Polymorphic Type Inference and Semi-Unification

Fritz Henglein[1]
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th floor
New York, N.Y. 10003, USA
Internet: henglein@nyu.edu

April 30th, 1989

[1]This is NYU/Courant Institute of Mathematical Sciences Technical Report 443, May 1989. It is a reformatted copy of a dissertation submitted to the Graduate School – New Brunswick, Rutgers, The State University of New Jersey, in partial fulfillment of the requirements for the degree of Doctor of Philosophy, Graduate Program in Computer Science. This research has been supported by the ONR under contract numbers N00014-85-K-0413 and N00014-87-K-0461.

Acknowledgements

I would like to thank my research advisor, Bob Paige, for his genuine personal concern for me, his academic interest in my progress, and his patience with a (possibly too) independent mind over many years.

I am also grateful to Eric Allender for his outgoing willingness to discuss and answer questions on complexity theory in great detail and his personal interest in my work; to Alex Borgida for keeping me informed on what the type theory community is up to; to Jiazhen Cai for applying his problem-solving skills and providing valuable ideas on how to approach the decidability of semi-unification; to Ernie Campbell for many discussions on types, type inference, type abstraction and for helping me make it to the Federal Express office by 9:37 p.m. on more than one occassion; to Corky Cartwright for helping me in more ways than he can probably imagine; to Ken Perry for sharing his insights into the complexity of unification-like problems with me; to Ed Schonberg for leaving me all the academic freedom in pursuing my work on the SETL Project that I could hope for while offering advice and encouragement; to Ravi Sethi for introducing me to the fascinating world of types in programming languages; to Matthew Smosna for being the best office mate this side of the Vistula and for providing me with endless supplies of scratch paper; and to Ann Yasuhara for being an authority on whose advice I could always count on.

I wish to thank my thesis committee, Eric Allender, Alex Borgida, Bob Paige, Ed Schonberg, Rick Statman, and Ann Yasuhara, for taking it upon themselves to give my thesis an intense and speedy scrutiny that is not easily matched.

Finally, it would have been impossible to accomplish this piece of work without the free access to the people and facilities I have enjoyed at Rutgers University, New York University, IBM T.J. Watson Research Center, and the financial support by the Office of Naval Research under contract numbers N00014-85-K-0413 and N00014-87-K-0461.

Abstract

In the last ten years declaration-free programming languages with a *polymorphic* typing discipline (ML, B) have been developed to approximate the flexibility and conciseness of dynamically typed languages (LISP, SETL) while retaining the safety and execution efficiency of conventional statically typed languages (Algol68, Pascal). These polymorphic languages can be type checked at compile time, yet allow functions whose arguments range over a variety of types.

We investigate several polymorphic type systems, the most powerful of which, termed Milner-Mycroft Calculus, extends the so-called **let**-polymorphism found in, e.g., ML with a polymorphic typing rule for recursive definitions. We show that semi-unification, the problem of solving inequalities over first-order terms, characterizes type checking in the Milner-Mycroft Calculus to polynomial time, even in the restricted case where nested definitions are disallowed. This permits us to extend some infeasibility results for related combinatorial problems to type inference and to correct several claims and statements in the literature.

We prove the existence of unique most general solutions of term inequalities, called most general semi-unifiers, and present an algorithm for computing them that terminates for all known inputs due to a novel "extended occurs check". We conjecture this algorithm to be uniformly terminating even though, at present, general semi-unification is not known to be decidable. We prove termination of our algorithm for a restricted case of semi-unification that is of independent interest.

Finally, we offer an explanation for the apparent practicality of polymorphic type inference in the face of theoretical intractability results.

Chapter 1

Introduction

1.1 Problem Background

Most programming languages provide the notion of *types* as their most fundamental abstraction from the unstructured universe of basic computer structures. While some languages perform type checking – checking for type consistent usage of program objects – at run-time (e.g., LISP, PROLOG, APL), others do it at compile-time (Pascal, Ada, ML, etc.). Doing it at compile time has the advantage that type errors, a common form of errors, are detected before the program is run. This usually comes at the price of cumbersome explicit type, variable and other declarations. Recently languages such as ML [32] have been designed that try to combine the safety of compile-time type checking with the flexibility of declaration-less programming by inferring type information from the program rather than insisting on extensive declarations. ML's type discipline allows for definition and use of *(parametric) polymorphic* functions; that is, functions that operate uniformly on arguments that may range over a variety of types.

A peculiarity in ML is that occurrences of a recursively defined function *inside* its definition body can only be used *monomorphically* (all of them have to have identically typed arguments and their results are typed identically), whereas occurrences *outside* its body can be used *polymorphically* (with arguments of different types). This thesis studies the computational implications for type inference in an extension of ML's typing system, which we primarily attribute to Mycroft [85], that treats recursively defined functions equally and uniformly inside and outside their bodies.

Although the motivation for studying Mycroft's extension to ML's typing discipline may seem rather esoteric and of purely theoretical interest, it stems from practical considerations. In ML many typing problems attributable to the monomorphic recursive definition constraint can be avoided by appropriately nesting function definitions inside the scopes of previous definitions. Since ML provides a form of polymorphic definition called **let**-polymorphism in most cases nesting definitions is, indeed, a workable scheme. Some languages, however, do not provide scoped nesting, but only top-level definition of functions. Consequently, all these definitions have to be considered, in general, as a single, mutually recursive definition. For example, B, SETL, and Prolog do not provide nested scopes. Adopting ML's monomorphic typing rule for recursive definitions in these languages would preclude polymorphic usage of any defined function inside any definition. In particular, since logic programs, as observed in [86], can be viewed as massive mutually recursive definitions, using an ML-style type system would eliminate polymorphism from strongly typed logic programming languages almost completely. Mycroft's extension, on the other hand, permits polymorphic usage in such a language setting.

In many cases it is possible to investigate the dependency graph ("call graph") of mutually recursive definitions and process its maximal strong components in topological order thus simulating polymorphically typed, nested **let**-definitions, but this is undesirable for several reasons:

1. The resulting typing discipline cannot be explained in a syntax-directed fashion, but is rather reminiscent of data-flow oriented reasoning. This runs contrary to structured programming and program understanding. For example, finding the source(s) of typing errors in the program text is made even more difficult than the already problematical attribution of type errors to source code in ML-like languages [51,119].

2. The topological processing does not completely capture the polymorphic typing rule. Mycroft reports on a mutually recursive definition he encountered in a "real life" programming project that could not be typed in ML, but could be typed by using the extended polymorphic typing rule for recursive definitions [85, section 8].

1.2 An Example

As an illustration of the monomorphic typing rule for recursive definitions consider the following standard definition of map and squarelist in Standard ML, taken directly from [85].

```
fun map f l = if null l then nil else f (hd l) :: map f (tl l)
    and
    squarelist l = map (fn x: int => x * x) l;
```

As it is written, this is a simultaneous definition of map and squarelist even though squarelist is not used in the definition of map. An ML-style type checker would produce the types

map: $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int\ list} \to \mathbf{int\ list})$
squarelist: $\mathbf{int\ list} \to \mathbf{int\ list}$

even though we would expect the type of map to be

map: $\forall\alpha.\forall\beta.(\alpha \to \beta) \to (\alpha\mathbf{list} \to \beta\mathbf{list})$,

which is the type produced by defining — sequentially — first map and then squarelist.

If we were to use map in another line of the same mutually recursive definition with an argument type different from **int list** we would even get a type error. This peculiarity comes from the fact that the Milner Calculus permits recursively defined functions to be used *monomorphically* only *inside* their bodies whereas they may still be used *polymorphically* — with arguments of different types — *outside* their bodies.

1.3 Outline of thesis

At the core of this thesis is a study of the type inference problem of ML's type system extended with a polymorphic typing rule, termed Milner-Mycroft Calculus here, and some of its relatives. Motivated by the well-known reduction of simple type inference to first-order unification we relate type inference calculi to unification-like problems that distill the combinatorial essence from the presentation of the typing problems. In particular, we show that *semi-unification* is at the heart of Milner-Mycroft-style type inference. Because of this central role, we study the algebraic and algorithmic aspects of semi-unification. Although semi-unification appears worthy of study on the merit of its fundamental character alone, we show that most of the results on semi-unification translate back to type inference and thus yield new results and new proofs of known results.

1.3.1 Simple type inference and unification

We expand on some work by Kanellakis and Mitchell [53] and give, in detail, a log-space reduction of first-order unification to simple type inference. This shows that simple type inference is log-space equivalent to unification; in particular, it is P-complete under log-space reductions. The encoding of first-order terms by λ-expressions is useful in later reductions.

1.3.2 Polymorphic type inference and semi-unification

Semi-unification is the problem of solving term inequalities, $M \leq N$, where $\leq$ is interpreted as the subsumption preordering on terms: $M \leq N \Leftrightarrow$ there is a substitution ρ such that $\rho(M) = N$. We present two polynomial-time reductions: from type inference in the Milner-Mycroft Calculus (and the Milner Calculus) to semi-unification, and from semi-unification to type inference in the Flat Milner-Mycroft Calculus, which is a (minimal) programming language with only top-level polymorphically typed recursive definitions. As corollaries we obtain that

1. semi-unification characterizes type inference in the Milner-Mycroft Calculus up to polynomial-time equivalence;
2. type inference in the Milner-Mycroft Calculus can be efficiently reduced to the case with only a single recursive definition and no other definitions (Flat Milner-Mycroft Calculus); this contradicts Mycroft's conjecture that the complexity of type inference depends exponentially on the degree of nesting of recursive definitions [85, p. 228];
3. Kanellakis and Mitchell's seminal result of PSPACE-hardness for the Milner Calculus [53] extends to the Flat Milner-Mycroft Calculus, solving a question posed by Kanellakis;
4. type inference in the programming language B [75] is no simpler than semi-unification and type inference in the Milner-Mycroft Calculus, and Meertens' uniformly terminating type inference algorithm [74] is incomplete in the sense that it indicates type errors for some typable B programs.

1.3.3 Algebraic structure of semi-unification

We show that strong equivalence, the standard formalization of "renaming of variables", does not adequately capture the structure of the solutions of semi-unification problems, thus correcting a statement by Chou [15]. A slightly weaker notion — weak equivalence — permits us to show that the set of solutions of any semi-unification problem form a complete lattice; in particular, there is always a most general solution (semi-unifier) unique up to weak equivalence if there exists a semi-unifier at all. As a corollary, the connection of polymorphic type inference and semi-unification yields a simultaneous proof of the principal typing property for the type systems we investigate.

1.3.4 Specification of most general semi-unifiers

Most general semi-unifiers exist and are unique modulo weak equivalence; we present a nondeterministic algorithm for computing the most general semi-unifier of any semi-unification problem. It contains an "extended occurs check" that eliminates all known cases that lead Mycroft's [85, section 6] and Meertens' [74, algorithm AA] type inference algorithms to nontermination. We *conjecture* that our algorithm terminates uniformly, thus implying decidability of the Milner-Mycroft Calculus and semi-unification, a currently open problem. This basic algorithm is described in three paradigmatic forms: as a functional, a rewriting, and a graph-theoretic program specification. All three are proved partially correct.

1.3.5 Efficient algorithm for uniform semi-unification

We study a space-efficient algorithm for uniform semi-unification, a provably decidable subclass of general (nonuniform) semi-unification. Kapur *et al.* have an elegant algorithm for deciding semi-unifiability in polynomial time. We present our own, independently devised, somewhat more complicated algorithm; it is less efficient, but computes a most general semi-unifier, in contrast to their decision algorithm.

1.3.6 Decidability — elementary approaches

We present some basic combinatorial properties of the graph-theoretic version of our basic semi-unification algorithm in the hope that some deeper investigation will eventually lead to estab lishing its uniform termination property. This seems appropriate to us since the "nonlocal" nature of the extended occurs check in our specifications suggests that combinatorial properties are stated most easily in a graph-theoretic setting.

1.3.7 Implications for practical programming languages

Beginning with the PSPACE-hardness result for the Milner Calculus there has been a gap between the theoretical infeasibility of polymorphic type inference and its observed practical success. This discrepancy appears even more pronounced in the Milner-Mycroft Calculus. We offer a tentative explanation of this gap in terms of resource-bounded typings, justified by the intent of typings as computational and conceptual abstractions of the computations of a program. If we impose the — as we think — reasonable restriction that the inferred type information must not be super-polynomially bigger than the size of the underlying programs, we can show that polymorphic type inference in the style of the Milner and Milner-Mycroft calculi are both practically *and* theoretically tractable.

Chapter 2

Implicitly Typed Lambda Calculi

The aim of our work is to study the principal aspects of type checking and type inference in programming languages, especially as they relate to parametric polymorphic features. To do this we shall use a language that contains only the features we are interested in so as to understand them independently of their possible interactions with other language features. This is not to say that other features are irrelevant or of less interest. In fact, operator overloading [52,118], implicit and explicit type coercions [103,78,27], abstract and dependent types [82,69,34], recursive types [110,70,77] and especially inclusion polymorphism [10,11,112,50,98,121,122]), a type-theoretic view of the behavior of object-oriented programming languages, are significant in the typing disciplines of modern strongly typed programming languages (e.g., [99,12]). But we cannot hope to combine several features and study their interactions, before we understand them individually. We refer the reader to [100] and [13] for an introduction and exposition of types and type checking in programming languages.

2.1 Untyped Lambda Calculus

We start with a simple functional language Λ, the *extended λ-calculus* [90], also called Exp in [23,85]. It has function abstraction, application, definition, and fixed point computation. We shall refer to it as the (untyped) λ-calculus even though the (pure) λ-calculus classically contains only function abstraction and application [3].

2.1.1 Syntax

The set Λ of λ-expressions (*expressions*) is defined by the following abstract syntax.

$$\begin{aligned} \mathrm{e} ::= {} & x \mid \lambda\ x.e \mid (ee') \mid \\ & \mathbf{let}\ x = e'\ \mathbf{in}\ e \mid \\ & \mathbf{fix}\ x.e \end{aligned}$$

where x ranges over a countably infinite set V of *variables.* In these productions λ, **let**, and **fix** bind x in e; **let** does *not* bind x in e', and application, denoted by juxtaposition, does not bind anything at all. A variable or variable occurrence in an expression e that is bound by λ is a *λ-bound* variable, respectively variable occurrence; same for **let** and **fix**. If a variable occurrence is not bound, it is *free.* A variable is free in a λ-expression e if it has a free occurrence in e. The convention for omitting parentheses is that application associates to the left, and application has higher precedence than any other construction. We may abbreviate $\lambda x_1.\lambda x_2.\dots\lambda x_k.e$ to $\lambda x_1 x_2 \dots x_k.e$ or $\lambda\vec{x}.e$ if $\vec{x}$ denotes the sequence $x_1 x_2 \dots x_k$.

λ-expressions will usually be denoted by the letter e and primed or subscripted versions of e; variables by x, y along with their sub- and superscripted variants.

2.1.2 Operational Semantics

Instead of encoding renaming of λ-bound variables by an explicit axiom of α-conversion (see, e.g., [42, definition 1.16]) we follow Barendregt [3] and write $e \equiv e'$ if e' is identical to e except that it may have some λ-bound variables systematically renamed. Every λ-expression is then understood as a representative of its $\equiv$-equivalent expressions, and all operations on λ-expressions are always defined on $\equiv$-equivalence classes. For $e, e' \in \Lambda, x \in V$, $e[e'/x]$ denotes the simultaneous replacement of all free occurrences of x in e by e'; as usual we assume that bound variables in e are renamed appropriately to avoid "capturing" free variables in e'. This is an acceptable convention with the proviso just made [3, p. 26].

The operational semantics of λ-expressions is defined as the reflexive, transitive, compatible[1] closure, $\xrightarrow{*}$, of the union of the following notions of reduction (see [3, chapter 3]).

$$\begin{array}{lll} (\lambda\, x.e)e' & \rightarrow_\beta & e[e'/x] \\ \mathbf{let}\, x = e' \,\mathbf{in}\, e & \rightarrow_{\mathbf{let}} & e[e'/x] \\ \mathbf{fix}\, x.e & \rightarrow_{\mathbf{fix}} & \mathbf{let}\, x = (\mathbf{fix}\, x.e)\,\mathbf{in}\, e \end{array}$$

In our examples we may sometimes add "constants" such as natural numbers with some arithmetic operators and the Boolean values with some logical operators to our λ-calculus. Whenever suitable we shall use infix notation for constant operations instead of prefix. We may tacitly assume the existence of suitable reduction relations, summarily called δ-reductions, that implement the usual semantics on those constants. Our theory is developed only for the "pure" λ-calculus, although — or because — it can easily be extended to include constants.

As an example of an expression with constants,

$$\mathbf{fix}\, f.\lambda x.\mathbf{if}\; x = 0 \;\mathbf{then}\; 1 \;\mathbf{else}\; x * f(x-1)$$

denotes the factorial function, and

$$\begin{array}{l} \mathbf{let}\, \text{fact} = \\ \quad \mathbf{fix}\, f.\lambda x.\mathbf{if}\; x = 0 \;\mathbf{then}\; 1 \;\mathbf{else}\; x * f(x-1)\;\mathbf{in} \\ \quad \text{fact } 5 \end{array}$$

reduces to 120 via $\xrightarrow{*}$.

Equality (β-equality), $=$, is the congruence relation generated by $\xrightarrow{*}$. As is well-known, for the untyped λ-calculus we could have dispensed with **let** and **fix** since they are both definable by abstraction and application alone:

$$\begin{array}{rcl} \mathbf{let}\, x = e' \,\mathbf{in}\, e & = & (\lambda x.e)e' \\ \mathbf{fix}\, x.e & = & Y(\lambda x.e) \end{array}$$

where $Y = \lambda f.WW$ and $W = \lambda x.f(xx)$ or $Y = W'W'$ and $W' = \lambda x.\lambda y.y(xxy)$. For the second definition of W we also have $Y(\lambda x.M) \xrightarrow{*} (\lambda x.M)(Y(\lambda x.M))$.

Nonetheless we shall keep **let** and **fix** forms since there are typed versions of the λ-calculus in which the above replacements are not possible since the right-hand sides may not necessarily satisfy the typing rules, which is to say that the sort of typing we shall consider is in general not closed w.r.t. equality).

[1]A relation R is compatible if it is closed under taking contexts; that is, $(e_1, e_2) \in R$ implies $(C[e_1], C[e_2]) \in R$ for any context $C[]$ surrounding e_1, respectively e_2.

2.2 Type Inference Systems

It is not easy to find a modern set-theoretic interpretation of the λ-calculus in which application is modeled by (set-theoretic) function application, and λ-abstraction is interpreted as the definition of a (set-theoretic) function. This is mainly due to the possibility of unbridled self-application, as in xx. Also, concerns over representation independence and type integrity in the design of programming languages lead to the introduction of typing disciplines that restrict the class of λ-expression that are considered acceptable (well-typed). We shall briefly present the mechanism for specifying various related typing disciplines.

