in T_\Sigma(N \cup \square)$ and $\tau_i \sqsubseteq \tau_i'$, $1 \leq i \leq n$.

3. $\tau_1 = \tau_2 \sqsubseteq \tau_1' = \tau_2'$, if $\tau_1, \tau_2, \tau_1', \tau_2' \in T_\Sigma(N \cup \square)$, $\tau_1 \sqsubseteq \tau_1'$ and $\tau_2 \sqsubseteq \tau_2'$.

The partial order $\sqsubseteq$ on weak term equations induces a lower power-domain order over sets of weak term equations: $E \sqsubseteq E'$ if for each $\eta \in E$, there is an $\eta' \in E'$ such that $\eta \sqsubseteq \eta'$.

Given a set E of weak term equations, the set of *weakenings of* E is defined to be the downset $E\!\downarrow$ of E. Weakenings are a generalization of *linearizations* (Le Chenadec [63]), which are obtained by replacing any one occurrence of a variable occurring more than once with a $\square$.

Example 5.5.21: For example, the set of weakenings of $\{n_1 = n_2 \to n_3\}$ is the set of equations:

$$\begin{array}{cccc}
\square = \square \to \square & n_1 = \square \to \square & \square = n_2 \to \square & \square = \square \to n_3 \\
n_1 = n_2 \to \square & \square = n_2 \to n_3 & n_1 = \square \to n_3 & n_1 = n_2 \to n_3
\end{array}$$

The set of weakenings of $\{n_4 = n_5\}$ consists of the equations

$$\square = \square \qquad n_4 = \square \qquad \square = n_5 \qquad n_4 = n_5$$

$\diamond$

The weakening relation on systems of type equations corresponds to the simulation relation on their unification graphs.

Lemma 5.5.1 *(Weakenings are graph simulations)*

Let E' and E be flat sets of weak type equations such that $E' \sqsubseteq E$. Let G' and G be the unification graphs of E' and E respectively. Then $G' \sqsubseteq G$.

Proof (Sketch) Let $n'_0 = f(n'_1, \ldots, n'_k)$ be an equation in E'. Then there is an equation $n_0 = f(n_1, \ldots, n_k)$ in E such that $n'_i \sqsubseteq n_i$, for $0 \leq i \leq k$. The unification graph G' of E' contains a vertex w' and an edge from w' to n'_i provided n'_i is not $\square$ for each i, $1 \leq i \leq k$. Similarly, there is an equational edge from n'_0 to w' if n'_0 is not $\square$. For the equation $n_0 = f(n_1, \ldots, n_k)$ in E, the unification graph G of E contains a vertex w, and an edge from w to n_k provided n_k is not $\square$. Similarly, there is an equational edge from n_0 to w if n_0 is not $\square$. For each pair of equations $n'_0 = f(n'_1, \ldots, n'_k)$ and $n_0 = f(n_1, \ldots, n_k)$, we can construct the set $\{(w', w)\} \cup \{(n'_i, n_i) \mid n'_i \neq \square\}$. It is easy to verify that the union of all such sets for each pair of equations $\langle n'_0 = f(n'_1, \ldots, n'_k), n_0 = f(n_1, \ldots, n_k)\rangle$ yields a simulation from G' to G. $\dashv$

5.6 Weakening-based generating function for CH

Let e be a closed Λ-expression, represented by the term graph of syntax equations $\langle W_e, X_e, b_e\rangle$. Let E_e be the set of type equations generated for e under the type system CH. The type equation generation function $gen_e : b_e \longrightarrow E_e$ may be extended to $wgen_e$, whose domain is $b_e\!\downarrow$ (the set of weakenings of b_e, also called *weak syntax equations*), and whose range is $E_e\!\downarrow$ (the set of weakenings of E_e, also called *weak type equations*). The definition of $wgen_e$, shown in Figure 5.7 is identical to that of gen_e except that locations range over $W_e \cup X_e \cup \{\square\}$. It is easy to verify that $wgen_e$ is strictly monotone and onto

$$\begin{array}{lcl} \textit{wgen}_e(n_i = \lambda(n_j, n_k)) & = & n_i \stackrel{?}{=} n_j \rightarrow n_k \\ \textit{wgen}_e(n_i = @(n_j, n_k)) & = & n_j \stackrel{?}{=} n_k \rightarrow n_i \\ \textit{wgen}_e(n_i = \lambda\mathsf{var}(n_j)) & = & n_i \stackrel{?}{=} n_j \end{array}$$

Figure 5.7: Generating function $\textit{wgen}_e$ that maps equations in $b_e{\downarrow}$ to equations in $E_e{\downarrow}$ in the CH type system , where $T_e = \langle W_e, X_e, b_e \rangle$ is the ASG of e, and E_e is the canonical set of type equations for e.

but not one-to-one: for example, the type equation $\Box \stackrel{?}{=} \Box \rightarrow \Box$ is generated by the syntax equations $\Box = @(\Box, \Box)$ and $\Box = \lambda(\Box, \Box)$. Since $\textit{wgen}_e$ is onto, every weakening η in $E_e{\downarrow}$ has at least one source, and there is at least one source function for $\textit{wgen}_e$. We use $\textit{wsrc}_e$ to denote an arbitrary source function for $\textit{wgen}_e$.

A weak type equation $\tau_1 = \tau_2$ is *non-trivial* if each of τ_1 and τ_2 contains at least one type variable, and *trivial* otherwise. A non-trivial type constraint generated by $\textit{wgen}_e$ has a unique source.

Lemma 5.6.1 *(Unique source for non-trivial constraints)*

Let $T_e = \langle W_e, X_e, b_e \rangle$ be the term graph of a closed Λ-expression e, and let E_e denote the canonical set of type equations generated for e. If $\eta \in E_e{\downarrow}$ is non-trivial, then there is a unique source for η under $\textit{wgen}_e$.