2.2.1 Notational Prerequisites

The notational conventions used here are fairly standard. The reader familiar with [23] and [85] or any number of logically specified polymorphic type systems is encouraged to skip this subsection.

Type Expressions

The type expressions (*types*) are formed according to the following productions.

$$\begin{array}{rcl} \tau & ::= & \kappa \mid \alpha \mid \tau \rightarrow \tau \\ \sigma & ::= & \tau \mid \forall \alpha . \sigma \end{array}$$

where α ranges over an infinite set TV of *type variables* disjoint from V, and κ ranges over given primitive types, such as **integer, Boolean**, *etc*, and $\forall$ is a (type) variable binding operator. The distinction between free and bound variables (variable occurrences) in type expressions is as expected: all occurrences of $\forall$-bound variables are bound, all other occurrences are free. The type expressions M derivable from τ are the *monotypes*;[2] the type expressions Π derivable from σ are called *polytypes*.

For $\vec{\tau} = \tau_1 \tau_2 \ldots \tau_k$ we may write $\forall \tau_1 \tau_2 \ldots \tau_k . \tau'$ or $\forall \vec{\tau} . \tau'$ for $\forall \tau_1 . \forall \tau_2 . \ldots \forall \tau_k . \tau'$. The function type constructor, $\rightarrow$, is right-associative; that is, $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ should be parsed as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. For any type expression σ we write $\sigma[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]$ to denote the type expression resulting from simultaneously substituting τ_i for all free occurrences of $\alpha_i, 1 \leq i \leq k$, in σ.

Note that the $\forall$-quantifiers in polytypes can only appear as prefixes of type expressions, which is the critical difference from the Second Order λ-calculus [29,101].

The Greek letter τ always indicates a monotype, while the Greek letter σ signals a polytype, and letters from the beginning of the Greek alphabet stand for type variables. This is the same convention as in [23] and [85].

Type Assignments

A *type assignment* (or *type environment*) A is a mapping from a finite subset of V (variables) to Π (polytypes). Type assignments are mostly used to formulate assumptions about the types of variables occurring free in some expression under consideration. This is necessary since the type of an expression e depends, in general, on the types of variables occurring free in e. For given A we define

$$A\{x : \sigma\}(y) = \begin{cases} A(y), & y \neq x \\ \sigma, & y = x; \end{cases}$$

that is, the value of $A\{x : \sigma\}$ at x is σ, and at any other value it is identical to A. We say a type variable α occurs free in A if it occurs free in $A(x)$ for some x in the domain of A.

The capital letter A henceforth always denotes a type assignment.

[2]Note that, in contrast to [76] and [85] our monotypes can contain (necessarily free) occurrences of type variables.

Full name	Abbreviation	Acronym
Curry-Hindley Calculus	Hindley Calculus	CH
Damas-Milner Calculus	Milner Calculus	DM
Milner-Mycroft Calculus	Mycroft Calculus	MM
Flat Milner-Mycroft Calculus	Flat Mycroft Calculus	FMM

Figure 2.1: Names and abbreviations of typing calculi

Typings

Typings are the well-formed formulae (judgments) of our type calculi. A typing consists of three parts: a type assignment A, an expression e, and a type expression σ, written as $A \supset e : \sigma$. It should be read as "In the type environment A, the expression e has type σ". Of course, not all typings are acceptable. Acceptability is defined statically by derivability in inference systems.

2.2.2 The Hindley, Milner, Mycroft, and Flat Mycroft Calculi

We shall study four type inference systems: the *Curry-Hindley Calculus*, the *Damas-Milner Calculus*, the *Milner-Mycroft Calculus*, and the *Flat Milner-Mycroft Calculus*. Instead of using their full names we shall abbreviate them throughout by using only the second component of their compound names in running text or their acronym in derivations, tables, *etc.* (see Figure 2.1).

With the exception of the Flat Mycroft Calculus all typing calculi under consideration here share the fact that they are defined over the same class of programs (λ-expressions) and the same set of judgments (typings). Their only differences are that they do not have the same inference rules. Since they share several of their axiom and rule schemes, though, a list of all axioms and rules is given in Table 2.1. Table 2.2 shows which of the axioms and rules are present in which calculus, and which ones are not.

Let X = CH, DM, MM, FMM. We write $X \vdash A \supset e : \sigma$ if $A \supset e : \sigma$ is derivable in the Hindley Calculus (X = CH), the Milner Calculus (X = DM), the Milner-Mycroft Calculus (X = MM), or the Flat Milner-Mycroft Calculus (X = FMM). If X is clear from the context, we may simply write $A \supset e : \sigma$ to indicate that this typing is derivable in X. Let e be a λ-expression, and let X = CH, DM, MM, or FMM. We say e is *well-typed* or *typable* in X (or simply well-typed/typable, if it is clear with respect to which typed calculus) if there is a type environment A and a type expression σ such that $A \supset e : \sigma$ is derivable in X. The *typability* problem for X is the problem of deciding the set of all well-typed expressions in the X. We may often abbreviate "the typability problem for the X Calculus" to simply "the X Calculus" as in "The Hindley Calculus is log-space equivalent to unification". As we shall see below, every expression e typable in the X Calculus has a unique (modulo some simple equivalence) "principal" type expression, given a type assumption A, no matter what choice of X. The functional problem of computing the principal type or outputting an indication of untypability for given e, A will be called the *type inference* problem for the X Calculus.

The Hindley Calculus corresponds to a language without mandatory variable or parameter type declarations; yet every variable has exactly one monotype. This is in the spirit of conventional statically typed languages such as Pascal where every program variable and every procedure has a unique type. That type has to be declared within the program itself, in contrast to the Hindley Calculus.

The Milner Calculus encodes the polymorphism that results from the ability in languages such as ML [31,32], SPS [120], Miranda [117] to give **let**-bound variables x a parameterized type that is automatically and implicitly instantiated at all applied occurrences of x. Note that in the rule (FIX-M) the type

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over monotypes; σ, σ' over polytypes. The following are type inference axiom and rule schemes.

Name	Axiom/rule
(TAUT)	$A\{x : \sigma\} \supset x : \sigma$
(GEN)	$\dfrac{A \supset e : \sigma \quad (\alpha \text{ not free in } A)}{A \supset e : \forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e : \forall\alpha.\sigma}{A \supset e : \sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x.e : \tau' \to \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \to \tau \quad A \supset e' : \tau'}{A \supset (ee') : \tau}$
(LET-M)	$\dfrac{A \supset e : \tau \quad A\{x : \tau\} \supset e' : \sigma'}{A \supset \textbf{let}\ x = e\,\textbf{in}\ e' : \sigma'}$
(LET-P)	$\dfrac{A \supset e : \sigma \quad A\{x : \sigma\} \supset e' : \sigma'}{A \supset \textbf{let}\ x = e\,\textbf{in}\ e' : \sigma'}$
(FIX-M)	$\dfrac{A\{x : \tau\} \supset e : \tau}{A \supset \textbf{fix}\ x.e : \tau}$
(FIX-P)	$\dfrac{A\{x : \sigma\} \supset e : \sigma}{A \supset \textbf{fix}\ x.e : \sigma}$

Table 2.1: Type inference axioms and rules

Axiom/rule	CH	DM	MM	FMM
TAUT	$\surd$	$\surd$	$\surd$	$\surd$
GEN	$\surd$	$\surd$	$\surd$	$\surd$
INST	$\surd$	$\surd$	$\surd$	$\surd$
ABS	$\surd$	$\surd$	$\surd$	$\surd$
APPL	$\surd$	$\surd$	$\surd$	$\surd$
LET-M	$\surd$			
LET-P		$\surd$	$\surd$	
FIX-M	$\surd$	$\surd$		
FIX-P			$\surd$	$\surd$

The mark $\surd$ indicates the corresponding axiom/rule is present in the calculus in whose column it appears; blank space means it is not included. The Flat Mycroft Calculus is restricted to λ-expressions with no **let**-operator and with only one occurrence of a **fix**-operator, which must occur at top-level.

Table 2.2: The Hindley, Milner, Mycroft, and Flat Mycroft type inference calculi

associated with the (presumably) recursively defined x is a monotype.[3] This implies that, intuitively, all occurrences of x in a recursive definition **fix** $x.e$ are monomorphic; that is, they have the same monotype.

The Mycroft Calculus models a language such as Hope [8] that permits **fix**-bound variables (i.e., for the most part recursively defined functions) to have parameterized types that can be instantiated arbitrarily inside the scope of their definition. Hope will admit such polymorphically typed recursive definitions only at the top-level and requires explicit type declarations, whereas our Milner-Mycroft Calculus permits even nested polymorphically typed recursive definitions and does not require explicit declarations.

The Flat Mycroft Calculus has only λ-expressions of the form **fix** $f.e$ where e contains only variables, λ-abstractions, and applications, but no **let**- or **fix**-constructs. It adopts the polymorphic typing rule from the Mycroft Calculus for its sole recursive definition. We call it "flat" since no nesting of polymorphically typed definitions — as in the Milner Calculus (**let**-rule (LET-P)) and in the Mycroft Calculus (**let**-rule (LET-P) and **fix**-rule (FIX-P)) — is permitted. This essentially models polymorphic programming languages with only top-level definitions that are automatically mutually recursive, as in (Polymorphic) Prolog [86], B [75], or (Polymorphic) SETL [36].

In our calculi we have deliberately excluded programming language features that have a strong bearing on type checking, such as coercion, overloading, inclusion polymorphism, union types, dependent types; not to mention assignment, references, exceptions. Note also that the typing disciplines are *implicit*: there is no mention of types in the programs (λ-expressions) themselves, only in the typing statements about them. This is to say, ours is the "Curry viewpoint": types are properties of (untyped) programs. This is in contrast to the "Church viewpoint": types occur in programs and are instrumental in the definition of what constitutes the notion of (typed) program in the first place. More importantly, though, the programming language considered here has a fixed point constructor and is thus universal, similar to LCF, yet very much in contrast to many typed calculi that are of interest for the very absence of a general fixed point operator. This is the main reason why we do not refer to what we call the Hindley Calculus as Church's Typed λ-calculus. What we call Milner Calculus is called ML (by Kfoury *et al.* [58]), or more loosely **let**-polymorphism or Milner-style polymorphism. Since it is well-known that side-effects and pointers have an effect on the soundness of polymorphic typing disciplines [22,68,116], we prefer not to call this typing calculus ML, a concrete programming language with side-effects, pointers and several

[3]Remember that τ always stands for a monotype.

other features. For similar reasons Kfoury *et al.*'s ML$^+$ is our Mycroft Calculus.[4] The general rationale for our choice of names is that the calculi are named after researchers that are prominently associated with investigating their properties.

2.2.3 Properties of Typed Calculi

It is quite clear that the Milner-Mycroft Calculus is more powerful than the Milner Calculus, which in turn is more powerful than the Hindley Calculus; that is to say, every λ-expression typable in the Hindley Calculus is typable in the Milner Calculus, and every λ-expression typable in the Milner Calculus is typable in the Mycroft Calculus. Even stronger, the sets of derivable typings in each of these calculi are in a containment relation along the same lines. These inclusions of typable expressions are proper. Consider, for example, the expressions $e_0 \equiv \mathbf{let}\ x = \lambda y.y\ \mathbf{in}\ (xx)$ and $e_1 \equiv \mathbf{fix}\ f.\lambda x.(ff)$. The expression e_0 is typable in the Milner Calculus due to the rule (LET-P), but not in the Hindley Calculus; e_1 is typable in the (Flat) Mycroft Calculus due to rule (FIX-P), but not in the Milner Calculus. For example,

$$DM \vdash \{\} \supset \mathbf{let}\ x = \lambda y.y\ \mathbf{in}\ (xx) : \forall\alpha.\alpha \rightarrow \alpha$$
$$MM \vdash \{\} \supset \mathbf{fix}\ f.\lambda x.(ff) : \forall\alpha.\forall\beta.\alpha \rightarrow \beta$$

This shows that, indeed, the Hindley Calculus, the Milner Calculus, and the Mycroft Calculus form a hierarchy of properly more powerful typing disciplines. For completeness' sake we shall briefly touch upon results that show that the type systems we consider here are not just syntactic in nature, but interact with the semantics of λ-expressions in an orderly fashion.

Soundness

Milner [76] presents a formal denotational semantics for expressions and types that allows specification of a semantic notion of validity. A type system is said to be *sound* (with respect to Milner's semantics) if all typings derivable are also semantically valid. We present only the following theorem and refer the reader to [76], [23], [85] or [70] for an exposition of semantic issues only alluded to here.

Theorem 1 *1. The Milner Calculus is sound (with respect to Milner's semantics).*
2. The Milner-Mycroft Calculus is sound (with respect to Milner's semantics).

Proof:

1. See [23].
2. See [85].

It may be noted that the soundness of the Milner Calculus also follows immediately from the soundness of the Milner-Mycroft Calculus and the fact that the Milner-Mycroft Calculus subsumes the Milner Calculus.

Subject Reduction

None of our typing disciplines are semantically complete since the property of typability is not invariant under β-equality. For example, for $K \equiv \lambda x.\lambda y.x, I \equiv \lambda x.x$ the expression $(KI)(\lambda x.(xx))$ is not typable in any of the typing disciplines under consideration here, yet $(KI)(\lambda x.(xx)) = I$, and I is clearly typable. A *dynamically typed* language is a programming language with a nontrivial typing discipline that is invariant

[4] Compounding the potential for confusion is that Jategaonkar and Mitchell are investigating an object-oriented extension to ML, called ML++. They call their initial design in this direction ML+.

under equality. Examples are LISP (but not the pure λ-calculus), APL, and SETL. Every dynamically typed (universal) language has a necessarily undecidable typability problem in view of Scott's version of Rice's theorem [3, chapter 6.6]. This in fact necessitates run-time type checking, hence motivating calling it "dynamically typed" in the first place.

Even though our static typing disciplines are not invariant under equality, a slightly weaker, yet very desirable property holds.

Theorem 2 *(Subject reduction property)*
Let X = CH, DM, or MM. If $X \vdash A \supset e : \sigma$ and $e \xrightarrow{*} e'$, then $X \vdash A \supset e' : \sigma$.

Proof: See Curry and Feys [21] for X = CH. The proofs for DM and MM are simple generalizations of Curry and Feys's original proof.

This theorem expresses that once a λ-expression has been been found to have some type, reducing the expression will preserve that type. In particular, it is never possible to encounter an untypable intermediate result when evaluating (reducing) any typable expression.

Principal Typings

Note that there may be many different typings for a single expression. In this subsection we briefly summarize for our type systems what has been called the *principal typing* property: Given a type assignment A every expression that has a type under A has a unique *most general* type under A.

The *generic instance preordering* $\sqsubseteq$ between types is given by

$$\forall \alpha_1 \ldots \alpha_n . \tau \sqsubseteq \forall \beta_1 \ldots \beta_m . \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

for any monotypes $\tau_1, \ldots, \tau_n$ whenever every $\beta_i (1 \leq i \leq m)$ is not free in $\forall \alpha_1 \ldots \alpha_n . \tau$. The equivalence induced by $\sqsubseteq$ is simply renaming of $\forall$-bound type variables and is denoted by $\equiv$.

Let X = CH, DM, MM, or FMM Calculus. We say σ is a *principal type* for e under A in X if

$$X \vdash A \supset e : \sigma$$

and for any type σ' such that

$$X \vdash A \supset e : \sigma'$$

we have $\sigma \sqsubseteq \sigma'$. Clearly, principal types are unique modulo $\equiv$. If for every A and every e, e has a principal type under A (or has no type under A), then we say the whole calculus X has the *principal typing property.*

Theorem 3 *(Principal typing property)*
Let X = CH, DM, MM, or FMM Calculus. X has the principal typing property.

Proof: For CH, see [41,20]; for DM, [23]; for MM and FMM, [85].

It is easy to see that if a closed λ-expression e (a λ-expression without free variables) has a type σ under any type assignment A then it has type σ under the empty assignment $\{\}$, and vice versa. For this reason we can speak of *the* principal type of e (independent of any type assignment).

The *type inference problem* is the (functional) problem of computing a principal type for given A and e or flagging untypability. Of course, the (decision) problem of typability is trivially solvable once the type inference problem has been solved. The converse, though, is not necessarily true even though essentially all current type checking algorithms for our typing disciplines also compute, directly or indirectly, principal types.[5]

Note that even though all type systems under consideration here have the principal typing property, it may be that the principal type for an expression e in one calculus is different from the principal type in another (for fixed A). Consider, for example, the Standard ML definition of "map" and "squarelist" in the program example in chapter 1. In the Milner Calculus the principal type of "map" is the monotype $(\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int\ list} \rightarrow \mathbf{int\ list}$ whereas in the Mycroft Calculus it is $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta) \rightarrow \alpha\mathbf{list} \rightarrow \beta\mathbf{list}$. Of course, this presumes an encoding of the mutually recursive definition of "map" and "squarelist" and of the SML type constructor **list** into the λ-calculus and the language of our type expressions. This is difficult since lists are a recursive data type, but a simpler "pure" example illustrating the difference is

$$\begin{aligned} &\mathbf{fun}\ Ix = x\ \mathbf{and} \\ &\quad J = Iy_0 \end{aligned}$$

under the type assignment $A_0 = \{y_0 : \mathbf{int}\}$. There are standard ways for encoding tuples and mutually recursive definitions by single recursive definitions in the λ-calculus. The above SML program can thus be transformed into a single recursive definition e_0 in the λ-calculus,

$$\mathbf{fix}\ f.\lambda g.g(\lambda x.x)(f(\lambda x.\lambda y.x)y_0).$$

This expression, e_0, is typable under A_0 both in the Milner Calculus and in the Mycroft Calculus. The principal types, however, are $((\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int} \rightarrow \mathbf{int}) \rightarrow \mathbf{int}$ in the Milner Calculus and $\forall\alpha.\forall\beta.((\alpha \rightarrow \alpha) \rightarrow \mathbf{int} \rightarrow \beta) \rightarrow \beta$ in the Mycroft Calculus.

2.3 Background

The calculi we consider have appeared in the literature before, with some variations. Curry, Hindley and others investigated the properties of "functionality" of combinatory logic [21,84,20,41,4], which is essentially what we call the Hindley Calculus. The Milner Calculus, in its logical form as a typed λ-calculus, was investigated by Damas and Milner [23,22] on the basis of earlier work by Milner [76].