Proof By inspection of $\textit{wgen}_e$. If η is the equation $n_1 \stackrel{?}{=} n_2$, where $n_1, n_2 \in N_e$, then the equation $n_1 = \lambda\mathsf{var}(n_2) \in b_e{\downarrow}$ is the unique source of η. If $n_1 = \Box \rightarrow n_3$, then either

$n_1 = \lambda(\Box, n_3) \in b_e{\downarrow}$ or $n_3 = @(n_1, \Box) \in b_e{\downarrow}$, where $n_1, n_3 \in W_e \cup X_e$. Since the term graph T_e is acyclic, exactly one of these equations are in $b_e{\downarrow}$. The case when $n_1 = n_2 \to \Box$ is similar. $\dashv$

5.7 Non-unifiability and untypability of weakenings

We now examine the structure of non-unifiable sets of weak type equations. A system of weak type equations E is represented as an incomplete unification graph. We say that a system of weak type equations E is *unifiable* if the incomplete unification graph representing E is unifiable. The following property is easy to verify.

Lemma 5.7.1 *(Weakening and non-unifiability)*

Let E and E' be two systems of weak term equations such that $E' \sqsubseteq E$. If E' is non-unifiable, then E is non-unifiable.

Proof By Lemma 5.5.1, $G' \sqsubseteq G$ where G' and G are the unification graphs of E' and E respectively. Let R be the simulation relation between the vertices of G' and G. Hence for each unification path p' from u' to v' in G', where u', v' are vertices in G', there is a path p in G from u to v such that $u'Ru$ and $v'Rv$ and the labels of p and p' are equal. Therefore, if G' has a clash or a cycle witnessed by a path p' then G has a clash or a cycle witnessed by a path with the same label as that of p'. $\dashv$

It follows that a system of weak term equations E is unifiable if and only if the normal form of E is unifiable. A set of weak type equations E is *minimally non-unifiable under* $\sqsubseteq$ if E is non-unifiable, and for all $E' \sqsubset E$, E' is unifiable.

We now examine the form of minimally non-unifiable sets of weak type equations and the sets of syntax equations generating them. Let e be a closed Λ-expression with term graph $\langle W_e, X_e, b_e \rangle$ and canonical set of type equations E_e. Let *wsrc*$_e$ be a source function for *wgen*$_e$.

It is easy to verify that if E is minimally non-unifiable due to cyclicity, then the path obtained by ignoring the edges incident on $\Box$ variables in the unification graph of E is a unification cycle. Furthermore, each equation in E contributes two strict vertices of the cycle. Hence, every type equation in E is non-trivial and has the form $n_1 \stackrel{?}{=} n_2$, $n_1 \stackrel{?}{=} \Box \to n_2$, or $n_1 \stackrel{?}{=} n_2 \to \Box$, where $n_1, n_2 \in W_e \cup X_e$. By Lemma 5.6.1, it follows that if E is minimally non-unifiable, then every weak syntax equation in *wsrc*$_e[E]$ is non-trivial and has one of the following five forms: $n_1 = \lambda\mathsf{var}(n_2)$, $n_1 = \lambda(\Box, n_3)$, $n_1 = \lambda(n_2, \Box)$, $n_1 = @(n_2, \Box)$, or $\Box = @(n_2, n_3)$, where $n_1, n_2, n_3 \in N_e$

5.7.1 Untypability of weak syntax equations

Given an expression e with term graph $\langle W_e, X_e, b_e \rangle$, the notion of untypability of sets of weakenings of b_e can be defined in a manner analogous to the definition of untypability of sets of syntax equations that are subsets of b_e.

Definition 5.7.3: Let e be a Λ-expression whose term graph is $\langle W_e, X_e, b_e \rangle$. A set of weakenings $S \sqsubseteq b_e$ is *untypable* if *wgen*$_e[S]$ is non-unifiable. $S \sqsubseteq b_e$ is *minimally untypable* if S is untypable and for all $S' \sqsubset S$, *wgen*$_e[S]$ is unifiable. $\square$

The result of Lemma 5.4.1 about deriving minimally untypable syntax equation sets as sources of minimally non-unifiable subsets of E_e is valid if we replace the subset ordering $\subseteq$ with the lower power-domain ordering induced by the weakening partial order $\sqsubseteq$ on term equations.

Lemma 5.7.2 *(Minimally untypable sets of weak syntax equations)*

Let $e = \langle N_e, b_e \rangle$ be a decorated expression whose canonical set of type equations is E_e. Let wsrc$_e$ be a source function of e with respect to wgen$_e$. If $E \sqsubseteq E_e$ is a minimally non-unifiable antichain, then wsrc$_e[E]$ is minimally untypable.

Proof Let $S =$ *wsrc*$_e[E]$. S is a source for E because *wgen*$_e[S] = E$. To show that S is minimally untypable, we need to show that if $S' \sqsubset S$, then $E' =$ *wgen*$_e[S'] \sqsubset E$. That is, we need to show that $E' \sqsubseteq E$ and $E \not\sqsubseteq E'$. Clearly, $E' \sqsubseteq E$. Assume, for the sake of contradiction, that $E \sqsubseteq E'$. Since $S' \sqsubset S$, it follows that $S \not\sqsubseteq S'$. $S \not\sqsubseteq S'$ means that there is an $s \in S$ such that for all $s' \in S'$, s and s' are comparable implies $s' \sqsubset s$.

Let $y =$ *wgen*$_e(s)$. Since $E \sqsubseteq E'$, let $y' \in E'$ such that $y \sqsubseteq y'$. Let $y' =$ *wgen*$_e(s')$, where $s' \in S'$. There are two cases:

- s and s' are comparable: Then $s' \sqsubset s$. Hence $y' \sqsubset y$, since $wgen_e$ is strictly monotone. But this contradicts the assumption that $y \sqsubseteq y'$.