As early as in the late 70s Wadsworth reportedly worked on extending the well-known type inference algorithm W for the Milner Calculus [76] to capture the more general typing rule (FIX-P) in (what we call) the Mycroft Calculus, but apparently did not publish his work [67]. The polymorphic programming language B, [75] has an extended rule for typing recursive definitions analogous to Mycroft's. Meertens [74], who designed it unaware of ML's polymorphic type system, presents a uniformly terminating type inference algorithm for B. Since B has neither higher-order functions nor nested declarations, Meertens raised the question of whether type inference in what we call the Mycroft Calculus is decidable.[6] Exploring static typing for logic programming [86], Mycroft [85] investigated the properties of ML with an extended rule for recursive definitions that allows for polymorphically typed occurrences of the defined function in its body. He was able to show that the resulting calculus, which we have called the Mycroft Calculus, is sound with respect to Milner's [76] semantics and that the principal typing property of the Milner Calculus is preserved. The standard unification-based type inference algorithm is not complete for the extended calculus, though. Mycroft provided a semi-algorithm for computing principal typings, but he

[5]The fact that the uniform semi-unifiability algorithm of Kapur *et al.* [54] does *not* compute most general uniform semi-unifiers — the equivalent of principal types — can be viewed as a remarkable exception.

[6]In chapter 4 we shall see that Mycroft-style type inference is not greatly affected by the absence or presence of higher-order functions and nested definitions. Meertens' uniformly terminating type inference algorithm is syntactically incomplete in that it signals a type error for some programs that are type correct with respect to the typing rules for B (or, equivalently, with respect to his semi-algorithm AA).

left the computability of that question and the decidability of the calculus open. Leiß[64] gave an alternate type inference system for the Mycroft Calculus (along with an extension of polymorphic type inference to record-based subtyping) based on term inequalities with context conditions. The decidability of the Mycroft Calculus was specifically addressed by Kfoury *et al.* [58]. They showed that typability in the Mycroft Calculus can be reduced to a "Generalized Unification Problem (GUP)" [56], which is similar to Leiß's formulation of inequalities with context conditions, and embarked on showing that, if a GUP instance has a solution, it has a solution whose size can be bounded recursively as a function of the size of the input. The authors have reported a flaw in their proof, and the general decidability of the Mycroft Calculus remains open. Once proven this would give an essentially nonalgorithmic proof of decidability for the Mycroft Calculus.

Whereas the Milner-Mycroft Calculus only admits polytypes with universal quantifiers in prefix position only on the top level, the Second Order λ-calculus [101] relaxes this constraint and permits polytypes with nested quantifiers. In such a system the **let**-construct is unnecessary since the equivalent description of a **let**-expression in the pure λ-calculus is typable if and only if the **let**-expression itself is typable. Böhm [6] showed that *partial* type inference in the Second Order λ-calculus is undecidable whereas, interestingly, the decidability of *full* type inference for the same type system is has been open [65,28] since the inception of the Second Order λ-calculus. Girard's system F_ω [29] generalizes the 2nd Order λ-calculus to type expressions of arbitrary finite order. Pfenning [91] refined Böhm's result by showing that type inference in the n-th Order λ-calculus, F_n, is equivalent to n-th order unification [46,30]. The typable λ-expressions in the conjunction type discipline of [18] (see also [106,83]) are exactly the strongly normalizing (untyped) λ-expressions, which implies that type inference is undecidable. Nonetheless, the Second Order λ-calculus and the conjunction type discipline have had a direct influence on programming language design. The language LEAP employs is directly based on F_ω and employs partial type inference with satisfactory practical performance [92]. Reynolds' language Forsythe [99] makes use of a conjunction type discipline.

There are many more very powerful type systems whose immediate application is in proof theory. They exploit and extend the "types-as-propositions" (and "expressions-as-proofs") analogy [44] to formulate constructive proof systems. A sample of such systems is AUTOMATH [24], Martin-Löf type theory [73], the Calculus of Constructions [19], and LF [33].

Chapter 3

Semi-Unification: Basic Notions and Results

Semi-unification is the problem of solving sets of inequalities of the form $M_1 \leq M_2$ in the subsumption lattice of free first-order terms. The special case of solving single inequalities has found applications in proving nontermination of term rewriting systems [54] while the general case characterizes type inference in the Mycroft Calculus (see chapter 4). Since this problem does not seem to have attracted broad attention in computer science, in this chapter and chapter 5, we give a comprehensive treatment of its basic algebraic properties and contrast it with unification, the problem of solving term equations.

Unification and semi-unification deal with related problems. Unification addresses solving equations between free first-order terms while semi-unification tackles the more general question of solving systems of equations and inequalities[1] (*SEI's*) where inequalities, $M_1 \leq M_2$, between terms M_1 and M_2 refer to the subsumption preordering $\leq$ on terms.

In this chapter we introduce the basic machinery of semi-unification. In particular, section 3.1 describes terms and substitutions and their basic algebraic structure, and section 3.2 contains definitions of systems of equations and inequalities and their solutions, semi-unifiers, as well as some basic results. In chapter 4 we shall show why semi-unification is relevant to type inference, and in chapter 5 we investigate the algebraic structure of semi-unifiers, which in turn has reverberations on the structure of typings. Algorithms for computing most general semi-unifiers can be found in chapter 6, and some combinatorial properties of our basic semi-unification algorithm are in chapter 7.

3.1 The Algebraic Structure of Terms and Substitutions

In this section we define the objects of our universe of discourse, terms and substitutions, and investigate elementary aspects of their algebraic structure. The material is mostly extracted from [45], [26], and [63]; much of the material dates back to [93], [94], [102], and [46]. Some definitions and results appear to be new. Though simple refinements of standard concepts and results, they are useful in later sections.

[1]We find the prevalent terminology somewhat unfortunate. While there is a distinction between "equation" (something that is to be *solved*) and "equality" (something that *holds*), there is no corresponding distinction with "inequality" since the term "inequation" is not commonly used in the English language. Even worse, "inequality" gives no indication as to whether $\leq$ (*less-than-or-equal-to*) or $\neq$ (*not-equal-to*) is meant, and there is no standard linguistic mechanism for distinguishing between these two. The term "inequation" has popped up in the literature, but, since it is still uncommon, we will use "inequality" throughout. This also makes it possible, admittedly somewhat artificially, to distinguish our systems of equations and inequalities from the related, but different, systems of equations and inequations in [17] and [63].

3.1.1 Basic Definitions

Definition 1 *(Ranked alphabet, functors, constants, variables, terms)*

A (ranked) alphabet $\mathcal{A}$ *is a pair* (F, a) *where* F *is a nonempty, denumerable set whose elements are called* functors *and* $a : F \longrightarrow \mathcal{N}$ *maps every functor* f *to its* arity $a(f)$. *Functors with arity* 0 *are called* constants. $\mathcal{A}$ *is* linear *if all its functors have arity at most* 1, nonlinear *otherwise.*

A set of variables V *for* $\mathcal{A} = (F, a)$ *is an infinite denumerable set disjoint from* F.

The set of proper (first-order) terms $T(\mathcal{A}, V)$ *(or simply* T *whenever* A *and* V *are understood), where* V *is a set of variables for* $\mathcal{A}$, *consists of all strings derivable from* M *in the grammar*

$$M ::= x | f(\underbrace{M, \ldots, M}_{k\,times})$$

where f *is a functor from* $\mathcal{A}$ *with arity* k, *and* x *is any variable from* V. *The set of* (first-order) terms $T^{\Omega}(\mathcal{A}, V)$ *(or simply* T^{Ω}*) is* $T(\mathcal{A}, V)$ *with an additional distinguished element* Ω *called the* undefined term.[2]

Variables are usually denoted by u, v, x, y, z, constants by c, d, nonconstant functors by f, g, h, and terms by M, N, as well as by their respective subscripted and superscripted versions. To indicate the arity k of a functor f we may write $f^{(k)}$. With these conventions in place we shall omit the parentheses following constants appearing in terms since this cannot lead to any confusion.

Two terms $M_1, M_2 \in T$ are *equal*, denoted $M_1 = M_2$, if and only if M_1 and M_2 are identical as strings; e. g., $f(x, y) = f(x, y)$, but $f(x, y) \neq f(u, v)$. The special term Ω is equal to itself and no other term.

The distinction between linear and nonlinear alphabets is crucial since terms over a linear alphabet can have at most one variable whereas terms over a nonlinear alphabet can contain any number of variables. In a nonlinear alphabet it is always possible to "emulate" a functor of arbitrary arity. For example, $g^{(4)}(M, N, N, N)$ can be viewed as a term $f^{(2)}(M, N)$ with a fictitious binary functor $f^{(2)}$, and $h^{(2)}(h^{(2)}(M_1, M_2), M_3)$ can be interpreted as a term $f^{(3)}(M_1, M_2, M_3)$ with an "emulated" ternary functor $f^{(3)}$. In this sense we are justified in stipulating the existence of a functor f with any arity $k \geq 1$ without loss of generality in a nonlinear alphabet. Note that this is, of course, not possible with linear alphabets.

Definition 2 *(Substitutions)*

A proper (first-order) substitution *is a mapping from* V *to* $T(\mathcal{A}, V)$ *that is the identity on all but a finite subset of* V. *Every substitution* $\sigma : V \longrightarrow T(\mathcal{A}, V)$ *can be extended uniquely to* $\bar{\sigma} : T^{\Omega}(\mathcal{A}, V) \longrightarrow T^{\Omega}(\mathcal{A}, V)$ *by the equations*

$$\begin{aligned}
\bar{\sigma}(x) &= \sigma(x), \text{ if } x \in V \\
\bar{\sigma}(\Omega) &= \Omega \\
\bar{\sigma}(f^{(k)}(M_1, \ldots, M_k)) &= f^{(k)}(\bar{\sigma}(M_1), \ldots, \bar{\sigma}(M_k)).
\end{aligned}$$

The domain $\mathbf{dom}\,\sigma$ *of* $\sigma : V \longrightarrow T(\mathcal{A}, V)$ *is* $\{x \in V \mid \sigma(x) \neq x\}$. *The* canonical representation *of* σ *with* $\mathbf{dom}\,\sigma = \{x_1, \ldots, x_n\}$ *is* $\{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$.

The mapping $\omega_{\mathcal{A},V}$, *which maps all terms* $M \in T^{\Omega}(\mathcal{A}, V)$ *to* Ω, *is called the* undefined substitution. *The set of all proper substitutions is denoted by* $\mathcal{S}(\mathcal{A}, V)$ *(or simply* $\mathcal{S}$ *whenever* $\mathcal{A}$, *and* V *are understood from the context). The set of* (first-order) substitutions $\mathcal{S}^{\omega}(\mathcal{A}, V)$ *consists of* $\mathcal{S}(\mathcal{A}, V)$ *with the additional mapping* $\omega_{\mathcal{A},V}$.

[2] Of course we make the standard assumption here that neither $\mathcal{A}$ nor V contain Ω or any of the symbols '(', ')', or ','.

We shall omit the subscript from $\omega_{T(\mathcal{A},V)}$ below whenever A, V, and thus $T(\mathcal{A},V)$ are clear from the context. Similarly, we will identify, as is usual, every substitution σ with its extension $\bar{\sigma}$. In this chapter and chapter 5 substitutions are ranged over by $\rho, \sigma, \tau, \upsilon$ along with their sub- and superscripted variations. To avoid confusion with type expressions, in the other chapters they may also be denoted by letters R, S, U.

A substitution specifies the simultaneous replacement of some set of variables by specific terms. For example, for $\sigma_0 = \{x \mapsto u, y \mapsto v, u \mapsto y, v \mapsto x\}$ we have $\sigma_0(f(x,y)) = f(u,v)$. The undefined substitution maps everything to the undefined term; e. g., $\omega(f(x,y)) = \Omega$ and $\omega(\Omega) = \Omega$.

For $\sigma \in \mathcal{S}$, we will write $\sigma\mid_W$ for the substitution defined by

$$\sigma\mid_W(x) = \begin{cases} \sigma(x), & x \in W \\ x, & x \notin W \end{cases}$$

Furthermore, $\omega\mid_W = \omega$.

Clearly substitutions, if understood as acting on terms, are closed with respect to functional composition. The undefined term Ω and the undefined substitution ω are useful in providing a meaning for the dynamic notion of "failure" in unification and other applications. They also lead to a very satisfying algebraic structure of terms and substitutions (see theorems 4 and 16) in chapter 5.

3.1.2 Term Subsumption

Let $\mathcal{A}$ be an arbitrary, but fixed alphabet in this section, and let V be a set of variables for $\mathcal{A}$.

Definition 3 *(Subsumption, α-congruence)*

The preordering $\leq$ of subsumption[3] *on T^Ω is defined by*

$$M_1 \leq M_2 \Leftrightarrow (\exists \sigma \in \mathcal{S}^\omega)\, \sigma(M_1) = M_2$$

for any $M_1, M_2 \in T^\Omega$.

The congruence relation $\cong$ of α-congruence *on T^Ω is defined by*

$$M_1 \cong M_2 \Leftrightarrow M_1 \leq M_2 \wedge M_2 \leq M_1$$

for all $M_1, M_2 \in T^\Omega$. We write $M_1 < M_2$ if $M_1 \leq M_2$, but $M_1 \not\cong M_2$. For any $M \in T^\Omega$, $[M]$ denotes the equivalence class of M in T^Ω.

If $M_1 \leq M_2$ we say M_1 *subsumes* M_2; e. g., $f(x,y)$ subsumes $f(g(y),z)$ since for $\sigma_1 = \{x \mapsto g(y), y \mapsto z\}$ the equality $\sigma_1(f(x,y)) = f(g(y),z)$ holds. If $M_1 \cong M_2$ we say M_2 is an *α-variant* of M_1 and vice versa; e. g., $f(x,y)$ is an α-variant of $f(u,v)$.

Recall that a partial order $(L, \leq)$ is a *(complete) lower semi-lattice* if it has a greatest lower bound for every finite (finite or infinite) subset of L. It is a *(complete) upper semi-lattice* if it has a least upper bound for every finite (finite or infinite) subset of L. It is a *(complete) lattice* if it is both a (complete) lower semi-lattice and a (complete) upper semi-lattice [66]. Recall also that a partial ordering is *Noetherian* if it has no infinite descending chains $M_1 > M_2 > \ldots$ [45]. It is well-known that any Noetherian lower semi-lattice is a complete lower semi-lattice, and any complete (lower or upper) semi-lattice is already a complete lattice.

The preordering $\leq$ on T^Ω induces a partial order on the quotient set $T^\Omega/_{\cong} = \{[M] \mid M \in T^\Omega\}$, which we will also denote by $\leq$. The structure of terms with respect to subsumption is captured in the following theorem.

[3]Note that this definition follows [45] and [26], but is dual to the definition in [63].

Theorem 4 *1.* $(T^\Omega/_{\cong}, \leq)$ *is Noetherian.*

2. $(T^\Omega/_{\cong}, \leq)$ *is a complete lattice.*

Proof: See [45].

The least upper bound of a set of terms is called its *most general common instance*; its greatest lower bound is called its *most specific common anti-instance*. The theorem expresses that both most general common instance and most specific common anti-instance are unique modulo α-congruence. Finding the most general common instance of a pair of terms is a special case of the unification problem (disjoint variable case). Finding the most specific common anti-instance of a pair is the anti-unification problem [46,63]. A most general common instance of $\{f(x, g(y)), f(g(y), z)\}$ is $f(g(y), g(z))$, but also $f(g(u), g(v))$; a most specific common anti-instance is $f(s,t)$. Clearly, $[x] = V$ (x any variable) is the least element and $[\Omega] = \{\Omega\}$ is the greatest element in $T^\Omega/_{\cong}$.

The subsumption preorder can be extended to substitutions, but not in a unique fashion. Different notions and their implications are studied in chapter 5.

3.2 Systems of Equations and Inequalities and Semi-Unifiers

In this section we present basic definitions and properties of inequalities over the subsumption preordering of terms and their solutions.

Definition 4 *(System of equations and inequalities, nonuniform/uniform semi-unifier, unifier)*

A system of equations and inequalities (SEI) *is a pair* $S = (E, \mathcal{I})$ *where* $\mathcal{I} = (I_1, \ldots, I_k)$ *for some* $k \in \mathcal{N}$ *and* $E, I_1, \ldots, I_k$ *each consist of a set of pairs of terms from* T*, usually written in the form*[4]

$$\left\{\begin{array}{ccc} M_{11} & = & M_{12} \\ M_{21} & = & M_{22} \\ & \ldots & \\ M_{m1} & = & M_{m2} \end{array}\right\} \quad E$$

$$\left\{\begin{array}{ccc} N^1_{11} & \leq & N^1_{12} \\ N^1_{21} & \leq & N^1_{22} \\ & \ldots & \\ N^1_{n_1 1} & \leq & N^1_{n_1 2} \end{array}\right\} \quad I_1$$

$$\ldots$$

$$\left\{\begin{array}{ccc} N^k_{11} & \leq & N^k_{12} \\ N^k_{21} & \leq & N^k_{22} \\ & \ldots & \\ N^k_{n_k 1} & \leq & N^k_{n_k 2} \end{array}\right\} \quad I_k$$

A substitution σ *for which there exist* quotient substitutions[5] $\rho_1, \ldots, \rho_n$ *such that*[6]

[4]Note that "=" and "$\leq$" are only formal here.

[5]It is actually irrelevant whether ω is permitted amongst the ρ_i or not.

[6]Here the symbols = and $\leq$ denote their logical meanings.

$$\left.\begin{array}{rcl}\sigma(M_{11}) & = & \sigma(M_{12}) \\ \sigma(M_{21}) & = & \sigma(M_{22}) \\ & \ldots & \\ \sigma(M_{m1}) & = & \sigma(M_{m2})\end{array}\right. \quad (E)$$

$$\left.\begin{array}{rcl}\rho_1(\sigma(N^1_{11})) & = & \sigma(N^1_{12}) \\ \rho_1(\sigma(N^1_{21})) & = & \sigma(N^1_{22}) \\ & \ldots & \\ \rho_1(\sigma(N^1_{n_1 1})) & = & \sigma(N^1_{n_1 2})\end{array}\right. \quad (I_1)$$

$$\ldots$$

$$\left.\begin{array}{rcl}\rho_k(\sigma(N^k_{11})) & = & \sigma(N^k_{12}) \\ \rho_k(\sigma(N^k_{21})) & = & \sigma(N^k_{22}) \\ & \ldots & \\ \rho_k(\sigma(N^k_{n_k 1})) & = & \sigma(N^k_{n_k 2})\end{array}\right. \quad (I_k)$$

hold simultaneously[7] *is called a* (nonuniform) semi-unifier *of S. If $\rho_1 = \rho_2 = \ldots = \rho_n = \rho$ for some ρ, then σ is called a* uniform semi-unifier*, and if furthermore $\rho = \iota$, the identity substitution, then σ is called a* unifier*.*

S is solvable *if it has a semi-unifier other than ω.* $\mathbf{SU}(S)$ *is the set of semi-unifiers of S,* $\mathbf{USU}(S)$ *the set of its uniform semi-unifiers, and* $\mathbf{U}(S)$ *is the set of its unifiers.*

The special symbol $\Box$ is an additional SEI that has only ω for a unifier and for a (non)uniform semi-unifier; we call $\Box$ the (only) improper SEI.[8] *The set of all proper systems of equations and inequalities over alphabet $\mathcal{A}$ and variables V is denoted by $\Gamma(\mathcal{A}, V)$ (or simply Γ whenever $\mathcal{A}$ and V are understood from the context). $\Gamma(\mathcal{A}, V)$ with the additional improper SEI $\Box$ is denoted by $\Gamma^{\Box}(\mathcal{A}, V)$.*

Semi-unifiability is the decision problem of determining if a given SEI is solvable (has a proper semi-unifier). As we shall see in chapter 5, every solvable SEI has a most general semi-unifier that is unique up to an appropriate equivalence relation on substitutions. The term *semi-unification* refers to the (functional) problem of computing a most general semi-unifier of a given SEI or flagging non-semi-unifiability. Similarly, *uniform semi-unifiability* and *uniform semi-unification* as well as *unifiability* and *unification* are the decision, respectively, functional problems that correspond to finding uniform (proper) semi-unifiers and (proper) unifiers. Often we will be sloppy and use the term for the functional problem to also denote the decision problem.

A semi-unifier, in other words, is a solution to a given set of equations and inequalities where the inequalities are split into groups that "share" the same quotient substitution, but the quotient substitutions across different groups of inequalities can be different. A uniform semi-unifier additionally solves the inequalities in a "uniform" fashion[9], and a unifier solves the inequalities by making both sides equal. By definition, if an SEI has a unifier it has a uniform semi-unifier, and if it has a uniform semi-unifier it has a semi-unifier.

Clearly, for unifiers there is no need to distinguish between equations and inequalities, and we can view, in this case, an SEI $S = (\mathcal{E}, \mathcal{I})$ as a system of equations alone made up of $\mathcal{E} \cup \bigcup \mathcal{I}$.

It is well-known that a set of equations can be expressed by a single equation in the sense that the set of its solutions (unifiers) is identical to the set of solutions of the original set of equations. An analogous

[7] Here "=" denotes term equality.

[8] Note that there are proper SEI's that have only the improper ω as their sole semi-unifier.

[9] Note that $(\emptyset, \{\{x \leq c_1\}, \{x \leq c_2\}\})$ has a semi-unifier — the identity substitution ι — but no uniform semi-unifier.

result, with the same simple proof, holds for *uniform* semi-unifiers, but apparently not for *nonuniform* semi-unifiers.

Proposition 1 *The following statements are equivalent.*

1. *$\mathcal{A}$ is nonlinear.*
2. $\{\mathbf{U}(S) : S \in \Gamma(\mathcal{A}, V)\} = \{\mathbf{U}(S) : S \in \Gamma(\mathcal{A}, V), S = (\mathcal{E}, \mathcal{I}), |\mathcal{E}| \leq 1, |\mathcal{I}| = 0\}$
3. $\{\mathbf{USU}(S) : S \in \Gamma(\mathcal{A}, V)\} = \{\mathbf{USU}(S) : S \in \Gamma(\mathcal{A}, V), S = (\mathcal{E}, \mathcal{I}), |\mathcal{E}| \leq 1, |\mathcal{I}| \leq 1, (\forall I \in \mathcal{I})\, |I| = 1\}$
4. $\{\mathbf{SU}(S) : S \in \Gamma(\mathcal{A}, V)\} = \{\mathbf{SU}(S) : S \in \Gamma(\mathcal{A}, V), S = (\mathcal{E}, \mathcal{I}), |\mathcal{E}| \leq 1, (\forall I \in \mathcal{I})\, |I| = 1\}$

Proof:

Statements 2, 3, and 4 follow from 1 by "tupling". For given SEI S form term M_1 by tupling all the left-hand sides of S, and M_2 by tupling all the right-hand sides. Define $S' = (\{M_1 = M_2\}, \emptyset)$; this proves 2. For 3 and 4 proceed similarly by tupling both sides of equations and all inequalities separately, respectively by tupling equations and the groups of inequalities separately.

Each of 2, 3, and 4 individually imply 1, which indicates that the ability to "tuple" is instrumental in embedding the theories of semi-unifiers and unifiers in the above subclasses of systems of equations and inequalities. We only prove $3 \Rightarrow 1$, the other implications being very similar.

Assume $\{\mathbf{USU}(S) : S \in \Gamma(\mathcal{A}, V)\}$ and $\{\mathbf{USU}(S) : S \in \Gamma(\mathcal{A}, V), S = (\mathcal{E}, \mathcal{I}), |\mathcal{E}| \leq 1, |\mathcal{I}| \leq 1, (\forall I \in \mathcal{I})\, |I| = 1\}$ are identical. Consider the SEI $S_1 = (\emptyset, \{\{y_0 \leq x_1, y_0 \leq x_2\}\})$ for pairwise distinct y_0, x_1, x_2. Clearly $\sigma_1 = \{x_1 \leftarrow x_2\}$ is a semi-unifier of S_1, but ι is not. If we assume that no functor in $\mathcal{A}$ has arity greater than 1, we already know that all terms in $T(\mathcal{A}, V)$ have at most one variable occurrence. Thus if an inequality $M \leq N$ has a solution at all then there must be subterms M' and N' of M and N, respectively, such that $M' \leq N'$ has the same set of semi-unifiers as $M \leq N$ and either M' is a variable or N' is a variable or none of M, M', N, N' contains a variable. If M' is a variable then the identity substitution $\iota = \{\}$ is a semi-unifier, and if it is not, then σ_1 is not a semi-unifier of $M' \leq N'$, and, finally, if M' and N' contain no variable then either all substitutions are semi-unifiers of $M' \leq N'$ (including ι) or none are (excluding σ_1). This holds also in the presence of an additional term equation. Consequently there is no SEI with at most one equation and one inequality with the same set of semi-unifiers as S_1 under the assumption that $\mathcal{A}$ has no functor with arity greater than 1, and we can conclude that $\mathcal{A}$ must be nonlinear.

In view of this proposition, whenever working with nonlinear alphabets we could have defined systems of equations and inequalities to consist of at most one equation and a set of inequalities instead of *sets* of equations and *sets* of sets of inequalities . We have chosen the present formulation because it permits a slightly more natural reduction of type inference to semi-unification. Furthermore, we can give a simple specification for computing most general semi-unifiers by rewritings over our systems of equations and inequalities, but not so easily if we adopted the simpler definition.

Nonetheless, when investigating the structure of semi-unifiers over a nonlinear alphabet — as we shall do almost exclusively — we shall often make use of the possibility of "contracting" sets of equations and groups of inequalities into single equations and single inequalities. In this vein, we may often omit the set brackets for singleton sets; e. g., $(\{M = N\}, \{\{M_1 \leq N_1\}, \{M_2 \leq N_2\}\})$ may be written simply $(M = N, \{M_1 \leq N_1, M_2 \leq N_2\})$ or even $(M = N, M_1 \leq N_1, M_2 \leq N_2)$.

3.3 Previous Work on Unification and Semi-Unification

Unification is the problem (and informally also the process) of finding solutions to term equations of the form $\tau_1 = \tau_2$ where $\tau_1, \tau_2 \in T$. A solution of $\tau_1 = \tau_2$ is a substitution σ such that $\sigma(\tau_1) = \sigma(\tau_2)$.

Although Herbrand [39] and Prawitz [95] had already used unification algorithms, the utility of and interest in unification was essentially initiated by Robinson's novel resolution principle in theorem proving [105] at the heart of which was a unification algorithm.

Since then papers on unification as well as applications of unification have abounded. While Robinson's original algorithm took exponential time to compute the solutions, new representations and algorithms have been found (see, e. g., [89] and [72]) that achieve linear bounds on the computation time, and the unification problem has been found to be P-complete [112]. Unification is also investigated in term algebras that are subject to equational [109] or conditional-equational [48] laws such as associativity, commutativity, and idempotence. Several unification algorithms (e. g., [114], [7], or see [109]) for such term algebras have been presented. Kapur and Narendran [55] showed that most of these unification problems are NP-hard. Huet [47,46] investigated third- and higher-order unification and proved that it is recursively undecidable. Goldfarb [30] showed that second-order unification is also undecidable.

Unification has permeated the field of resolution-based and even non-resolution-based theorem proving [5]. With the identification of a subset of First Order Logic that is especially amenable to resolution theorem proving (Horn Clause Logic, c. f. [60]) unification plays an eminent role in logic programming languages such as Planner [40] and PROLOG [123,113].

A concise and clean treatment of the algebraic aspects of unification can be found in [63] or in [26]. A recent survey on unification is [59].

Semi-unification addresses the problem of solving inequalities of the form $\tau_1 \preceq \tau_2$ where $\tau_1, \tau_2 \in T$. A substitution σ is a solution to $\tau_1 \preceq \tau_2$ if there exists $\rho \in S$ such that $\rho(\sigma(\tau_1)) = \sigma(\tau_2)$.

Whereas classical unification has numerous well-known uses and applications, semi-unification and related problems have apparently only recently received attention. The question of finding proofs with a minimum number of proof steps in some classical logical systems can be reduced to unification-like problems, in particular also to semi-unification. This sort of question has been addressed by Parikh and Statman in the early 70's [111] and, recently, by Krajicek, Pudlák [96,61] and other proof theoreticians. Kapur *et al.* [54] observe that solvability of a single term inequality yields a sufficient condition for showing nontermination in term rewriting systems, and they trace the history of this connection back to [62]. Semi-unification [15,38,57], has been shown to be at the heart of type checking in implicitly typed polymorphic programming languages. Term inequalities have also been explored as a partial order theory for constraint logic programming [49,88] and, in general, as a form of "partial order programming" [87]. The decidability of uniform semi-unification (see chapter 3) is proved independently in [96], [54], and [38] (see also section 6.4). Another special case of semi-unification, in which any identifier may occur at most once in left-hand sides of term inequalities, is shown decidable in [57]. The decidability of general semi-unification is currently open.

Chapter 4

Equivalence of Mycroft Calculus and Semi-Unification

This chapter is divided into two main sections and one minor section. In the first section we show that the type inference problems in the Milner and the Mycroft Calculi can be reduced efficiently to semi-unification, the problem of solving systems of equations and inequalities over the subsumption preordering of first-order terms. As a by-product we also obtain the well-known reduction of the Hindley Calculus to unification. The main achievement of this reduction lies in showing that the prefix-quantified theory of type correctness in the Milner and Mycroft Calculi can be completely embedded in semi-unification, a strictly first-order concept. Similar reductions to some sort of inequality constraints have been found by Kfoury *et al.* [58] and by Leiss [64]. Their inequalities, however, carry context conditions that stem from type quantification, whereas our reduction is to inequalities that are completely "first-order": there are no implicit or explicit constraints on variables in equations and inequalities. This makes semi-unification an instance of the "Generalized Unification Problem" [56] in that all instances have trivial context conditions, namely none.

In the second section, we present the converse reduction. In fact we show that semi-unification can be efficiently reduced to the Flat Mycroft Calculus, a small subclass of the general Mycroft Calculus that admits at most one occurrence of the polymorphically typed **fix**-operator and no **let**-operator. This can be interpreted as follows:

> The difficulty of type inference is completely subsumed in a single polymorphically typed recursive definition. Neither (polymorphic) **let**-bindings nor nested **let**- and **fix**-bindings add anything to this problem (in contrast to a statement by Mycroft [85]).

This shows that the Mycroft Calculus, the Flat Mycroft Calculus, and semi-unification are polynomial-time equivalent. This equivalence has several consequences. It answers in the affirmative a question raised by Kanellakis whether the PSPACE-hardness result for the Milner Calculus [53] can be extended to the Flat Mycroft Calculus. Also, we obtain a log-space reduction of unification to typability in the Hindley Calculus, and as a consequence this shows that the Hindley Calculus is P-complete under log-space reductions. Furthermore, we feel justified in claiming that semi-unification is the "right" combinatorial problem to look at when investigating the algorithmic properties of Mycroft-style polymorphic type inference since it comes with minimal machinery (no quantification, no "syntax", no scoping), yet captures the Mycroft Calculus up to polynomial time.

Characterizations of type inference by inequality constraints involving quantified types in the Second Order λ-calculus have been given in [79,28]. The characterization of polymorphic type inference by semi-unification in this chapter has also been proved, independently, by Kfoury *et al.* [57]; in fact, they have extended it to include the Second Order λ-calculus limited to "rank 2"-derivations [65].

All reductions mentioned here refer to Karp-reductions; i.e., input transformations. Our reductions from type inference to semi-unification preserve not only the basic decision problem (typability), but also map the structure of typings to semi-unifiers. This connection is exploited in chapter 5 to transfer results about the structure of semi-unifiers back to typings. In particular, proof of existence of most general semi-unifiers can be interpreted as a simultaneous "algebraic" proof of the principal typing property for all of CH, DM, MM, and FMM.

4.1 Reduction of Typability to Semi-Unification

The reduction from the Mycroft Calculus to semi-unification has already been described in [38]. We present here a detailed presentation of that reduction with all details supplied that were omitted in [38]. We also correct two minor errors in [38].

4.1.1 Syntax Trees and Variable Occurrences

The first step of the reduction consists of labeling the nodes in the syntax trees of λ-expressions with monotypes. For this purpose we have to introduce some notions to formalize the concept of syntax tree, binding, free and bound occurrences, and so on. The machinery necessary to do this unfortunately encumbers the overall exposition of the material with heavy notation and a multitude of definitions. This is mostly due to the fact that the intuitively quite clear concept of an (variable or term) *occurrence* in an expression is difficult to formalize. Huet [45] defines occurrences in expressions by terms and integer sequences that specify a "path" from the "root" of that term to a subterm. We use a different presentation that makes the connection with the graph-theoretic image inherent in the term "syntax tree" precise.

Definition 5 *(Term graph)*

Let $\mathcal{A} = (F, a)$ be a ranked alphabet and let V be a set of variables for $\mathcal{A}$. A term graph *G over $\mathcal{A}$ and V is a quadruple (N, N_F, E, L) where N is a set, $N_F \subset N$, $E : N_F \rightarrow N^*$, $L : N \rightarrow F \cup V$, and the following conditions hold.*

1. *$L(N_F) \subset F$ and $(\forall n \in N_F, f \in F)\, L_F(n) = f \Rightarrow |E(n)| = a(f)$;*
2. *$L(N - N_F) \subset V$.*

The induced (directed) graph *of a term graph $G = (N, N_F, E, L)$ is defined as $G^I = (N(G), E(G))$ where $N(G) = N, E(G) = \{(n, n') : n \in N_F, n' \in N \mid E(n) = (\ldots, n', \ldots)\}$.*

The term graph G is acyclic *if its induced graph G^I is acyclic.*

The elements of N are called *nodes.* A node n is a *functor node* if $n \in N_F$, and it is a *variable node* if $n \in N - N_F$. The mapping E is called an *edge map*, and if $E(n) = (n_1, \ldots, n_k)$ then n is a *parent* of all $n_i, 1 \leq i \leq k$, and the n_i are the *children* of n. For $n \in N$, $L(n)$ is the *label* of n; L is called a *labeling.*

Term graphs are graphical representations of terms that encode the term/subterm structure explicitly in their edge maps. Their definition is necessarily complicated since their nodes are labeled and the out-edges of every node are ordered. The digraph induced by a term graph is just the information left if we ignore this particular "additional" structure.

In an acyclic term graph $G = (N, N_F, L, E)$ over $\mathcal{A}$ and V every node represents a unique term. This representation is given by the following mapping $[.] : N \rightarrow T(\mathcal{A}, V)$.[1]

$$[n] = \begin{cases} x, & n \in N - N_F, L(n) = x \\ f([n_1], \ldots, [n_k]), & n \in N_F, L(n) = f, E(n) = (n_1, \ldots, n_k). \end{cases}$$

Let $\mathcal{A}_\lambda = (\{\lambda, @, \mathbf{let}, \mathbf{fix}\}, \{\lambda \mapsto 2, @ \mapsto 2, \mathbf{let} \mapsto 3, \mathbf{fix} \mapsto 2\})$. Clearly, $\mathcal{A}_\lambda$ is an appropriate alphabet for representing λ-expressions as first-order terms. We can now define what a syntax tree for a λ-expression is: a special kind of term graph over A_λ. Since λ-expressions have variable-binding operators we also define some concepts we shall need later.

[1]Note that [.] is implicitly parameterized by G.

Definition 6 *(Syntax tree, free variable occurrences, bound variable occurrence map)*

We define the notions of syntax tree, *its* bindings *and* scopes, *and its* free variable occurrences (FVO) *and* bound variable occurrence map (BVOM) *by simultaneous induction on the structure of λ-expressions e.*

$e = x$ **(variable):** *Any one-node term graph T with $N = \{n\}$ and $L(n) = x$ is a syntax tree for e with root n.*

$$\begin{aligned} FVO_T &= \{n\}, \\ BVOM_T &= \{\}; \end{aligned}$$

n is not a binding.

$e = \lambda x.e'$**:** *If T' is a syntax tree for e' with root n' and T_x is a vertex-disjoint syntax tree for x with root n_x (and no other node) then the term graph T that is the union of T_x and T' with an additional node n and $L(n) = \lambda, E(n) = (n_x, n')$ is a syntax tree for e with root n.*

$$\begin{aligned} FVO_T &= FVO_{T'} - \{n' \in FVO_{T'} \mid L(n') = x\}, \\ BVOM_T &= BVOM_{T'} \cup \{n_x \mapsto \{n' \in FVO_{T'} \mid L(n') = x\}\}; \end{aligned}$$

n_x is a λ-binding*; its scope is $N(T')$ (the nodes in T').*

$e = e'e''$**:** *If T', T'' are vertex disjoint syntax trees for (e', e'') and with roots (n', n''), respectively, then the term graph T that is the union of T' and T'' with an additional node n and $L(n) = @, E(n) = (n', n'')$ is a syntax tree for e with root n.*

$$\begin{aligned} FVO_T &= FVO_{T'} \cup FVO_{T''}, \\ BVOM_T &= BVOM_{T'} \cup BVOM_{T''}. \end{aligned}$$

$e = \mathbf{let}\ x = e'\ \mathbf{in}\ e''$**:** *If T_x, T', T'' are vertex disjoint syntax trees for (x, e', e'') and with roots (n_x, n', n''), respectively, then the term graph T that is the union of T_x, T' and T'' with an additional node n and $L(n) = \mathbf{let}, E(n) = (n_x, n', n'')$ is a syntax tree for e with root n.*

$$\begin{aligned} FVO_T &= FVO_{T'} \cup (FVO_{T''} - \{n'' \in FVO_{T''} \mid L(n'') = x\}), \\ BVOM_T &= BVOM_{T'} \cup BVOM_{T''} \cup \\ &\quad \{n_x \mapsto \{n'' \in FVO_{T''} \mid L(n'') = x\}\}; \end{aligned}$$

n_x is a **let**-binding*; its scope is $N(T'')$.*

$e = \mathbf{fix}\, x.e'$**:** *If T_x, T' are vertex disjoint syntax trees for x, e' and with roots n_x, n', respectively, then the term graph T that is the union of T_x and T' with an additional node n and $L(n) = \mathbf{fix}, E(n) = (n_x, n')$ is a syntax tree for e with root n.*

$$\begin{aligned} FVO_T &= FVO_{T'} - \{n' \in FVO_{T'} \mid L(n') = x\}, \\ BVOM_T &= BVOM_{T'} \cup \{n_x \mapsto \{n' \in FVO_{T'} \mid L(n') = x\}\}; \end{aligned}$$

n_x is a **fix**-binding*; its scope is $N(T')$.*

BVOM is a finite (single-valued) mapping from nodes to finite sets of nodes, and thus it is treated notationally as a finite set of pairs $n \mapsto \{n_1, ..., n_k\}$. Since all syntax-trees for a λ-expression e are isomorphic (i.e., there is a bijection between nodes that transforms any one syntax tree for e into any other syntax tree for e) we shall denote by $T(e)$ a canonical syntax tree for e and by $N(e)$ the (variable and functor) nodes in $T(e)$.