- s and s' are non-comparable. Since $S' \sqsubseteq S$, let $s'' \in S : s' \sqsubseteq s''$. Let $y'' = wgen_e(s'')$. Since $wgen_e$ is monotone, $y' \sqsubseteq y''$. Hence, we have $y \sqsubseteq y' \sqsubseteq y''$, which implies $y \sqsubseteq y''$. Since E is an antichain, it follows that $y = y''$ and hence $s' \sqsubseteq s'' = s$, contradicting the assumption that s and s' are non-comparable.

$\dashv$

Example 5.7.22: Continuing Example 5.3.18, where e is $\lambda_0\ x_1.\ (@_2\ (@_3\ x_5\ x_6)\ x_4)$, and the unification graph of Figure 5.4 E_e is cyclic, the cycle C_1 equal to $a_3 b_{31} a_6 a_5^{-1}$ consists of the path segments $a_3 b_{31}$, a_6 and a_5^{-1}. The first segment corresponds to the weakening $n_5 = n_6 \rightarrow \Box$ of the type equation $n_5 = n_6 \rightarrow n_3$. The second and third correspond to the type equations $n_6 = n_1$ and $n_5 = n_1$. Thus, the cycle C_1 corresponds to the minimally non-unifiable set of weak type equations:

$$
\begin{aligned}
n_5 &\stackrel{?}{=} n_6 \rightarrow \Box \\
n_6 &\stackrel{?}{=} n_1 \\
n_5 &\stackrel{?}{=} n_1
\end{aligned}
$$

The cycle C_1 is shown in Figure 5.8. The weakening $n_5 = n_6 \rightarrow \Box$ is generated from the weakening $\Box = @(n_5, n_6)$ of the syntax equation $n_3 = @(n_5, n_6)$ using the generating function *wgen*$_e$. The equations $n_6 = n_1$ and $n_5 = n_1$ are generated from the syntax equations $n_6 = \lambda\mathsf{var}(n_1)$ and $n_5 = \lambda\mathsf{var}(n_1)$, respectively. Thus the following set S_1 of weak syntax equations together generates the cycle C_1:

$$\begin{array}{rcllcl} \Box & = & @(n_5, n_6) & \mapsto & n_5 \stackrel{?}{=} n_6 \to \Box \\ n_6 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_6 \stackrel{?}{=} n_1 \\ n_5 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_5 \stackrel{?}{=} n_1 \end{array}$$

The term graph of S_1 is shown in Figure 5.8.

Similarly, the cycle C_2 equal to $a_3 b_{32} a_2 b_{21} a_4 a_5^{-1}$ and shown in Figure 5.9 corresponds to the minimally non-unifiable set of weakenings

$$\begin{array}{lcl} n_3 \stackrel{?}{=} n_4 \to \Box & \qquad & n_5 \stackrel{?}{=} \Box \to n_3 \\ n_4 \stackrel{?}{=} n_1 & & n_5 \stackrel{?}{=} n_1 \end{array}$$

The cycle C_2 is generated by the following set S_2 of syntax equation weakenings:

$$\begin{array}{rcllcl} \Box & = & @(n_3, n_4) & \mapsto & n_3 \stackrel{?}{=} n_4 \to \Box \\ n_3 & = & @(n_5, \Box) & \mapsto & n_5 \stackrel{?}{=} \Box \to n_3 \\ n_4 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_4 \stackrel{?}{=} n_1 \\ n_5 & = & \lambda\mathsf{var}(n_1) & \mapsto & n_5 \stackrel{?}{=} n_1 \end{array}$$

and is graphed in Figure 5.9.

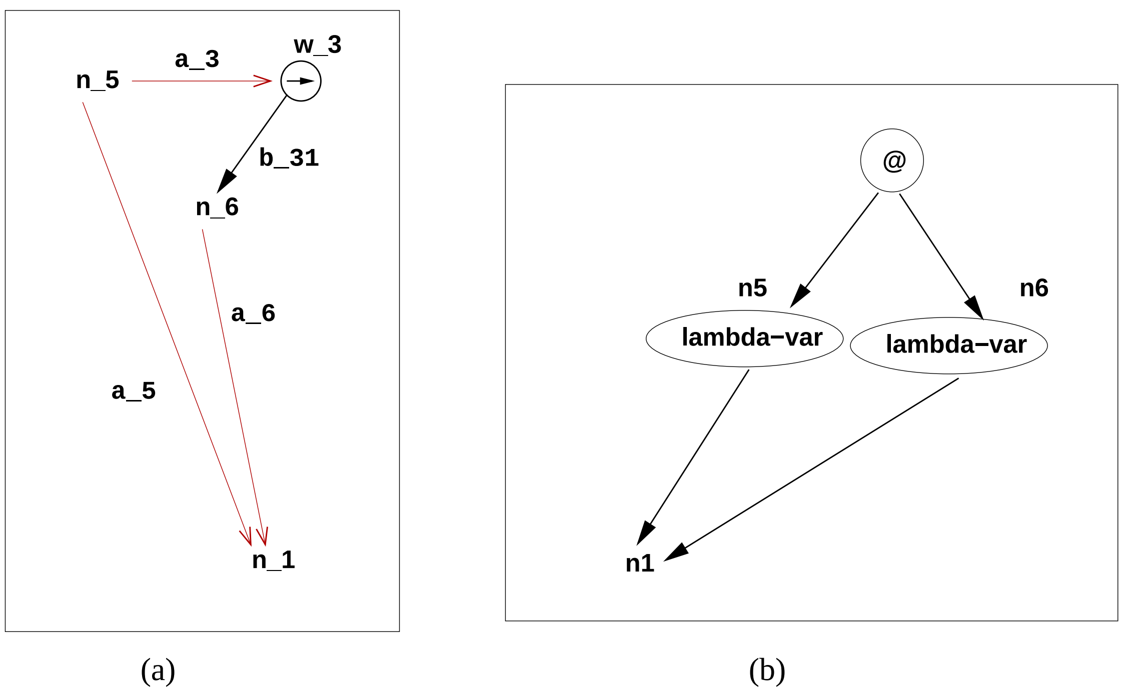
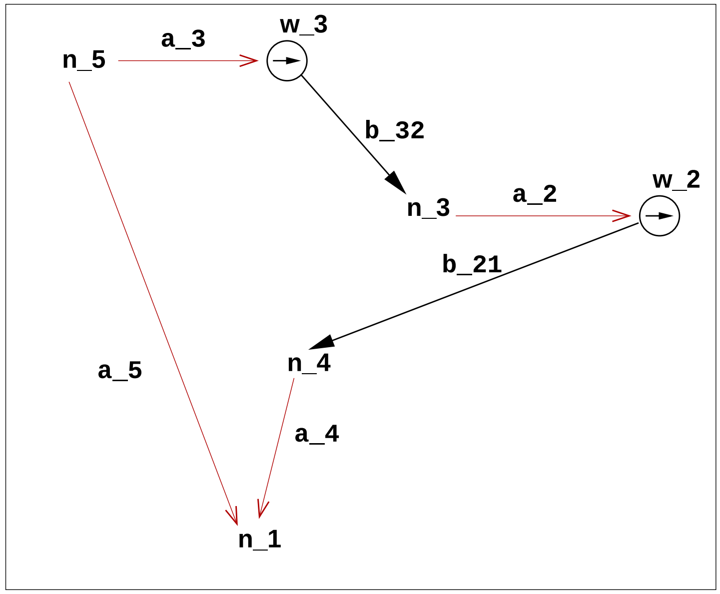
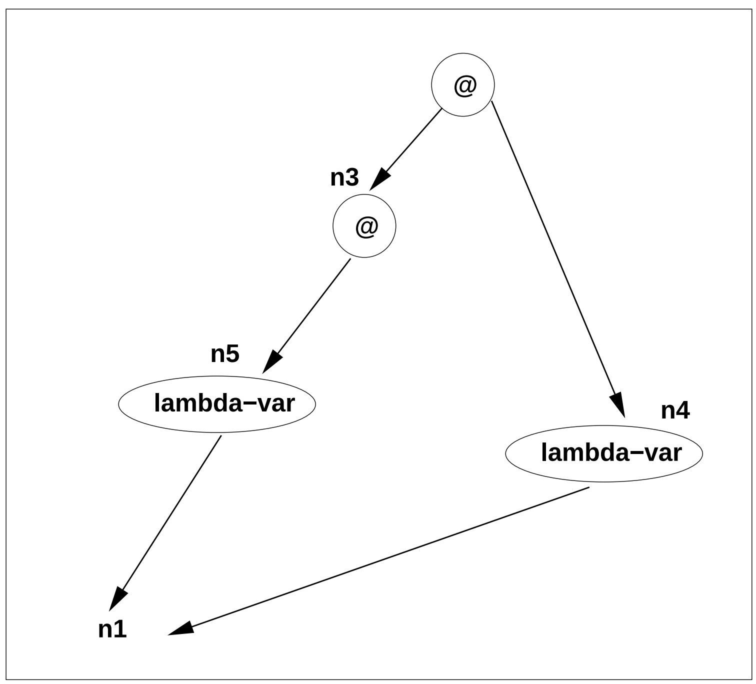


Figure 5.8: (a) The unification graph of the cycle C_1 defined in Example 5.7.22, and (b) the set of weakenings S_1 generating the cycle C_1 in Example 5.7.22 (the vertex labeled @ is anonymous).



(a)



(b)

Figure 5.9: (a): The cycle C_2 defined in Example 5.7.22; (b): Graph representing the set S_2 of weakenings generating the cycle C_2 in Example 5.7.22 (one of the vertices labeled @ is anonymous).

It is easy to verify by inspection that S_1 and S_2 are minimally untypable. $\diamond$

5.8 Properties of source functions

It is not clear whether generating functions for type systems other than the CH type system considered here will enjoy the property that every subsystem of its canonical system of type equations will have a source. There are two properties of generated type equations that are necessary for the existence of source functions. We discuss these properties next.

5.8.1 Locality

Let s denote a syntax equation and let *vars*(s) denote the set of non-$\Box$ variables occurring in s. The generating functions *gen*$_e$ and *wgen*$_e$ of the CH type system have the property that *vars*(*gen*$_e(s)) \subseteq$ *vars*(s) and *vars*(*wgen*$_e(s)) \subseteq$ *vars*(s). We call this the *locality property*. If locality is violated for some generating function on some domain and range, source information may not be available. For example, if g_e is the generating function

$$
\begin{array}{lllcll}
n_0 & = & f(n_1) & \mapsto & n_0 & \stackrel{?}{=} & n_1 \rightarrow n_2 \\
\Box & = & f(n_1) & \mapsto & \Box & \stackrel{?}{=} & n_1 \rightarrow n_2 \\
n_0 & = & f(\Box) & \mapsto & n_0 & \stackrel{?}{=} & \Box \rightarrow n_2 \\
\Box & = & f(\Box) & \mapsto & \Box & \stackrel{?}{=} & \Box \rightarrow n_2
\end{array}
$$

of some hypothetical type system, then *vars*$(g(n_0 = f(n_1))) \not\subseteq$ *vars*$(n_0 = f(n_1))$. The weakening $n_0 = n_1 \rightarrow \Box$ has no source. On the other hand, $n_0 = n_1 \rightarrow \Box$ has an imprecise source $n_0 = f(n_1)$, because $n_0 = n_1 \rightarrow \Box$ is weaker than $n_0 = n_1 \rightarrow n_2$, which is equal to $g(n_0 = f(n_1))$.

The locality property is not satisfied for type systems that involve the generation of "fresh" type variables, as in the Damas-Milner polymorphic type system [28]. Understanding the origin of these fresh (or non-local) variables is an additional complexity in the source-tracking of polymorphic type systems.

5.8.2 Linearity

A syntactic entity s is *linear* if there are no repeated occurrences of any non-$\Box$ variables, otherwise s is *non-linear*. For example, $f(n_1, n_1)$, and $n_1 = f(n_2, n_2)$ are non-linear whereas $f(n_1, n_2)$ and $n_1 = f(n_2, n_3)$ are linear. Each syntax equation in the term graph for a Λ-expression is linear. The generating functions considered in this chapter preserve the linearity of the entities on which they operate. Since each syntax equation generates a linear type equation, all the generated type equations in E_e and their weakenings, are linear entities.