It is easy to see that our syntax trees are indeed trees: The induced digraph $T^I(e)$ is acyclic, and every node has at most one inedge, and exactly one node — the root — has no inedge.

4.1.2 Syntax-Oriented Type Inference Systems

The inference systems that describe our typing calculi are not syntax-oriented. This means that for a given expression e there may be several proof steps in a derivation that are not compositional in terms of the syntax of λ-expressions. This is solely due to the rules (INST) and (GEN) (see Figure 2.1 in chapter 2) since proof steps involving any one of these rules do not change the expression in a typing. In a syntax-oriented system a derivation for expression e has essentially the same tree structure as a syntax tree for e. The advantage of a syntax-oriented inference system is that we can think of a derivation for e as an attribution of the syntax tree $T(e)$.

In this subsection we present equivalent syntax-oriented type inference systems for CH, DM, MM, and FMM. In the next subsection we show how every derivation in these syntax-oriented inference systems can be translated into an attribution of $T(e)$ that satisfies certain properties, and vice versa.

The syntax-oriented versions of CH, DM, MM, FMM will be denoted by a "prime": CH', DM', MM', FMM'. In general if X is any one of CH, DM, MM, FMM, then X' is the corresponding syntax-oriented version of X. The list of all axiom and rule schemes that occur in the syntax-oriented inference systems is given in Figure 4.1. Table 4.2 shows which of the axioms and rules are present in which syntax-oriented calculus, and which ones are not.

For completeness we have included those rules that are unchanged from the original inference systems. Changed axioms and rules are marked with a "prime" ($'$). We have taken some liberties in our notation; in particular the sequence $\vec{\alpha} = \alpha_1 \ldots \alpha_n$ may also be regarded as a set.

Note that the syntax-oriented inference systems do not contain either (INST) or (GEN). The ability to instantiate polytypes to monotypes has been included into the new axiom, (TAUT'), for variables; and the ability of (GEN) to form polytypes is localized in applications of the polymorphic typing rules (LET-P') and (FIX-P'). An additional benefit of the syntax-oriented versions is that derivable typings are exclusively of the form $A \supset e : \tau$ where τ is a monotype. This is one step in the direction of eliminating constraints involving quantified types. Somewhat paradoxically this corresponds to traversing chronologically backwards the evolution of the Milner Calculus from the type system with explicitly quantified type expressions [23] to Milner's original "implicit" distinction of generic and nongeneric type variables [76].

We shall now prove that the new inference systems are indeed no weaker (or stronger) than the original systems. First we will need a technical proposition, though.

Proposition 2 *Let X be CH, DM, MM, or FMM. For any type environment A, λ-expression e, type expressions σ, σ', and type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_k, \vec{\alpha'} = \alpha'_1 \ldots \alpha'_k$ we have*

1. $X \vdash A \supset e : \forall\vec{\alpha}.\sigma \Leftrightarrow X \vdash A \supset e : \forall\vec{\alpha'}.\sigma[\vec{\alpha'}/\vec{\alpha}]$
2. $X \vdash A\{x : \forall\vec{\alpha}.\sigma\} \supset e : \sigma' \Leftrightarrow X \vdash A\{x : \forall\vec{\alpha'}.\sigma[\vec{\alpha'}/\vec{\alpha}]\} \supset e : \sigma'$

Theorem 5 *Let X = CH, DM, MM, or FMM. For any type environment A, λ-expression e, type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_k$ not free in A, and monotypes τ we have*

$$X \vdash A \supset e : \forall\vec{\alpha}.\tau \Leftrightarrow X' \vdash A \supset e : \tau$$

Corollary 3 *For any $e \in \Lambda$, e is typable in X if and only if it is typable in X'.*

For X = DM this theorem is similar to theorem 2.1 in [16]; and for X = MM it is almost identical to proposition 2.1 in [58]. Note, however, that it is technically a little bit stronger since it states that the type of e is literally identical in its quantifier-free part, without necessitating a renaming of type variables. Similar proofs, localizing applications of the INST rule at the leaves (variables) of λ-expressions can be found in [80] and [9].

Let A range over type environments; x over variables; e, e' over λ-expressions; $\vec{\alpha}$ over sequences of type variables; τ, τ' over monotypes, and $\vec{\tau}$ over sequences of monotypes. The following are type inference axiom and rule schemes for CH', DH', MM', FMM'.

Name	Axiom/rule
(TAUT')	$A\{x : \forall\vec{\alpha}.\tau\} \supset x : \tau[\vec{\tau}/\vec{\alpha}]$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x.e : \tau' \to \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \to \tau \quad A \supset e' : \tau'}{A \supset (ee') : \tau}$
(LET-M)	$\dfrac{A \supset e : \tau \quad A\{x : \tau\} \supset e' : \tau'}{A \supset \mathbf{let} x = e \mathbf{in} e' : \tau'}$
(LET-P')	$\dfrac{A \supset e : \tau \quad A\{x : \forall\vec{\alpha}.\tau\} \supset e' : \tau' \quad (\vec{\alpha} \text{ not free in } A)}{A \supset \mathbf{let} x = e \mathbf{in} e' : \tau'}$
(FIX-M)	$\dfrac{A\{x : \tau\} \supset e : \tau}{A \supset \mathbf{fix} x.e : \tau}$
(FIX-P')	$\dfrac{A\{x : \forall\vec{\alpha}.\tau\} \supset e : \tau \quad (\vec{\alpha} \text{ not free in } A)}{A \supset \mathbf{fix} x.e : \tau[\vec{\tau}/\vec{\alpha}]}$

Table 4.1: Syntax-oriented axioms and rules

Axiom/rule	CH'	DM'	MM'	FMM'
TAUT'	$\surd$	$\surd$	$\surd$	$\surd$
APPL	$\surd$	$\surd$	$\surd$	$\surd$
ABS	$\surd$	$\surd$	$\surd$	$\surd$
LET-M	$\surd$			
LET-P'		$\surd$	$\surd$	
FIX-M	$\surd$	$\surd$		
FIX-P'			$\surd$	$\surd$

The mark $\surd$ indicates the corresponding axiom/rule is present in the calculus in whose column it appears; blank space means it is not included. The Flat Mycroft Calculus is restricted to λ-expressions with no **let**-operator and with only one occurrence of a **fix**-operator, which must occur at top-level.

Table 4.2: The syntax-oriented versions of the Hindley, Milner, Mycroft, and Flat Mycroft type inference calculi

Somewhat unsurprisingly, the theorem is a consequence of a stronger lemma that can be shown by structural induction on derivations.

Lemma 4 *Let X = CH, DM, MM, or FMM. For any type environment A, λ-expression e, type variables $\vec{\alpha} = \alpha_1 \dots \alpha_k$, $\vec{\alpha}$ not free in A, and monotypes τ we have*

$$X \vdash A \supset e : \forall \vec{\alpha}.\tau \Leftrightarrow (\forall \vec{\tau} \in M^k)\, X' \vdash A \supset e : \tau[\vec{\tau}/\vec{\alpha}]$$

Proof:

⇒: We proceed by structural induction on MM-derivations. The other cases, CH and DM, are simplifications of this proof; FMM is a subcase of MM.

(TAUT) If we have a trivial derivation involving only (TAUT) in MM,
$A\{x : \forall \vec{\alpha}.\tau\} \supset x : \forall \vec{\alpha}.\tau$
then, by (TAUT') in MM' we have
$A\{x : \forall \vec{\alpha}.\tau\} \supset x : \tau[\vec{\tau}/\vec{\alpha}]$

(ABSTR), (APPL) Trivial.

(INST) If $A \supset e : \forall \alpha_2 \dots \alpha_k.\tau[\tau_1/\alpha_1]$ is proved in MM invoking the (INST) rule,

$$\frac{A \supset e : \forall \vec{\alpha}.\tau}{A \supset e : \forall \alpha_2 \dots \alpha_k.\tau[\tau_1/\alpha_1]}$$

then, since we may assume by proposition 2 that α_1 is free in A we have, by the induction hypothesis, that the conclusion

$$A \supset e : \tau[\tau_1/\alpha_1][\tau_2/\alpha_2, \dots, \tau_k/\alpha_k]$$

is derivable in MM', for any $\tau_2, \dots, \tau_k$ since

$$\tau[\tau_1/\alpha_1][\tau_2/\alpha_2, \dots, \tau_k/\alpha_k] = \tau[\tau_1'/\alpha_1, \dots, \tau_k'/\alpha_k]$$

for some $\tau_1', \dots, \tau_k'$.

(GEN)

If $A \supset e : \forall \vec{\alpha}.\tau$ is proved in MM with the (GEN) rule,

$$\frac{\begin{array}{l} A \supset e : \forall \alpha_2 \dots \alpha_k.\tau \\ \quad (\alpha_1 \text{ not free in } A) \end{array}}{A \supset e : \forall \vec{\alpha}.\tau}$$

then, since $\vec{\alpha}$ is not free in A by assumption, we have that $A \supset e : \tau[\vec{\tau}/\vec{\alpha}]$ is provable in MM', by the induction hypothesis.

(LET-P) Assume $A \supset \textbf{let}\ x = e\ \textbf{in}\ e' : \forall \vec{\alpha'}.\tau'$ is proved with the (LET) rule; i.e.,

$$\frac{\begin{array}{l} A \supset e : \forall \vec{\alpha}.\tau \\ A\{x : \forall \vec{\alpha}.\tau\} \supset e' : \forall \vec{\alpha'}.\tau' \end{array}}{A \supset \textbf{let}\ x = e\ \textbf{in}\ e' : \forall \vec{\alpha'}.\tau'}$$

In view of proposition 2 we may assume, w.l.o.g., that $\vec{\alpha'}$ is not free in A and $A\{x : \forall \vec{\alpha}.\tau\}$. By induction assumption we have, for any $\vec{\tau'}$, that $A \supset e : \tau$ and $A\{x : \forall \vec{\alpha}.\tau\} \supset e' : \tau'[\vec{\tau'}/\vec{\alpha'}]$ are derivable in MM'. Consequently,

$$\frac{\begin{array}{l} A \supset e : \tau \\ A\{x : \forall \vec{\alpha}.\tau\} \supset e' : \tau'[\vec{\tau'}/\vec{\alpha'}] \\ \qquad\qquad (\text{since } \vec{\alpha} \text{ not free in } A) \end{array}}{A \supset \textbf{let}\ x = e\ \textbf{in}\ e' : \tau'[\vec{\tau'}/\vec{\alpha'}]\ \text{(LET-P')}}$$

(FIX-P)

Assume that $A \supset \mathbf{fix}x.e : \forall\vec{\alpha}.\tau$ is derivable in MM by the (FIX-P) rule; that is,

$$\frac{A\{x : \forall\vec{\alpha}.\tau\} \supset e : \forall\vec{\alpha}.\tau}{A \supset \mathbf{fix}x.e : \forall\vec{\alpha}.\tau}$$

W.l.o.g. (proposition 2) we may assume that $\vec{\alpha}$ is not free in A and $A\{x : \forall\vec{\alpha}.\tau\}$. By the induction hypothesis we know that $A\{x : \forall\vec{\alpha}.\tau\} \supset e : \tau$ is derivable in MM', and consequently we get

$$\frac{\begin{array}{l} A\{x : \forall\vec{\alpha}.\tau\} \supset e : \tau \\ \quad (\text{since } \vec{\alpha} \text{ is not free in } A) \end{array}}{A \supset \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}]\ (\text{FIX-P'})}$$

$\Leftarrow$: It is sufficient to show $X' \vdash A \supset e : \tau \Rightarrow X \vdash A \supset e : \forall\vec{\alpha}.\tau$. We shall prove that every axiom and rule in MM' is derivable in MM. The proof for X = DM and CH is similar.

Note that it is easy (but not completely trivial) to show that

$$(\text{INST}^k)\quad \frac{A \supset e : \forall\vec{\alpha}.\tau}{A \supset e : \tau[\vec{\tau}/\vec{\alpha}]}$$

and

$$(\text{GEN})^k\quad \frac{\begin{array}{l} A \supset e : \tau \\ (\vec{\alpha} \text{ not free in } A) \end{array}}{A \supset e : \forall\vec{\alpha}.\tau}$$

are derivable rule schemes in MM.

(TAUT') Let $MM' \vdash A\{x : \forall\vec{\alpha}.\tau\} \supset x : \tau[\vec{\tau}/\vec{\alpha}]$. We have the following proof tree in MM:

$$\frac{A\{x : \forall\vec{\alpha}.\tau\} \supset x : \forall\vec{\alpha}.\tau\ (\text{TAUT})}{A\{x : \forall\vec{\alpha}.\tau\} \supset x : \tau[\vec{\tau}/\vec{\alpha}]\ (\text{INST}^k)}$$

(APPL), (ABS) Trivial.

(LET-P') In MM' we have

$$\frac{\begin{array}{l} A \supset e : \tau \\ A\{x : \forall\vec{\alpha}.\tau\} \supset e' : \tau' \\ \quad (\vec{\alpha} \text{ not free in } A) \end{array}}{A \supset \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'}$$

and in MM

$$\frac{\dfrac{\begin{array}{l} A \supset e : \tau \\ \quad (\vec{\alpha} \text{ not free in } A) \end{array}}{A \supset e : \forall\vec{\alpha}.\tau\ (\text{GEN}^k)} \quad A\{x : \forall\vec{\alpha}.\tau\} \supset e' : \tau'}{A \supset \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'\ (\text{LET-P})}$$

(FIX-P') In MM' we have the rule

$$\frac{\begin{array}{l} A\{x : \forall\vec{\alpha}.\tau\} \supset e : \tau \\ \quad (\vec{\alpha} \text{ not free in } A) \end{array}}{A \supset \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}]}$$

and in MM

$$\frac{\dfrac{\dfrac{\begin{array}{l} A\{x : \forall\vec{\alpha}.\tau\} \supset e : \tau \\ \qquad\qquad (\vec{\alpha} \text{ not free in } A) \end{array}}{A\{x : \forall\vec{\alpha}.\tau\} \supset e : \forall\vec{\alpha}.\tau\ (\text{GEN}^k)}}{A \supset \mathbf{fix}x.e : \forall\vec{\alpha}.\tau\ (\text{FIX-P})}}{A \supset \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}]\ (\text{INST}^k)}$$

Proof: (Proof of theorem)

Immediate from Lemma 4.

4.1.3 Consistently Labeled Syntax Trees

In this section we define (type) labeled syntax-trees and a notion of consistency of such labelings. We shall prove that consistently labeled syntax-trees and derivations in the syntax-oriented versions of our type inference systems are in a one-to-one relation.

Definition 7 *(Typed syntax tree, generic/nongeneric type variable occurrences, well-typed syntax tree)*
A typed syntax tree T^τ *is a syntax tree* T *with a function* $\tau : N(T) \to M$*, called a* type labeling.
For a given type environment A*, expression* e *with syntax tree* $T = T(e)$*, and type labeling* τ *for* T*, we say a type variable* α *is* nongeneric *at node* n'' *in* T *if* n'' *is in the scope of a* λ*-binding* n*,* $E(n) = (n_x, n')$*, and* α *occurs (free) in* $\tau(n_x)$*; or if* α *occurs free in* A*. If* α *is not nongeneric at* n''*, it is* generic *at* n''*.* $NGTV(n'')$ *denotes the set of all nongeneric type variables at* n''*, and* $GTV(n'')$ *is* $TV - NGTV(n'')$*.*[2]
For fixed syntax-tree $T = (N, N_F, E, L)$ *of* λ*-expression* e *and type labeling* τ *the labeled syntax tree* T^τ *is* (MM-)consistently labeled *under type assignment* A *if it satisfies the following properties.*

1. *(Local conditions)*
   For all $n \in N_F$,
   (a) $L(n) = \lambda, E(n) = (n', n'') \Rightarrow \tau(n) = \tau(n') \to \tau(n'')$
   (b) $L(n) = @, E(n) = (n', n'') \Rightarrow \tau(n') = \tau(n'') \to \tau(n)$
   (c) $L(n) = \mathbf{let}, E(n) = (n', n'', n''') \Rightarrow \tau(n') = \tau(n'') \wedge \tau(n''') = \tau(n)$
   (d) $L(n) = \mathbf{fix}, E(n) = (n', n'') \Rightarrow \tau(n') = \tau(n'') \wedge (\exists R)\ R|_{GTV(n)}(\tau(n'')) = \tau(n)$
2. *(Scoping conditions)*
   For all $n \in N$,
   (a) *if* n *is a* λ*-binding then*
   $(\forall n' \in BVOM_T(n)\ \tau(n) = \tau(n')$
   (b) *if* n *is a* **let***-binding then*
   $(\forall n' \in BVOM_T(n))(\exists R)\ R|_{GTV(n)}(\tau(n)) = \tau(n')$
   (c) *if* n *is a* **fix***-binding then*
   $(\forall n' \in BVOM_T(n))(\exists R)\ R|_{GTV(n)}(\tau(n)) = \tau(n')$
3. *(Context condition)*
   For all $n \in FVO_T$*, if* $L(n) = x$ *and* $A(x) = \forall \vec{\alpha}.\tau'$ *then* $(\exists R)\ R|_{\vec{\alpha}}(\tau') = \tau(n)$*.*

The labeled tree T^τ *is* DM-consistently labeled *if it is MM-consistently labeled and the conditions*
For all $n \in N$,

- $L(n) = \mathbf{fix}, E(n) = (n', n'') \Rightarrow \tau(n') = \tau(n'') = \tau(n)$
- *if* n *is a* **fix***-binding then*
  $(\forall n' \in BVOM_T(n))\ \tau(n) = \tau(n')$

are satisfied. T^τ *is CH-consistently labeled if it is DM-consistently labeled and additionally the following constraint is satisfied.*
For all $n \in N$,

- *if* n *is a* **let***-binding then*
  $(\forall n' \in BVOM_T(n))\ \tau(n) = \tau(n')$

[2] Of course, $NGTV(n'')$ and $GTV(n'')$ are parameterized over A, $T(e)$, and τ.