If the linearity property is violated, precise source information may not exist. For example, if a generating function g for some type system maps a linear syntax equation s

equal to $n_0 = f(n_1)$ to a non-linear indexed type equation η given by $n_0 = n_1 \rightarrow n_1$, then the weakening $n_0 = n_1 \rightarrow \Box$ has only an imprecise source $n_0 = f(n_1)$.

It is possible for a type system to have rules that generate non-linear type equations as side conditions. This is certainly true in polymorphic type systems like Damas-Milner, where, for example, instantiating the type of a polymorphic identity function at a location n may yield the non-linear type equation $n = t \rightarrow t$, where t is a type variable. Again, non-linearity makes the tracking of source information more difficult.

5.9 Summary

In this chapter, we introduced a generating-function based framework for solving the problem of source-tracking for type inference. Generating functions for the Curry-Hindley type system (CH) were defined. We presented two different ways of associating source information with type constraints by considering different orderings between the domains and ranges of the generating function. We defined the notion of minimal untypability for sets of syntax equations and their weakenings, and showed how these sets can be extracted given minimally non-unifiable sets and weakenings of type equations and weakenings for the generating functions defined for CH. Finally, we considered how the properties of locality and linearity determine the applicability of these results to other type systems.

6

Conclusions and Future Work

The initial goal of this research was to build a system to facilitate the diagnosis of otherwise hard to comprehend type error messages in languages like ML. Implementations of some of the published and well-known type error diagnosis algorithms such as Wand [108], revealed that these algorithms were sometimes not effective in practice, probably because they lacked a sound formal basis. The formal basis for type inference is unification, and this thesis has analyzed the reasons for non-unifiability and tied this information back to the fragments of the program source associated with the non-unifiable equations.

6.1 Conclusions

We have shown how the debugging of term unification and type inference may be approached as exercises in proof construction in an appropriate logical system. By expressing the proof of non-unifiability as a context-free path in the unification graph, we have shown how optimal proofs of non-unifiability may be computed in polynomial time. On the other hand, construction of a single, not necessarily optimal proof can be done by a simple, yet practical extension of the standard unification algorithm. Since this proof may contain redundant inferences, a rewriting technique for simplifying proofs is introduced. The use of a rewriting technique simplifies the proofs generated by the diagnostic unification algorithm. Simplification of debugging information seems to be a novel application of rewrite systems.

As an application of diagnostic unification, this thesis presented a solution to the problem of tracking the source of type inference. This was based on a framework of generating functions mapping program source code to subsystems of type equations. The framework was applied to the problem of type inference in the Curry-Hindley system. The main insight developed here is that source-tracking is more accurately expressed as constraints between program subexpressions expressed as first-order logic predicates over vertices of the program syntax tree, rather than as a collection of subtrees or locations in a program.

There appear to be no substantial experimental studies comparing alternative approaches to unification failure diagnosis. Results are needed to quantify the utility of the approach outlined in this thesis in comparison with other approaches. One important experiment consists of comparing the size of proofs computed by the algorithms proposed in this thesis with that of the the shortest path algorithms suggested by Barret et al. [5], and algorithms for minimal proofs suggested in Cox [25], Chen [17], and Gandhe [38] for realistically sized unification problems. The simplicity of our proposed modification to the unification algorithm, both in terms of complexity considerations and ease of implementation, should be factored into any comparative analysis.

There are three directions in which the present research may be used or extended: immediate theoretical and practical extensions of the the current work, applications, and integration with automatic debugging.

6.2 Immediate Extensions

6.2.1 Damas-Milner Type Inference

An important extension to the framework for diagnosing Curry-Hindley type inference is the computation of source-tracking information for type inference in the Damas-Milner type system. The Damas-Milner type system [27, 28] is at the heart of the programming language ML [41, 77]. A distinguishing feature of ML is that it not only supports type inference at compile-time, but allows one to write *polymorphic* functions that can be re-used to operate on different data-types, a feature typically found in untyped functional languages like LISP. This is achieved by introducing a new variable-binding let construct with a variable, an expression and a body. The let-bound variable is assigned a type in such a manner that each reference to it in the body is associated with a fresh instance of the type of the variable. For example, in the ML program $\mathsf{let} f = \lambda x.x \ \mathsf{in} \ ...f(2)...f(\mathsf{true})$, the function f is used with two different types $\mathsf{num}{\rightarrow}\mathsf{num}$ and $\mathsf{bool}{\rightarrow}\mathsf{bool}$. Each of these types is an *instantiation* of the *polymorphic type* $\forall t.t{\rightarrow}t$ attributed to the binding occurrence of f. The polymorphic type $\forall t.t{\rightarrow}t$ is parameterized by the type variable t. Languages that permit the definition and use of functions with parameterized polymorphic types are said to support *parametric polymorphism* and their typing disciplines are described by *parametric polymorphic type systems*, or *polymorphic type systems* for short. Other languages with polymorphic type disciplines include Haskell [50, 86] and Miranda [101]. More recently, there have been proposals to add polymorphic types to Java [12, 98] and compile Standard ML to Java [73].

Type inference in the Curry-Hindley type system involves deriving and solving equational constraints between type variables associated with program locations. The Curry-Hindley type system allows a clear separation between the generation of the type constraints and their solution. Type inference in Damas-Milner is implemented by Milner's algorithm W [75]. The implementation of W, however, makes such a separation impossible because the generation of equations is dependent on the solution of a subset of previously generated equations. An important aspect of Damas-Milner type inference is the generalization and instantiation of type expressions. Generalization of a type expression τ in a type environment A results in the *polymorphic type* $\forall\vec{\alpha}\tau$, where $\vec{\alpha}$, the set of type variables in τ not occurring free in A, is called the set of *generic type variables* in τ relative to A.