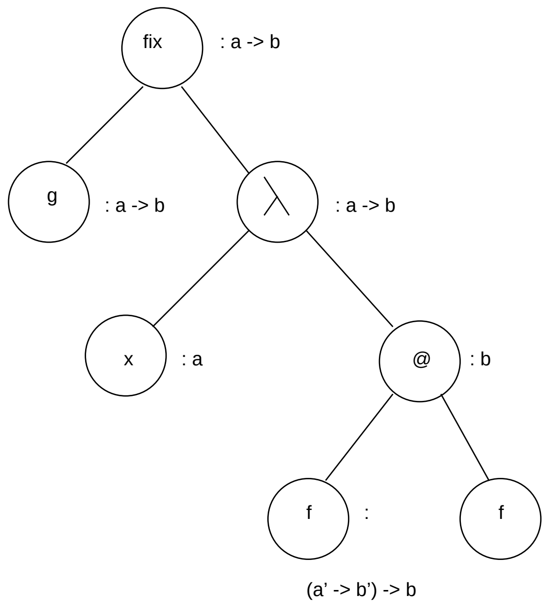

Figure 4.1: A consistently labeled syntax tree

Let X = CH, DM, or MM. An expression e is X-consistently labelable *if there is a type labeling τ for syntax tree $T = T(e)$ such that T^{τ} is X-consistently labeled.*

Consider the λ-expression $g(\mathbf{fix}\, f.\lambda x.ff)$ in the type environment $\{g : \forall\alpha.(\alpha \to \alpha) \to \alpha\}$. A consistently labeled syntax tree, T, for e is presented in Figure 4.1.

The syntax tree in the example has nine nodes, $n1, \ldots, n9$. Its only free variable occurrence is the node $n2$; that is, $\mathrm{FVO}_T = \{n2\}$. The node $n4$ is a **fix**-binding, and $n5$ is a λ-binding. The bound variable occurrence map associates with each one of these bindings the set of all their applied occurrences: $BVOM_T = \{n4 \mapsto \{n8, n9\}, n5 \mapsto \emptyset\}$. Since A contains no free type variable, all type variables are generic at nodes $n1, n2, n3, n4, n5$, and $n6$. Since a occurs in the type labeling of $n6$, a is nongeneric at nodes$n7, n8$, and $n9$; all other type variables are generic at $n7, n8, n9$. Note that T is consistently labeled since all the conditions in definition 7 are satisfied; in particular, the types at the applied occurrences of f $n8$ and $n9$ are substitution instances of the type at the **fix**-binding of f, node $n4$, and a, b are generic type variables at $n4$.

The following theorem shows that derivations in the syntax-oriented type inference systems are characterized by corresponding consistently labeled syntax trees and vice versa.

Theorem 6 *Let X = CH, DM, MM. For A, e, τ', $X' \vdash A \supset e : \tau' \Leftrightarrow (\exists\tau)\, T(e)^{\tau}$ is an X-consistently labeled syntax tree for e with root n, and $\tau(n) = \tau'$.*

Proof:

We shall only give the proof for X = MM. The modifications for the other typing disciplines are trivial.

($\Rightarrow$) Let A_0, e_0 be fixed. Let $T_0 = T(e_0)$ be a syntax tree for e_0 with root n_0, as usual. Let P_0 be an MM'-proof tree for $A_0 \supset e_0 : \tau_0'$. Since MM' is syntax-directed, P_0 and T_0 are isomorphic, and consequently we can define a mapping $\tau_0 : N(T_0) \to M$ by

$\tau_0(n, A \supset e : \tau) =$
case $e =$
x **(variable):**

$$\{n \mapsto \tau\}$$

$\lambda x.e''$**:** for $E(n) = (n', n'')$, and
$\dfrac{A\{x : \tau'\} \supset e'' : \tau''}{A \supset e : \tau}$
in the proof tree P_0,

$$\{n \mapsto \tau, n' \mapsto \tau'\} \cup \tau_0(n'', A\{x : \tau'\} \supset e'' : \tau'')$$

$e'e''$**:** for $E(n) = (n', n'')$, and
$\dfrac{A \supset e' : \tau' \quad A \supset e'' : \tau''}{A \supset (e'e'') : \tau}$
in the proof tree P_0,

$$\{n \mapsto \tau\} \cup \tau_0(n', A \supset e' : \tau') \cup \tau_0(n'', A \supset e'' : \tau'')$$

let $x = e''$ **in** e'''**:** for $E(n) = (n', n'', n''')$, and
$\dfrac{A \supset e'' : \tau'' \quad A\{x : \forall\vec{\alpha}.\tau''\} \supset e''' : \tau}{A \supset \textbf{let}\ x = e''\ \textbf{in}\ e''' : \tau}$
in the proof tree P_0,

$$\begin{aligned}\{n \mapsto \tau, n' \mapsto \tau''\} \quad &\cup \quad \tau_0(n'', A \supset e'' : \tau'') \\ &\cup \quad \tau_0(n''', A\{x : \forall\vec{\alpha}.\tau''\} \supset e''' : \tau)\end{aligned}$$

fix $x.e''$**:** for $E(n) = (n', n'')$, and
$\dfrac{A\{x : \forall\vec{\alpha}.\tau''\} \supset e : \tau''}{A \supset \textbf{fix}\ x.e : \tau}$
in the proof tree P_0,

$$\{n \mapsto \tau, n' \mapsto \tau''\} \cup \tau_0(n'', A\{x : \forall\vec{\alpha}.\tau''\} \supset e : \tau'')$$

and, furthermore, $\tau_0(n_0, A_0 \supset e_0 : \tau_0') = \tau_0'$.
Now, it is easy to check that $T_0^{\tau_0}$ is an MM-consistently labeled syntax tree with root n_0 and $\tau_0(n_0) = \tau_0'$.

($\Leftarrow$) Let A_0, e_0 be fixed. Let $T_0^{\tau_0}$ be an MM-consistently labeled syntax tree for e_0 with root n_0. There is an assignment $\mathcal{A}$ from $N(T_0)$ to type environments that satisfies the following properties.
$\mathcal{A}(n_0) = A_0$ and for all $n \in N(T_0)$,

- if $L(n) = \lambda, E(n) = (n', n'')$, then
  $\mathcal{A}(n') = \mathcal{A}(n'') = \mathcal{A}(n)\{L(n') : \tau_0(n')\}$
- if $L(n) = @, E(n) = (n', n'')$, then
  $\mathcal{A}(n') = \mathcal{A}(n'') = \mathcal{A}(n)$
- if $L(n) = \textbf{let}, E(n) = (n', n'', n''')$, then
  $\mathcal{A}(n') = \mathcal{A}(n''') = \mathcal{A}(n)\{L(n') : \forall\vec{\alpha}(n').\tau_0(n')\}$

- if $L(n) = \textbf{fix}$, $E(n) = (n', n'')$, then
  $\mathcal{A}(n') = \mathcal{A}(n'') = \mathcal{A}(n)\{L(n') : \forall\vec{\alpha}(n').\tau_0(n')\}$

where $\alpha(\vec{n'})$ consists of all the generic variables at node n' that occur in $\tau_0(n')$.
Now it is straightforward, by induction on the syntax of e_0, to show that $MM' \vdash \mathcal{A}(n) \supset [n] : \tau_0(n)$ for all $n \in N(T_0)$.

The proof shows that actually something even stronger is true. We can start with a consistently labeled syntax tree for e, construct a proof tree for e from it via the encoding A, and then generate a consistent labeling for e again via τ_0 from the proof tree. This labeling turns out to be the same one we started out with.

4.1.4 Extraction of Equations and Inequalities

In this section we make the connection between the consistent tree labeling characterization and solving a system of equations and inequalities (SEI) precise.

The tree labeling characterization gives us a different (yet in principle familiar) formulation of type inference problems. If we initially associate a distinct type variable α_n with every node n in T_e then the tree characterization gives us a collection of simultaneous constraints of equational form, such as $\alpha_n = \alpha_{n'} \rightarrow \alpha_{n''}$ and, essentially, of inequational form $\alpha_{n''} \leq \alpha_n$. A connection between consistent labelings and semi-unification seems close at hand. We have to be a little bit careful, though, since the quotient substitutions in the inequational constraints of consistent labelings carry context conditions: Their domains are restricted to generic type variables, the collection of which in turn is a function of the position of the node in the syntax tree where the constraint has to hold. We could always keep track of such context conditions in the form of conditional inequalities $(GTV(n))\alpha_{n''} \leq \alpha_n$ — this is essentially the "Generalized Unification Problem" of [57] — but this is not necessary. As we shall see in this subsection, the context conditions can be encoded efficiently in terms of additional (unconditional) inequalities the specific nature of which captures precisely the fact that the set of generic type variables is generally different from node to node in the same syntax tree. This will indeed lead us to a reduction of consistent labeling to semi-unification.

We shall consider a small, but instructive example due to Kfoury to see that it would be wrong in DM and MM to naively label a syntax tree with distinct type variables and then to collect equations from equality constraints in the consistent labeling definition and inequalities of the form $\alpha_{n''} \leq \alpha_n$ when the consistent labeling constraint reads, say, $R|_{GTV(n)}(\tau(n'')) = \tau(n)$ (see constraint 1d).

Let $e_0 \equiv \lambda y.\textbf{let}\ f = \lambda x.(xy)\ \textbf{in}\ (ff)$. A syntax tree $T_0 = T(e_0)$ with nodes $n1, \ldots, n12$ is given in Figure 4.2. By proceeding in the naive manner outlined above we associate distinct type variables $\alpha_{n1}, \ldots, \alpha_{n12}$ with each node and collect constraints for an MM-consistent (or DM-consistent) labeling. The equations and inequalities thus constructed are displayed in Table 4.3.

This SEI is solvable. For example, the substitution

$$\begin{aligned} S \;=\; & \{\alpha_{n1} \mapsto (\alpha_{n2} \rightarrow \alpha_{n9}), \alpha_{n3} \mapsto ((\alpha_{n2} \rightarrow \alpha_{n7}) \rightarrow \alpha_{n7}), \\ & \alpha_{n4} \mapsto (\alpha_{n2} \rightarrow \alpha_{n7}), \alpha_{n5} \mapsto (\alpha_{n2} \rightarrow \alpha_{n7}), \\ & \alpha_{n6} \mapsto \alpha_{n2}, \alpha_{n8} \mapsto ((\alpha_{n2} \rightarrow \alpha_{n7}) \rightarrow \alpha_{n7}), \\ & \alpha_{n10} \mapsto (((\beta_1 \rightarrow \alpha_{n9}) \rightarrow \alpha_{n9}) \rightarrow \alpha_{n9}), \\ & \alpha_{n11} \mapsto ((\beta_1 \rightarrow \alpha_{n9}) \rightarrow \alpha_{n9}), \alpha_{n12} \mapsto \alpha_{n9}\} \end{aligned}$$

where β_1 is a "new" type variable not occurring anywhere in the original constraints is a semi-unifier, in fact the most general in a sense to be made precise in chapter 5. Unfortunately, however, e_0 is untypable. This can be seen by looking at the quotient substitutions for the solution S. The quotient substitution for the first inequality is $R_1 = \{\alpha_{n2} \rightarrow (\beta_1 \rightarrow \alpha_{n9}), \alpha_{n7} \rightarrow \alpha_{n9}\}$ and for the second inequality it is

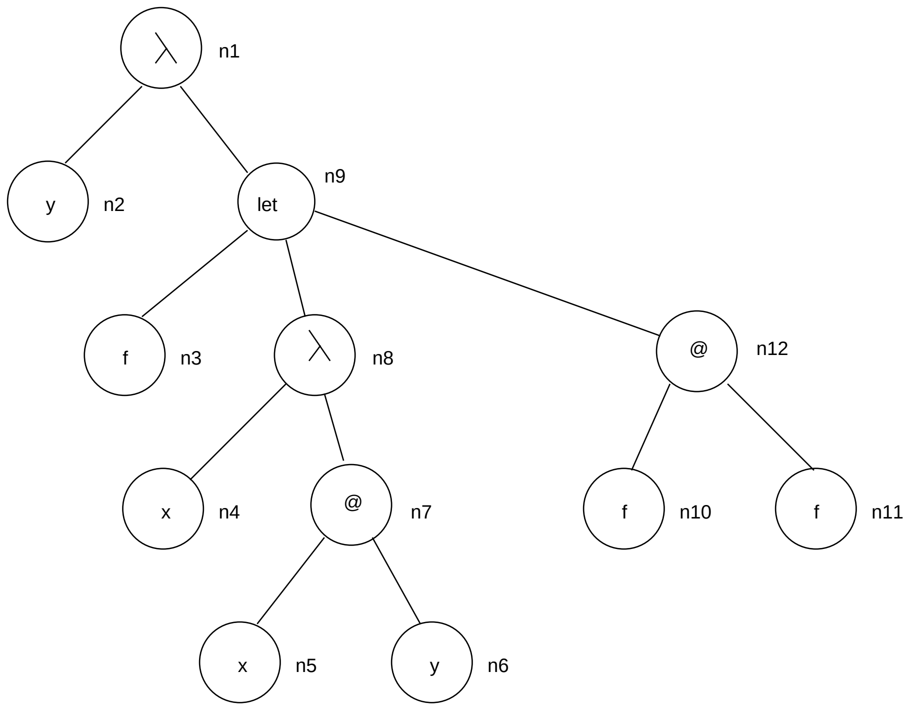

Figure 4.2: An untypable expression

$$\begin{array}{rcl@{\qquad}rcl}
\alpha_{n1} & = & \alpha_{n2} \rightarrow \alpha_{n9} & & & \\
\alpha_{n9} & = & \alpha_{n12} & \alpha_{n2} & = & \alpha_{n6} \\
\alpha_{n3} & = & \alpha_{n8} & \alpha_{n4} & = & \alpha_{n5} \\
\alpha_{n8} & = & \alpha_{n4} \rightarrow \alpha_{n7} & \alpha_{n3} & \leq & \alpha_{n10} \\
\alpha_{n5} & = & \alpha_{n6} \rightarrow \alpha_{n7} & \alpha_{n3} & \leq & \alpha_{n11} \\
\alpha_{n10} & = & \alpha_{n11} \rightarrow \alpha_{n12} & & &
\end{array}$$

Table 4.3: Incorrect SEI for untypable example

$R_2 = \{\alpha_{n2} \rightarrow \beta_1, \alpha_{n7} \rightarrow \alpha_{n9}\}$. Since α_{n2} is the type of the left node of the λ-binding $n1$, the type variable α_{n2} is nongeneric at nodes $n10$ and $n11$ and consequently our quotient substitutions violate the stipulation that their domain may only include generic type variables. Because of the two occurrences of β_1 there is no way of simply "making" α_{n2} equal to both $R_1(\alpha_{n2})$ and $R_2(\alpha_{n2})$. Since every other solution of the above constraints must be a substitution refinement of S (the technical details are in chapter 5) there is also no way of doing this for any other substitution. The expression e_0 is not consistently labelable, and consequently it is untypable (in the Mycroft Calculus and thus, trivially, in the Hindley and Milner Calculi).

If we can somehow encode with equations and inequalities the context constraint that certain type variables (the nongeneric ones) may not be instantiated to other variables or terms by quotient substitutions of candidate solutions then a consistent labeling can still be reduced to pure semi-unification. This is indeed possible and actually quite simple[3] Consider an inequality $\tau_1 \leq \tau_2$ containing variable α. Notice that the quotient substitution R of any *uniform* semi-unifier S of $\{\tau_1 \leq \tau_2, \alpha \leq \alpha\}$ will not instantiate *any* of the type variables occurring in $S(\alpha)$. This device makes it possible for a solution to instantiate the variable α, but it "protects" the resulting term from being instantiated any further by a quotient substitution.

With this insight we can now adjoin the inequality $\alpha_{n2} \leq \alpha_{n2}$ with each of our two inequality constraints in Table 4.3, thus forming small groups of inequalities that have to share the same quotient substitution.[4] The resulting SEI is indeed unsolvable, in correspondence to the fact that e_0 is not consistently labelable. The technical details showing correctness of the transformation that

1. collects equational and inequality constraints in a "naive" manner (in accordance with the consistent labeling requirements), and
2. adjoins inequalities of the form $\alpha \leq \alpha$ for every "naively" collected inequality that arises from a node that is in the scope of a λ-binding,

is presented below.

Definition 8 *Let $T_0 = T(e_0)$ be a syntax tree for e_0 with root n_0, and let $t : N(T_0) \rightarrow TV$ be an injective mapping from the nodes in T_0 to the set of type variables. The* canonical system of equations and inequalities *$SEI_t(e_0) = SEI_t^{MM}(e_0)$ is $(\mathcal{E}, \mathcal{I})$ where*

$$\begin{aligned}
\mathcal{E} \;=\; & \{t(n) = t(n') \rightarrow t(n'') : n, n', n'' \in N(T_0) \mid L(n) = \lambda, E(n) = (n', n'')\} \\
\cup\; & \{t(n') = t(n'') \rightarrow t(n) : n, n', n'' \in N(T_0) \mid L(n) = @, E(n) = (n', n'')\} \\
\cup\; & \{t(n') = t(n''), t(n''') = t(n) : n, n', n'', n''' \in N(T_0) \mid L(n) = \mathbf{let}, \\
& E(n) = (n', n'', n''')\} \\
\cup\; & \{t(n') = t(n'') : n, n', n'' \in N(T_0) \mid L(n) = \mathbf{fix}, E(n) = (n', n'')\} \\
\cup\; & \{t(n') = t(n'') : n' \lambda\text{-}binding, n'' \in BVOM_{T_0}(n')\}
\end{aligned}$$

for every **let**- *or* **fix**-*binding n, $n' \in BVOM_{T_0}(n)$, we define*

$$I_{n,n'} \;=\; \{t(n) \leq t(n')\} \cup \{t'' \leq t'' : t'' \in NGTV(n)\};$$

for $n, n', n'' \in N(T_0)$ such that $L(n) = \mathbf{fix}$ and $E(n) = (n', n'')$,

$$I^{\mathbf{fix}}_{n,n''} \;=\; \{t(n'') \leq t(n)\} \cup \{t'' \leq t'' : t'' \in NGTV(n)\};$$

[3]But it has been overlooked by others approaching the same problem (see [58]).

[4]This is the reason why we opted to introduce SEI's with groups of inequalities instead of simple inequalities.

and finally,
$\mathcal{I} = \{I_{n,n'} : (n,n') \in BVOM_{T(e_0)}\} \cup \{I^{\mathbf{fix}}_{n,n''} : n, n', n'' \in N(T_0) | L(n) = \mathbf{fix}, E(n) = (n', n'')\}$.

We shall usually drop the subscript from $SEI^{MM}_t(e)$ and simply write $SEI^{MM}(e)$ since the specific nature of t is obviously irrelevant. In a similar fashion we can define $SEI^X(e)$ for X = CH, DM.

Theorem 7 *Let X = CH, DM, or MM; let $T_0 = T(e_0)$ be a syntax-tree for e_0 with root n_0; let $t : N(T_0) \rightarrow TV$ be an arbitrary injective map; and let τ be an arbitrary monotype.*

There is an X-consistent type labeling τ_0 for T_0, with $\tau_0(n_0) = \tau$, if and only if there is a solution S of $SEI^X_t(e)$ such that $S(t(n_0)) = \tau$.

Proof:

As always we shall only consider the case X = MM.

($\Rightarrow$) Assume $T_0^{\tau_0}$ is a well-typed syntax tree for e_0 such that $\tau_0(n_0) = \tau$. Let $SEI_t(e_0) = (\mathcal{E}, \mathcal{I})$ as defined above. Define $S = \{t(n) \mapsto \tau_0(n) : n \in N(T_0)\}$. By assumption, $S(t(n_0)) = \tau$. Furthermore it is easy to see, by checking all four major cases, that all equations in $\mathcal{E}$ are satisfied. Now consider $I_{n,n'}$ where n is a **let**- or **fix**-binding and n' is a bound occurrence of n. We have

$$\begin{aligned} S(I_{n,n'}) &= \{S(t(n)) \leq S(t(n'))\} \cup \{S(t'') \leq S(t'') : t'' \in NGTV(n)\} \\ &= \{\tau_0(n) \leq \tau_0(n')\} \cup \{\tau_0(n'') \leq \tau_0(n'') : \\ &\quad FV(\tau_0(n'')) \subset NGTV(n)\} \end{aligned}$$

Since $T_0^{\tau_0}$ is consistently labeled there is a substitution $R_{n,n'}$ such that $R_{n,n'}|_{GTV(n)}(\tau_0(n)) = \tau_0(n')$.
This implies that $R_{n,n'}|_{GTV(n)}(\tau_0(n'')) = \tau_0(n'')$ for all n'' such that $FV(\tau_0(n'')) \subset NGTV(n)$. A similar analysis can be performed for every $I^{\mathbf{fix}}_{n,n''}$. This shows that S is a (proper, nonuniform) semi-unifier of $SEI_t(e_0)$.

($\Leftarrow$) Assume S is a solution of $SEI_t(e_0)$ such that $S(t(n_0)) = \tau$. Let T_0 be a syntax tree for e_0 with root n_0 as always. Define $\tau_0(n) = S(t(n)), n \in N(T_0)$. Clearly, $\tau_0(n_0) = \tau$ by assumption. We shall show that $T(e_0)^{\tau_0}$ is a well-typed syntax tree. Since S is a solution of $SEI_t(e_0) = (\mathcal{E}, \mathcal{I})$ all equalities in $S(\mathcal{E})$ are satisfied and it is easy to see that all equational constraints hold for $T(e_0)^{\tau_0}$ to be well-typed. Observe that, by definition of NGTV, the set of non-generic type variables at a node n in $T_0^{\tau_0}$ is exactly the set of type variables occurring in any $\tau_0(n')$ where n' is a λ-binding whose scope contains n. We also know that for any $I_{n,n'} \in \mathcal{I}$ there is a quotient substitution $R_{n,n'}$ such that

$$\begin{aligned} R_{n,n'}(S(t(n))) &= S(t(n')) \\ R_{n,n'}(S(t'')) &= S(t''), \qquad t'' \in NGTV(n). \end{aligned}$$

This implies

$$\begin{aligned} R_{n,n'}(\tau_0(n)) &= \tau_0(n') \\ R_{n,n'}(\tau_0(n'')) &= \tau_0(n''), \quad n'' \text{is a } \lambda\text{-binding whose scope contains } n. \end{aligned}$$

By the observation above we can conclude that $R_{n,n'}$ is the identity on $NGTV(n)$, which shows that $R_{n,n'} = R_{n,n'}|_{GTV(n)}$ and thus $R_{n,n'}|_{GTV(n)}(\tau_0(n)) = \tau_0(n')$. A similar argument holds for $I^{\mathbf{fix}}_{n,n''} \in \mathcal{I}$.

It is obvious that analogous transformations, only with "more" equational constraints and fewer inequality constraints, can be performed that give reductions from DM-consistent labeling, respectively CH-consistent labeling, to semi-unification. Actually, in the Hindley Calculus there is no problem with context conditions on inequalities in labeled syntax trees since there are no inequational constraints in the first place: all constraints are equational. Consequently the resulting SEI contains only equations, and classical unification produces the most general unifier rapidly for an appropriate representation of type expressions (namely term graphs) and substitutions ("downward closed" equivalences on term graphs). This establishes the connection of type inference with unification (e. g., see [90]). More specifically, it is easy to see that for an expression e of size n we can generate in linear or almost-linear time on a RAM (depending on the encoding of variables) a set E of monotype equations of size $O(n)$ such that e is typable if and only if E is unifiable. E can be checked for unifiability in linear [89,72] or almost-linear time [43]. This leads to a linear or almost-linear upper bound for the time complexity of deciding typability in the Hindley Calculus. Since the additional inequational constraints in the Milner Calculus seem rather innocuous at first sight, this may have led researchers to incorrectly claim linear or quadratic bounds on type inference for the whole Milner Calculus [65,81].

Theorem 8 *Let X = CH, DM, MM, or FMM. Typability in X is polynomial-time reducible to semi-unifiability.*

Proof:

Note that constructing $SEI(e)$ for X can easily be done in polynomial-time. By the three previous theorems $SEI(e)$ is solvable if and only if e is typable in X.

Corollary 5 *Semi-unifiability is PSPACE-hard (for polynomial-time reductions).*

Proof:

Kanellakis and Mitchell show that the Milner-Calculus is PSPACE-hard [53]. The result follows by theorem 8.

4.2 Reduction of Semi-Unification to the Flat Mycroft Calculus

Semi-unification is a problem without nesting, scoping, context conditions on inequalities, or quantification of (type) variables. These play an eminent role in the definition of typability in the Mycroft Calculus. Nonetheless, as we have seen, the Mycroft Calculus can be efficiently reduced to semi-unification. The Flat Mycroft Calculus is a typing discipline without nesting of polymorphically typed language constructs. Since semi-unification is a basically "flat" problem — scoping and nesting do not enter into its definition — it should not come as a surprise that semi-unification cannot only be reduced to the Mycroft Calculus, but in fact to the *Flat* Mycroft Calculus. Reductions from unification-like problems to typing problems have grown in importance since they allow us to prove lower bounds on the combinatorially simpler unification-like problems and then extend them to their "corresponding" typing problems. Kanellakis and Mitchell proved a combinatorial problem they called polymorphic unification to be PSPACE-hard and extended this lower bound via a polynomial-time reduction to the Milner Calculus [53]. We provide log-space reductions from unifiability to typability in the Hindley Calculus, and from semi-unifiability to typability in the Flat Mycroft Calculus. This shows that

1. typability in the Hindley Calculus is P-complete,
2. the Milner Calculus can be reduced to the Flat Mycroft Calculus thus extending the PSPACE-hardness result for the Milner Calculus to the Flat Mycroft Calculus and thereby answering a question raised by Kanellakis, and
3. the Mycroft Calculus, the Flat Mycroft Calculus, and semi-unification are polynomial-time equivalent.

4.2.1 Simplification of Systems of Equations and Inequalities

So far we have used the term "semi-unification" as if it was a single problem while actually it is parameterized by the ranked alphabet over which terms range. In this subsection we show that our usage is justified in that every SEI over any alphabet can be reduced to an equivalent (see below) SEI over the alphabet $\mathcal{A}_2 = (\{f\}, \{f \mapsto 2\})$ that contains only a single binary functor. In chapter 5 we shall see that this is the "minimal" possible alphabet, since no nonlinear alphabet can encode enough information to admit the same kind of reduction. To make these reductions effective and efficient we shall assume that infinite ranked alphabets have functors encoded by the binary numerals.

Definition 9 *(Equivalent systems of equations and inequalities)*
Let S and S' be SEI's, possibly over different alphabets. S and S' are equivalent *if*

1. *S is semi-unifiable if and only if S' is semi-unifiable, and*
2. *S is uniformly semi-unifiable if and only if S' is uniformly semi-unifiable, and*
3. *S is unifiable if and only if S' is unifiable.*

Replacement of Functors by Constants

Let $\mathcal{A}_{[]}$ be an alphabet that has exactly one functor for any given arity $k \geq 0$. We shall always write $[M_1, \ldots, M_k]$ for the term built up from $M_1, \ldots, M_k$ the unique k-ary functor in $\mathcal{A}_{[]}$. We shall address the collection of all these constructors as *the* list functor since we may view $[\ldots]$ as a single functor that has no arity requirements. W.l.o.g., we may always assume that $\mathcal{A}_{[]}$ is disjoint[5] from any other alphabet we may consider. Let $\mathcal{A}$ be any ranked alphabet, and let $\mathcal{A}_c$ be the alphabet that consists of $\mathcal{A}_{[]}$ and all the functors from $\mathcal{A}$, but such that every functor from $\mathcal{A}$ has its arity changed to 0. Define the transformation function $\mu_c : T(\mathcal{A}, V) \to T(\mathcal{A}_c, V)$,

$$\begin{array}{lcl} \mu_c(x) & = & x, \text{ if } x \in V \\ \mu_c(f(M_1, \ldots, M_n)) & = & [f, [\mu_c(M_1), \ldots, \mu_c(M_n)]], \text{ otherwise} \end{array}$$

The transformation μ_c is obviously well-defined and can be extended to SEI's. We have the following lemma.

Lemma 6 *For all $S \in \Gamma(\mathcal{A}, V)$, S and $\mu_c(S)$ are equivalent.*

The translation of $f_5(f_2(x_1), x_2)$ via μ_c returns $[f_5, [[f_2, [x_1]], x_2]]$. It is easy to see that μ_c can be implemented by a one-way finite state transducer (1FSM-reduction).

Elimination of Constants

Since we have assumed that the constants in A_c are encoded over the binary (unranked) alphabet $\{0, 1\}$, we can represent any constant by a list over 0 and 1.

Let $\mathcal{A}_c$ be as above, and let $\mathcal{A}_{01} = (\{0, 1\}, \{0 \mapsto 0, 1 \mapsto 0\}) \cup A_{[]}$, and define

$$\begin{array}{lcll} \mu_{01}(x) & = & x, & x \in V \\ \mu_{01}(f) & = & [b_1, \ldots, b_k], & f \text{ is encoded by } b_1 \ldots b_k \in \{0, 1\}^* \\ \mu_{01}([M_1, \ldots, M_n]) & = & [\mu_2(M_1), \ldots, \mu_2(M_n)] & \end{array}$$

Again, μ_{01} can be canonically extended to SEI's over $\mathcal{A}_c$. The correctness of this transformation is guaranteed by the next lemma.

Lemma 7 *For all SEI's $S \in \Gamma(\mathcal{A}_c, V)$, S and $\mu_{01}(S)$ are equivalent.*

The encoding of $[f_5, [[f_2, [x_1]], x_2]]$ via μ_{01} is $[[1, 0, 1], [[[1, 0], [x_1]], x_2]]$. Again, this translation can be implemented by a one-way finite state machine.

Elimination of List Constructor

So far all reductions were 1FSM-reductions. In [37] we presented a 1FSM-reduction of $\mathcal{A}_{01}$ into the set of pure (**let**- and **fix**-free) λ-expressions that translated unifiability of SEI's into CH-typability. Of course, this reduction is very sensitive to the particular representation of terms, SEI's and λ-expressions, but it is interesting to note that this translation is a purely "lexical" process with respect to the standard "string" representation of terms and λ-expressions that we have assumed throughout. Roughly, the only place where "parsing" is necessary in the following steps is in eliminating the list constructor. Even this parsing is "harmless" in that it can be performed by a log-space bounded transformation.[6] The transformation below is inspired — in fact a direct transliteration — of the encoding into the λ-calculus that we used in [37].

Let $\mathcal{A}_{01}$ be as above. Recall that $\mathcal{A}_2$ is the alphabet with only one functor, which is binary. Let x_0 be a variable in V, and let $i : V \to V$ be an injective map whose range does *not* contain x_0. Define μ_2 as follows.

[5] One can think of ranked alphabets as sets whose elements carry an attribute. In this sense we will often treat ranked alphabets simply as sets.

[6] Context-free languages are in general not known to be contained in DLOG [2].

$$\begin{array}{lcl}
\mu_2(x) & = & i(x), \text{ if } x \in V \\
\mu_2(0) & = & f(x_0, f(x_0, x_0)) \\
\mu_2(1) & = & f(x_0, f(x_0, f(x_0, x_0))) \\
\mu_2([]) & = & f(x_0, x_0) \\
\mu_2([M_1, \ldots, M_n]) & = & f(f(\mu_2(M_1), N), x_0) \\
 & & \text{if } \mu_2([M_2, \ldots, M_n]) = f(N, x_0)
\end{array}$$

The different encodings of $0, 1$ and $[]$ indicate why this reduction works: functor "clashes" (failure due to different functors) in unification or semi-unification instances are encoded by so-called "occurs" checks (see chapter 6). No two of the encodings of $0, 1, []$ can unify or semi-unify. The x_0 in the first argument position of all encodings requires that any quotient substitution map x_0 to x_0, and the second argument position would force an instantiation of x_0 were it to succeed, which is impossible (this is akin to the "adjoining" trick in the reduction of typability to semi-unification).[7] Lists of one length can never be unified or semi-unified with lists of another length (or 0 and 1) for essentially the same reason, only this time the x_0 that forces quotient substitutions to map x_0 to x_0 is in the second argument position (for no particular reason but to maintain the analogy to the above-mentioned translation into the λ-calculus). Since x_0 occurs also — deeply nested — in the first argument of the encodings of lists, semi-unification of lists of *different* length could only succeed if a quotient substitution maps a nonvariable term containing x_0 to x_0, which is manifestly impossible, or if it maps x_0 to a nonvariable term containing x_0, which is also impossible since x_0 is "fixed" in the second argument. These considerations lead to the following lemma.

Lemma 8 *For all SEI's $S \in \Gamma(\mathcal{A}_{01}, V)$, S and $\mu_2(S)$ are equivalent.*

For lemmas 6, 7, and 8 we may assume, by proposition 21 in chapter 3, that any given SEI S is in "normal form"; that is, it has only one equation and one inequality per inequality group. The proofs then proceed by induction on the number of inequalities and within each inequality by structural induction on terms.

Theorem 9 *Semi-unifiability, uniform semi-unifiability, and unifiability over any ranked alphabet $\mathcal{A}$ are log-space reducible to semi-unifiability, uniform semi-unifiability, and unifiability, respectively, over alphabet $\mathcal{A}_2$.*

Proof:

By lemmas 6, 7, and 8, and the definition of equivalence of SEI's.

Henceforth we shall assume that, w.l.o.g., our ranked alphabet over which terms are formed is $\mathcal{A}_2$, the minimal nonlinear alphabet that contains only one functor, f, which is binary.

4.2.2 Reduction of Term Equations to the Hindley Calculus

Terms over $\mathcal{A}_2$ (or even $\mathcal{A}_{[]}$) can be encoded in the familiar way in which lists are usually represented in the pure λ-calculus. We will show that this encoding is in fact a log-space reduction of unifiability to the typability problem in the Hindley Calculus (with only pure λ-expressions), which we will also call the *simple typability* problem.

[7] Of course, this argument remains valid if we substitute any term whatsoever for x_0 (but the same one for every occurrence of x_0).

λ-representation of Terms

We shall assume, w.l.o.g., that the set V of variables we have used in terms is identical to the set, also denoted by V, of variables that can occur in λ-expressions. Let also x_0 be a distinguished element of V, and $i : V \to V$ an injective mapping whose range does not contain x_0. Define the mapping $\mu_\lambda : T(\mathcal{A}_2, V) \to \Lambda$ as follows.

$$\begin{array}{lcl} \mu_\lambda(x) & = & i(x), \text{ if } x \in V \\ \mu_\lambda(f(M_1, M_2)) & = & \lambda x_0.x_0\mu_\lambda(M_1)\mu_\lambda(M_2) \end{array}$$

We shall abbreviate $\lambda x_0.x_0e_1e_2$ to $[e_1, e_2]$. Generally, the expression $\lambda x_0.x_0e_1 \ldots e_k$ will be written $[e_1, \ldots, e_k]$. Instead of $\mu_\lambda(M)$ we may also write $\bar{M}$. We let x_0 be a "reserved" variable that cannot occur in any term, whence we may assume that i is the identity function.

The map μ_λ not only gives us an encoding of terms as λ-expressions, but also in the form of the *types* of these λ-expressions in CH, DM, MM, and FMM: there is no difference as to which typing system we choose since the encodings are only pure λ-expressions, and for pure λ-expressions the typing rules in our type calculi are identical. To encode a term equation $M = N$ as a pure λ-expression all we have to do now is to "force" the types of the M and N to be equal. This is easily achieved by applying λ-bound variable, g, to both M and N. Since we can assume that a nontrivial unifiability problem instance consists of only one equation (see proposition 21 in chapter 3) we can thus extend μ_λ to a map $\mu_\lambda : \Gamma(\mathcal{A}_2, V) \to \Lambda$ by

$$\mu_\lambda(M = N) \quad = \quad \lambda g.[g\bar{M}, g\bar{N}] \text{ where } g \notin FV(\bar{M}) \cup FV(\bar{N}) \cup \{x_0\}.$$

For convenience' sake (and by abuse of notation) we will simply write $\bar{M} = \bar{N}$ for $\lambda g.[g\bar{M}, g\bar{N}]$ (which is already an abbreviation).

Correctness

It is easy to see that μ_λ can be computed in logarithmic space. To complete the reduction from unifiability to simple typability, it remains to be shown that μ_λ is indeed a problem reduction; more precisely, we will show that for all $M, N \in T(\mathcal{A}_2, V)$, it holds that the SEI $(M = N)$ is solvable if and only if there is a typing for the λ-expression $\lambda\vec{x}.\bar{M} = \bar{N}$ derivable in the Hindley Calculus where $\vec{x} = FV(\bar{M}) \cup FV(\bar{N})$ Again, as in the first half of this chapter, any sequence $\vec{x}$ may also be viewed as a set.

There are many possible proofs of correctness. For example, we can try to show that the principal types of $\bar{M}$ and $\bar{N}$ are unifiable if and only if M, N are unifiable. This is quite apparently true, but it is technically rather messy to prove since there are in general many more type variables in the principal types of $\bar{M}$ and $\bar{N}$ than there are variables in M and N. For this reason we take an approach in which we get rid of these extra type variables by "normalizing" principal types.