The *non-generic variables of* $\forall\vec{\alpha}.\tau$ are the variables *vars*$(\tau) - \vec{\alpha}$. Instantiating a polymorphic type $\forall\vec{\alpha}.\tau$ consists of creating a copy of τ by replacing each generic variable α by a fresh variable α', and leaving each non-generic variable unchanged. The new type variables are not directly associated with any program location. There are two questions relevant to source tracking of polymorphic types. The first consists of locating parts of the program that contribute to a type variable being marked as generic or non-generic. The second consists of computing relevant source information to track the origin of freshly generated type variables.

We briefly summarize a plausible approach to source-tracking Damas-Milner type inference. Most of the details of this approach have been worked out and will be reported elsewhere in the future. The approach is based on exploiting the observation, noted by Mitchell [78], that the let construct of ML is primarily an abbreviation mechanism, and unfolding the let binding at each occurrence of a let-bound variable reference preserves the static (type) and dynamic (valuation) semantics of an ML expression (see Kanellakis et al. [56]). Thus type variables created in W by type-variable instantiation are exactly those type variables that are associated with the new locations created by unfolding of let-bindings at their point of reference. The expression obtained by unfolding all the let-bindings in an expression e is called the letvar normal form of e. letvar normal forms have no let-bound variable references and their type inference problem may be expressed using a trivial extension of the Curry-Hindley type system. It is not practical to solve the type inference problem of e by reducing it to its letvar normal form because the normalization could likely result in an exponentially larger program. However, the type equations generated for the letvar normal form of e can be understood without the machinery of polymorphic types and may therefore be considered a canonical simplification of the type inference problem of e. Source-tracking information may be expressed as a path embedding from the term graph of the type of an expression e computed by W to the unification graph of the set of canonical type equations generated by the letvar normal form $\overline{e}$ of e in the trivial extension of the Curry-Hindley system. The source-tracking algorithm tracks the origin of each type variable and equation generated by W to a unification path in the unification graph of the equations generated by the type inference of $\overline{e}$. The source-tracking algorithm unfolds only those parts of the let-binding that are necessary for generating the path in the canonical set of type equations for $\overline{e}$, thus preventing unnecessary unfolding.

6.2.2 Equational and Semi-unification

We have solved unification diagnosis for unification in the empty equational theory, which is denoted $\emptyset$-unification. There are two important generalizations of $\emptyset$-unification: equational unification and semi-unification. *Equational unification*, or E-unification is the problem of unifying terms modulo a set of equational identities E. For example, the terms $f(b, a)$ and $f(a, b)$, where f is a functor and a, b are constants, do not unify under

-unification, but do unify if we consider unification modulo commutativity (allowing use of $f(x, y) = f(y, x)$ as an equational axiom). E-unification is a natural generalization of $\emptyset$-unification and is used extensively in automated deduction and rewrite systems. Developing a general framework for diagnosing E-unification, or specific cases of E-unification (associativity and commutativity, for example) is an open issue.

The unification problem solves equations of the form $\tau \stackrel{?}{=} \tau'$ by finding a substitution s such that *apply*$(s, \tau) =$ *apply*(s, τ'). Given two terms τ and τ' we say that $\tau \leq \tau'$ if there is a substitution τ such that *apply*$(s, \tau) = \tau'$. A term inequation I is a pair $\tau \stackrel{?}{\leq} \tau'$. A substitution s is a *semi-unifier* for I if and only if *apply*$(s, \tau) \leq$ *apply*(s, τ'). A system of equations and inequations $\langle E, I \rangle$ is solved by a *semi-unifier* s if s is a unifier for each equation in E and a semi-unifier for each inequation in I. Type inference in the presence of polymorphic recursion is reducible to semi-unification [44, 57], which is undecidable. Construction of a logic for semi-unification is an open problem. Such a logic would provide a formal basis for source-tracking of type inference in the presence of polymorphic recursion.

6.3 Other Applications

Since unification is at the core of several applications, the source-tracking unification algorithm can form the basis of debugging and source information for a variety of applications. We briefly mention two applications: Prolog debugging and soft typing.

6.3.1 Source-tracking of Prolog programs

There have been multiple proposals for Prolog debugging systems [33, 82]. The state of a Prolog program includes the current set of choice points, which determines the set of term equations unified so far, and the trail of bindings made so far, which determines the solution to those equations. The Prolog debugging problem can then be cast as a unification diagnosis problem. The unification diagnosis information can then be applied to isolate a Prolog program slice from these two pieces of information.

6.3.2 Soft-typing

Soft-typing [37] is a framework used to indicate the certainty of some type errors and the possible presence of others in programs in latently typed languages like Scheme. Soft typing transforms the original program by inserting runtime checks at points where the type of the program can not be inferred. The resulting program conforms to the static typing regime of the soft typing system, while preserving the runtime behavior of the original program. For example, given the constructors:

$$\begin{aligned}\mathbf{suc} &: 0 + \mathsf{suc} \rightarrow \mathsf{suc}\\ \mathbf{cons} &: \forall\alpha.(\mathsf{nil} + \mathsf{cons}(\alpha)) \rightarrow \mathsf{cons}(\alpha)\end{aligned}$$

the expression

$$\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \mathbf{suc}(0)\ \mathsf{else}\ \mathsf{nil}$$

would be assigned the type

$$\mathsf{true}\ +\ \mathsf{false} \longrightarrow \mathsf{suc} + \mathsf{nil}$$

When the soft-typing type checker derives a certain type, it may be hard to comprehend why the type was inferred. Unification source-tracking could be used for deriving information about why, for example, a certain variant appears in the type of an expression.

6.4 Automated debugging

Error diagnosis is an important, but small, component of the larger problem of automating debugging, whose goal is automatic repair of erroneous code. In our framework it is left to the user, or another system, to use the diagnostic information to fix the ill-typed program in one of possibly many ways. Integrating the diagnostic information with the context information of the program, for example usage patterns of variables, should yield a more effective user-interface for automatic type debugging and program repair. As we saw in the examples, there could be competing diagnoses. Another possibility is a debugging framework that systematically generates a subset of possible diagnoses and aids choosing the most likely one (much like spell-checking programs that offer a variety of corrections to a misspelt word). Recent work by McAdam [72] addresses the problem of semi-automated repair of erroneous code. Integration of the source-tracking framework of this thesis with McAdam's approach is worth investigating.

Bibliography

[1] P. Aczel. *Non-wellfounded Sets*. CSLI, 1988.

[2] F. Baader and T. Nipkow. *Term Rewriting and all that*. Cambridge University Press, 1998.

[3] F. Baader and J. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programmaning*. Oxford University Press, 1993.

[4] H. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North Holland, Amsterdam, 1981.

[5] C. Barrett, R. Jakob, and M. Marathe. Formal language constraint path problems. *SIAM Journal of Computing*, 30:809–837, 2000.

[6] J. Barwise and L. Moss. *Vicious Circles*. CSLI, 1996.

[7] L. D. Baxter. An efficient unification algorithm. Technical report, University of Waterloo, 1973.

[8] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages*, 1994.

[9] K. Bernstein and E. W. Stark. Debugging type errors (full version). Unpublished Technical Report, Computer Science Dept. State University of New York at Stony Brook, Nov. 1995.

[10] Y. Bertot. Origin Functions in λ-calculus and Term Rewriting Systems. In *CAAP'92*, 1992. Springer Verlag LNCS 581.

[11] R. Boyer and J. S. Moore. The sharing of structure in theorem proving programs. *Machine Intelligence*, 7:101–116, 1972.

[12] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA 98*, pages 183–200, Vancouver, Canada, 1998.

[13] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[14] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1996.

[15] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[16] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the the ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 278–292, 1991.

[17] T. Y. Chen, J.-L. Lassez, and G. S. Port. Maximal unifiable subsets and minimal non-unifiable subsets. *New Generation Computing*, pages 133–152, 1986.

[18] O. Chitil. Compositional explantion of types and debugging of type errors. In *Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM Press, September 2001.

[19] N. Chomsky and M. Schutzenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 118–161. North-Holland, 1963.

[20] V. Choppella. Implementation of unification source-tracking. `http://www.cs.indiana.edu/hyplan/chaynes/unif.tar.gz`, July 2002.

[21] A. Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[22] A. Colmeraur. *Logic Programming*, chapter Prolog and Infinite Trees, pages 231–251. Academic Press, 1982.

[23] J. Corbin and M. Bidoit. A rehabilitation of robinson's unification algorithm. In R. E. A. Mason, editor, *Information Processing*, pages 909–914. Elsevier Science Publishers (North Holland), 1983.

[24] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[25] P. T. Cox. Finding backtrack points for intelligent backtracking. In J. Campbell, editor, *Prolog Implementation*, pages 216–233. 1984.

[26] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[27] L. Damas. *Type assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985.

[28] L. Damas and R. Milner. Principal type-schemes for functional languages. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.

[29] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[30] T. Dinesh. Type checking revisited: Modular error handling. In *International Workshop on Semantics of Specification Languages*, 1993.

[31] T. Dinesh and F. Tip. A case-study of slicing-based approach for locating type errors. In *Proc. 2nd International Conference on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, September 1997.

[32] T. B. Dinesh and F. Tip. A slicing-based approach for locating type errors. In *Proceedings of the USENIX Conference on Domain-Specific Languages (DSL'97), Santa Barbara, CA*, Oct 1997.

[33] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19(20):351–384, 1994.

[34] D. Duggan. Correct type explanation. In *Workshop on ML*, pages 49–58. ACM SIGPLAN, 1998.

[35] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[36] E. Eder. Properties of substitutions and unifiers. *Journal of Symbolic Computation*, 1:31–46, 1985.

[37] M. Fagan. *Soft Typing: An approach to type checking for dynamically typed languages*. Rice COMP TR92-184, Rice University, 1992.

[38] M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting type errors in the Curry system. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th conference on Foundations of Software Technology and Theoretical Computer Science*, pages 347–358, 1996.

[39] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[40] C. K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of the 17th ACM Symposium on Programming Languages*, 1990.

[41] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[42] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978.

[43] C. T. Haynes. Infer: a statically–typed dialect of Scheme: Preliminary tutorial and documentation. Technical Report 367, Dept. of computer science, Indiana University, 1992.

[44] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[45] J. Herbrand. *Recherches sur la The'orie de la Demonstration*. PhD thesis, University de' Paris, 1930. In French.

[46] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[47] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[48] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[49] W. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic*, pages 479–490. Academic Press, New York, 1980.

[50] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical report, Yale University, 1990. Superseded by [86].

[51] G. Huet. *Resolution d'equations dans les languages d'ordre 1,2,...,ω*. PhD thesis, University de' Paris, 1976. In French.

[52] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *European Association of Theoretical Computer Science*, 62:222–259, 1997.

[53] J. Jaffar. Efficient unification over infinite terms. *New Generation Computing*, pages 207–219, 1984.

[54] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Symposium on Programming Languages*, pages 44–57, 1986.

[55] J. P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule based survey of unification. In J. Lassez and G. Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 257–321. MIT Press, 1991.

[56] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In J. Lassez and G. Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[57] A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proceedings of the Fifteenth Colloquium on Trees in Algebra and Programming*, 1990.

[58] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–2311, April 1993.

[59] K. Knight. Unification: A multidisciplinary survey. *Computing Surveys*, 21(1):93–124, March 1989.

[60] J. Lamping. An algorithm for optimal lambda-calculus reductions. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM Press, 1990.

[61] J. Lassez, M. J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufmann, 1988.

[62] P. Le Chenadec. Normalization and linearity in unification logic. Technical Report 922, INRIA, October 1988.

[63] P. Le Chenadec. On positive occur-checks in unification. Technical Report 792, INRIA, January 1988.

[64] P. Le Chenadec. On the logic of unification. *Journal of Symbolic computation*, 8(1):141–199, July 1989.

[65] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages*, 20(4):707–723, July 1998.

[66] M. Marathe. personal communication, May 2002.

[67] A. Martelli and U. Montanari. Unification in linear time and space: A structured presentation. Technical report, B76-16, University of Pisa, 1976.

[68] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst.*, 4(2):258–282, April 1982.

[69] H. Maruyama, M. Matsuyama, and K. Araki. Support tool and strategy for type error correction with polymorphic types. In *Proceedings of the Sixteenth annual international computer software and applications conference, Chicago*, pages 287–293. IEEE, September 1992.

[70] B. McAdam. Generalising techniques for type debugging. In P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Trends in Functional Programming*, pages 49–57. Intellect, 2000.

[71] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, A. J. T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, September 1998 1998.

[72] B. J. McAdam. *Repairing Errors in Functional Programs*. PhD thesis, University of Edinburgh, 2002.

[73] B. J. McAdam, A. Kennedy, and N. Benton. Type inference in MLj. In *Trends in Functional Programming*, volume 2 of *Proceedings of the Scottish Functional Programming Workshop*, pages 159–171. Intellect, 2000.

[74] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. In C. Kirchner, editor, *Unification*, pages 457–488. Academic Press, 1990.

[75] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[76] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[77] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[78] J. C. Mitchell. Type systems for programming languages. In van Leeuwen et al., editors, *Handbook of Theoretical Computer Science*. North-Holland, 1991.

[79] J. H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, MIT, 1968.

[80] K. Mukai. A unification algorithm for infinite trees. In *Proceedings IJCAI-83*, 1983.

[81] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Intl. Symp. on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[82] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[83] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI-conference on theoretical computer science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[84] M. Paterson and M. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[85] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal for the ACM*, 28(2):233–264, April 1981.

[86] S. Peyton-Jones and J. Hughes (Eds.). Haskell 98: A non-strict, purely functional language, February 1999. `http://www.haskell.org/onlinereport`.

[87] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[88] T. Pietrzykowski. A complete mechanization of second-order logic. Technical Report CSSR 2038, Dept. of Applied Analysis and Computer Science, Univ. Waterloo, 1971.

[89] G. S. Port. A simple approach to finding the cause of non-unifiability. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 651–665. MIT Press, 1988.

[90] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[91] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[92] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 5:63–72, 1971.

[93] B. Russell. *Principles of Mathematics*. Rutlidge, 1996 edition published by Norton, W. W. and Company, Incorporated edition, 1903. ISBN 0393314049.

[94] W. Savitch. How to make arbitrary grammars look like context-free grammars. *SIAM Journal of Computing*, 2:174–182, 1973.

[95] D. A. Schmidt. *The structure of typed programming languages*. MIT Press, 1994.

[96] W. Snyder and J. Gallier. Higher-order unification revisited: Complete set of transformations. *Journal of Symbolic Computation*, 8(1& 2):101–140, 1989. Special Issue on Unification, part 2.

[97] H. Soosaipillai. An explanation based polymorphic type checker for Standard ML. Master's thesis, Heriot-Watt University, 1990.

[98] Sun Microsystems. Java specification request 14: Add generic types to the java programming language. Technical report, Java Community Process, `http://www.jcp.org/jsr/detail/14.jsp`, 2001.

[99] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215–225, 1975.

[100] F. Tip. *Generation of Program Analysis Tools*. PhD thesis, Institute for Logic, Language and Computation, CWI, Amsterdam, 1995.

[101] D. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *IFIP International Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[102] D. Turner. Enhanced error handling in ML. Technical report, Dept. of Computer Science, University of Edinburgh, 1990. CS4 Project.

[103] A. van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[104] A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special issue on automatic programming.

[105] M. Venturini-Zilli. Complexity of the unification algorithm for first-order expressions. *Calcolo*, 12(4):361–371, 1975.

[106] J. A. Walz. *Extending Attribute Grammars and Type Inference Algorithms*. PhD thesis, Cornell University, February 1989. TR 89-968.

[107] M. Wand. A semantic prototyping system. In *Proc. ACM SIGPLAN Symposium on Compiler Construction*, pages 213–221, 1984.

[108] M. Wand. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages.*, pages 38–43, January 1986.

[109] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[110] M. Weiser. *Program Slices: Formal, Psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[111] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[112] M. R. Wick and W. B. Thompson. Reconstructive expert system explanation. *Artificial Intelligence*, 54:33–70, 1992.

[113] J. Yang. Explaining type errors by finding the source of a type conflict. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming (Proceedings of the Scotting Functional Programming Workshop*, pages 49–57. Intellect, 1999.

[114] J. Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, 2001.

[115] J. Yang and G. Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.

[116] J. Yang, G. Michaelson, and P. Trinder. Helping students understand polymorphic type errors. In *Proceedings of 8th Annual Conference on the Teaching of Computing*, pages 11–19. LTSN Centre for Information and Computer Science, August 2000.

[117] J. Yang, G. Michaelson, and P. Trinder. Human-like explanations of polymorphic types. In S. Curtis, editor, *Scottish Workshop on Functional Programming*, pages 57–66, 2001.

[118] J. Yang, P. Trinder, G. Michaelson, and J. Wells. Improved type error reporting. In *Proceeding of Implementation of Functional Languages, 12th International Workshop*, pages 71–86, September 2000.

Curriculum Vitae

Venkatesh Choppella was born in Visakhapatnam, India on May 15th, 1964. He received his B.Tech. in Computer Science from the Indian Institute of Technology, Kanpur, India in 1985, his M.Tech. in Computer Science from the Indian Institute of Technology, Madras, India in 1987 and his M.S. in Computer Science from Indiana University, Bloomington in 1995. He has held research and engineering positions at Derivation Systems Inc., Xerox Corporation, Hewlett-Packard Company, and Indiana University.