As a proviso to the following discussion let us note that $\lambda\vec{x}.(\bar{M} = \bar{N})$ is a closed λ-expression, and it is simply typable if and only if $\{\vec{x} : \vec{\tau}\} \supset \bar{M} = \bar{N} : \tau'$ is derivable where $\vec{\tau}$ is a sequence of monotypes and τ' is also a monotype.[8] For this reason we shall only work with monotype environments A here; that is, $A(x) \in M$ for all $x \in \mathbf{dom}\, A$.

Unifiability Implies Typability First we show that if a pair of terms is unifiable then the λ-representation of this unifiability instance is simply typable.

Define the *canonical type mapping* τ that maps type environments and terms to monotypes as follows.

$$\begin{array}{lcl} \tau(A, x) & = & A(x), x \in V \\ \tau(A, [M_1, \ldots, M_k]) & = & (\tau(A, M_1) \to \ldots \to \tau(A, M_k) \to \alpha_0) \to \alpha_0 \end{array}$$

[8]The notation $\{\vec{x} : \vec{\tau}\}$ is an obvious short-hand.

Here α_0 denotes a fixed type variable. Once more, we abbreviate $(\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \alpha_0) \rightarrow \alpha_0$ to $[\tau_1, \ldots, \tau_k]$. The following proposition is easy to prove by structural induction over terms.

Proposition 9 *Let A, A' be type environments, and $M, N_1, \ldots, N_k$ terms whose variables are contained in the domain of A.*

1. *$\tau(A, M)$ is well-defined and unique.*
2. *If A is injective then τ is injective with respect to its second argument; i.e., $\tau(A, N_1) = \tau(A, N_2)$ implies $N_1 = N_2$.*
3. *The typing $A \supset \bar{M} : \tau(A, M)$ is derivable.*
4. *If $\{x_1, \ldots, x_n\}$ is the domain of A then $A = \{x_1 : \tau(A, x_1), \ldots, x_n : \tau(A, x_n)\}$*

Given a substitution $\sigma : V \rightarrow T(\mathcal{A}_2, V)$ on terms (not type expressions) we define $\sigma(A)$, the application of σ to a type environment $A = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, as follows.

$$\sigma(A) = \{x_1 : \tau(A, \sigma(x_1)), \ldots, x_n : \tau(A, \sigma(x_n))\}.$$

Note that according to proposition 9, part 4, $\iota(A) = A$ for all A where ι denotes the identity substitution.

Lemma 10 *For all terms M, type environments A, and term substitutions σ, whenever* **dom** *$A \supset FV(\bar{M}) \cup FV(\bar{N})$, then $\tau(A, \sigma(M)) = \tau(\sigma(A), M)$.*

Proof:

We prove this lemma by structural induction on M.[9]

- (Base case) If M is a variable, x_i, then

$$\begin{aligned} \tau(\sigma(A), M) &= \tau(\sigma(A), x_i) \\ &= \sigma(A)(x_i) \\ &= \tau(A, \sigma(x_i)) \text{ (by definition of } \sigma(A)) \\ &= \tau(A, \sigma(M)) \end{aligned}$$

- (Inductive case) If $M = [N_1, \ldots, N_k]$ for some terms $N_1, \ldots, N_k$, then

$$\begin{aligned} \tau(\sigma(A), M) &= \tau(\sigma(A), [N_1, \ldots, N_k]) \\ &= [\tau(\sigma(A), N_1), \ldots, \tau(\sigma(A), N_k)] \\ &= [\tau(A, \sigma(N_1)), \ldots, \tau(A, \sigma(N_k))] \text{(ind. hyp.)} \\ &= \tau(A, [\sigma(N_1), \ldots, \sigma(N_k)]) \\ &= \tau(A, \sigma([N_1, \ldots, N_k])) \\ &= \tau(A, \sigma(M)) \end{aligned}$$

This completes the proof.

Lemma 11 *For all $M, N \in T(\mathcal{A}_2, V)$, if M and N are unifiable then $\lambda\vec{x}.(\bar{M} = \bar{N})$ is simply typable (typable in the Hindley Calculus).*

[9]It is actually more like a "proof by notation".

Proof: By assumption of the lemma there is a unifier υ of M, N; i.e., $\upsilon(M) = \upsilon(N)$. Let A be a type environment whose domain contains sufficiently many variables (that is, at least all variables in $FV(\bar{M}) \cup FV(\bar{N})$). By proposition 9, part 3, both $\upsilon(A) \supset \rho(M) : \tau(\upsilon(A), M)$ and $\upsilon(A) \supset \rho(N) : \tau(\upsilon(A), N)$ are derivable typings. According to lemma 10 and by the fact that υ is a unifier we have $\tau(\upsilon(A), M) = \tau(A, \upsilon(M)) = \tau(A, \upsilon(N)) = \tau(\upsilon(A), N)$. Call this type τ'. Consequently, for any $\alpha' \in TV$,

$$A'\{g : \tau' \to \alpha'\} \supset [g\bar{M}, g\bar{N}] : [\alpha', \alpha']$$

and

$$A' \supset (\bar{M} = \bar{N}) : \tau' \to [\alpha', \alpha']$$

are derivable typings, the latter of which shows that $\lambda\vec{x}.(\bar{M} = \bar{N})$ is simply typable.

Typability Implies Unifiability We now proceed to prove that if $\lambda\vec{x}.(\bar{M} = \bar{N})$, for given terms M and N, is typable then M and N are unifiable.

Some preliminary results on the normalization of typings are helpful in facilitating a translation of types to terms and from typings to substitutions. The *normalization* function ν on types is defined as follows.

$$\begin{array}{l} \nu(\tau) = \alpha, \text{ if } \tau = \alpha \text{ and } \alpha \in TV \\ \nu(\tau) = [\nu(\tau_1), \ldots, \nu(\tau_n)], \text{if } \tau = (\tau_1 \to \ldots \to \tau_n \to \tau') \to \tau' \\ \quad \text{for some } \tau_1, \ldots, \tau_n, \tau' \\ \nu(\tau) = \alpha_0, \text{otherwise} \end{array}$$

Proposition 12 *1. ν is well-defined and unique.*

2. For any set of type expressions $\tau_1, \ldots, \tau_k$ there is an injective type environment A and terms $N_1, \ldots, N_k$ such that $\nu(\tau_i) = \tau(A, N_i)$ for all i such that $1 \leq i \leq k$.

The mapping ν can be extended to type environments in the standard way: $\nu(A) = \{x_1 : \nu(\tau_1), \ldots, x_n : \nu(\tau_n)\}$ if $A = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

Lemma 13 *For any derivable typing $A \supset \bar{M} : \tau$, the typing $\nu(A) \supset \bar{M} : \nu(\tau)$ is also derivable, and $\nu(\tau) = \tau(\nu(A), M)$.*

Proof: This can be shown by simple induction on the structure of M.

Lemma 14 *For all $M, N \in T(\mathcal{A}_2, V)$, if $\lambda\vec{x}.(\bar{M} = \bar{N})$ is simply typable then M and N are unifiable.*

Proof: By assumption, there is a derivable typing $A \supset (\bar{M} = \bar{N}) : \tau$. By the definition of μ_λ this expands to

$$A \supset \lambda g.\lambda x_0.(x_0(g\bar{M})(g\bar{N})) : \tau.$$

Since the typing rules of the Hindley Calculus are syntax-directed, we can conclude, by "backwards reasoning", that there are type expressions τ', τ_2, τ_3 such that $\tau = (\tau' \to \tau_2) \to (\tau_2 \to \tau_2 \to \tau_3) \to \tau_3$ and, with $A' = A\{g : \tau' \to \tau_2, f : \tau_2 \to \tau_2 \to \tau_3\}$, both

$$A' \supset \bar{M} : \tau'$$

and

$$A' \supset \bar{N} : \tau'$$

are derivable. Let us define $A'' = \nu(A')$ and $\tau'' = \nu(\tau')$. By lemma 13, the typings

$$A'' \supset \bar{M} : \tau''$$

and

$$A'' \supset \bar{N} : \tau''$$

are both derivable. If $A'' = \{x_1 : \tau''_1, \ldots, x_k : \tau''_k\}$, proposition 12, part 2, implies that there are terms $M, N_1, \ldots, N_k$ and an injective type environment A_0 such that $\tau(A_0, M) = \tau'', \tau(A_0, N_1) = \tau''_1, \ldots, \tau(A_0, N_k) = \tau''_k$. If we define $\sigma = \{x_1 \to N_1, \ldots, x_k \to N_k\}$, the previous two typings can be rephrased as

$$\sigma(A_0) \supset M : \tau(\sigma(A_0), M)$$

and

$$\{x_1 : \tau(A_0, \sigma(x_1)), \ldots, x_k : \tau(A_0, \sigma(x_k))\} \supset \bar{M} : \tau(A'', M)$$

Also by lemma 13 we can conclude $\tau(\sigma(A_0), M) = \tau(A'', M) = \tau(\sigma(A_0), N)$. Finally, this yields $\tau(A_0, \sigma(M)) = \tau(A_0, \sigma(N))$ by lemma 10 and, since A_0 is injective, by proposition 9, part 2, $\sigma(M) = \sigma(N)$. Consequently, M and N are unifiable.

Theorem 10 *For all $M, N \in T(\mathcal{A}_2, V)$, M and N are unifiable if and only if $\lambda\vec{x}.(\bar{M} = \bar{N})$ is simply typable.*

Proof: Lemma 11 shows one direction, lemma 14 the other.

Corollary 15 *Simple typability (typability in the Hindley Calculus) is P-complete under log-space reductions.*

Proof:

Since simple typability is log-space reducible to unification, it is in P. By theorem 10 the result follows from the fact that unification is P-complete [25].

4.2.3 Reduction of Uniform Semi-Unification

To gain an intuition into the more complicated reduction of general (nonuniform) semi-unification problem instances to the Flat Mycroft Calculus we shall consider a special case here that yields an interesting characterization of uniform semi-unification in terms of a restricted version of the Flat Mycroft Calculus.

Let 1FMM ("Flat Milner-Mycroft Calculus with at most one occurrence of the **fix**-bound variable") denote MM restricted to expressions of the form $\mathbf{fix}\, y.e$ where e is a pure λ-expression containing at most one free occurrence of y. By extending the λ-encodings of terms we can also encode inequalities between terms. For this we need the polymorphic typing rule (FIX-P), though, and consequently we shall assume the (Flat) Mycroft Calculus when we talk about typability in the rest of this chapter.

The consistent labeling formulation for MM already gives an indication of how term inequalities can be captured in the constraints associated with a **fix**-binding. Note that the type for y in $\mathbf{fix} y.e$ in some sense "comes from" the type of e since they have to be equal. Now if we can "force" e to have the type of the λ-encoding $\bar{M}$ of M and if we can "hide" (in the sense that it does not affect the type of e) somewhere in e the λ-encoding $y = \bar{N}$, then the y in $y = \bar{N}$ is bound to have the same type as $\bar{N}$, but by the typing rules for **fix** the occurrence of y must also be a substitution instance of the type of e. In other words, we will have encoded the single term inequality $M \leq N$ as an instance of the 1FMM typability problem. Since M and N, and consequently $\bar{M}$ and $\bar{N}$ contain in general a lot of free variables we have to be a little bit more careful than this. To make sure that different occurrences of a free variable x, say, in M have the same type everywhere (which corresponds to a semi-unifier uniformly applying the same substitution to all occurrences of a variable), the variables in $\bar{M}$ and $\bar{N}$ have to be λ-bound some place, as was the case for encodings of equations (for the same reason, by the way). The λ-bindings for these variables cannot go outside of the whole expression, as in $\lambda\vec{x}.\mathbf{fix}\, y.e$, since — now we are in MM — this would mean that the **fix**-binding is in the scope of the λ-bindings, and essentially *no* type variable in the type of e could be instantiated. Consequently the place where the λ-bindings have to go is just after the **fix**-binding: $\mathbf{fix}\, y.\lambda\vec{x}.e$. This in turn complicates the encoding of the equation $y = \bar{N}$ above, but fortunately everything works out.

Theorem 11 *Uniform semi-unifiability and 1FMM-typability are log-space equivalent.*

Proof:

An inspection of the reduction of MM-typability to semi-unification shows that the instances of 1FMM are reduced to instances of uniform semi-unifiability. Conversely, consider a single inequality $M \leq N$ and the λ-expression

$$\mathbf{fix}\, y.\lambda\vec{x}.K\bar{M}(\lambda\vec{z}.(y\vec{z}) = \bar{N}),$$

which is clearly an instance of 1FMM-typability. Here $\vec{x}$ again is the sequence of all free variables in $\bar{M}$ and $\bar{N}$ in any order, and $\vec{z}$ is a sequence of variables with the same length as $\vec{x}$, but completely disjoint from it. K denotes the term $\lambda x.\lambda y.x$. Since $\bar{M}$ and $\bar{N}$ can be computed in logarithmic space, this expression can clearly be computed in logarithmic space from $M \leq N$. The correctness of this reduction automatically falls out the general case of reducing nonuniform semi-unification to FMM-typability, which is shown towards the end of this chapter.

Corollary 16 *1FMM-typability is P-complete under log-space reductions.*

Proof:

Kapur *et al.* [54] give a complicated algorithm for deciding uniform semi-unifiability in polynomial time (see chapter 6 for more information on their algorithm). Since unification is a subcase of uniform semi-unification this implies the theorem.

This corollary shows that, theoretically, uniform semi-unification is no harder than unification, although, in practice there is a big difference: The polynomial-time algorithm in [54] is very complicated and executes in a polynomial of some higher degree whereas unification has a theoretically and practically very fast algorithm, namely the equivalence class merging algorithm with delayed occurs checking and the union/find data structure (see, e.g., [1, section 6.7]) which seems to be continuously rediscovered (see, e.g., [107]). A theoretically faster, but less practical algorithm is the linear time decision algorithm of [89] and [72].

4.2.4 Reduction of Semi-Unification

We have seen how a single inequality can be encoded in the Flat Mycroft Calculus, even under the restriction that a **fix**-bound variabe may only occur once. Intuitively, it is clear how to proceed from here to encode a whole system of equations and inequalities:

1. Encode every inequality individually as a recursive definition and view the collection of all such recursive definitions as a single mutually recursive definition,
2. encode the mutually recursive definition as a single recursive definition in a "standard" way,
3. and along the way be careful about λ-binding the free variables in the given SEI and do not forget to add encodings for the equations.

The following technical proposition is used later in the proof of correctness of the reduction outlined above. We make use of another abbreviation: For fixed $k > 0$, $\bar{i} = \lambda z_1 \ldots z_k.z_i$ is the i-th projection function for $1 \leq i \leq k$.

Proposition 17 *Let X = CH, DM, MM, FMM. Let $\vec{\tau} = [\tau_1, \ldots, \tau_k]$.*

$$X \vdash A \supset e : \vec{\tau} \Leftrightarrow$$
$$X \vdash A \supset e\bar{i} : \tau_i, i \in \{1, \ldots, k\}$$

$$X \vdash A \supset [e_1, \ldots, e_k] : \vec{\tau} \Leftrightarrow$$
$$X \vdash A \supset e_i : \tau_i, i \in \{1, \ldots, k\}$$

$$(\exists \tau')\, A \supset e_1 = e_2 : \tau' \Leftrightarrow$$
$$(\exists \tau'')\, A \supset e_1 : \tau'' \textit{ and } A \supset e_2 : \tau''$$

Theorem 12 *Semi-unification is log-space reducible to typability in the Flat Mycroft Calculus.*

Proof:

Without loss of generality we may assume that $\mathcal{A} = \mathcal{A}_2$. As noted in chapter 2 it is sufficient to show that any SEI $S = (M_0 = N_0, M_1 \leq N_1, \ldots, M_k \leq N_k)$ is reducible to FMM. Let $\vec{x} = x_1 \ldots x_m$ where $x_1, \ldots, x_m$ are all the distinct variables occurring in S; let $\vec{z} = z_1 \ldots z_m$ be m distinct variables not occurring in S.

Now consider

$$\begin{array}{rcl}\lambda(S) & = & \mathbf{fix} y.\lambda\vec{x}.K[\bar{M}_1, \ldots, \bar{M}_k], \\ & & [\bar{M}_0 = \bar{N}_0, \lambda\vec{z}.(y\vec{z}\bar{1} = \bar{N}_1), \ldots, \lambda\vec{z}.(y\vec{z}\bar{k} = \bar{N}_k)]\end{array}$$

where $K = \lambda x.\lambda y.x$, as usual. $\lambda(S)$ can clearly be constructed in logarithmic space. We will show that S has a solution if and only if $\lambda(S)$ is typable in the Flat Mycroft Calculus.

Lemma 18 *There is a type τ such that*

$$\begin{aligned} FMM \vdash \{\} \quad \supset \quad & \mathbf{fix}\, y.\lambda\vec{x}.K[\bar{M}_1,\ldots,\bar{M}_k], \\ & [\bar{M}_0 = \bar{N}_0, \lambda\vec{z}.(y\vec{z}\bar{1} = \bar{N}_1),\ldots,\lambda\vec{z}.(y\vec{z}\bar{k} = \bar{N}_k)] : \tau \end{aligned}$$

if and only if there are monotypes $\vec{\tau} = \tau_1 \ldots \tau_k, \tau_{M_0}, \tau_{M_1},\ldots,\tau_{M_k},\tau_{N_0},\tau_{N_1},\ldots,\tau_{N_k}$ such that

$$\begin{aligned} &\{\vec{x} : \vec{\tau}\} \supset \bar{M}_i : \tau_{M_i} \\ &\{\vec{x} : \vec{\tau}\} \supset \bar{N}_i : \tau_{N_i} \\ &\tau_{M_0} = \tau_{N_0} \\ &\tau_{M_i} \leq \tau_{N_i}, i \in \{1,\ldots,k\} \end{aligned}$$

Proof:

$$\begin{aligned} \{\} \quad \supset \quad & \mathbf{fix}\, y.\lambda\vec{x}.K[\bar{M}_1,\ldots,\bar{M}_k], \\ & [\bar{M}_0 = \bar{N}_0, \lambda\vec{z}.(y\vec{z}\bar{1} = \bar{N}_1),\ldots,\lambda\vec{z}.(y\vec{z}\bar{k} = \bar{N}_k)] : \tau \end{aligned}$$

is derivable for some τ if and only if there is a τ_y with type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_n$ such that τ is a substitution instance of τ_y and

$$\{y : \forall\vec{\alpha}.\tau_y\} \supset \lambda\vec{x}.[\ldots]\bar{1} : \tau_y$$

is derivable in FMM. This, in turn, is derivable if and only if there are $\vec{\tau} = \tau_1 \to \ldots \to \tau_k$ and τ_M such that $\tau_y = \vec{\tau} \to \tau_M$ and

$$\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}\} \supset [\ldots]\bar{1} : \tau_M$$

are derivable. According to proposition 17, this is the case if and only if

$$\begin{aligned} &\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}\} \supset [\bar{M}_1,\ldots,\bar{M}_k] : \tau_M \text{and} \\ &\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}\} \supset [\lambda\vec{z}.(y\vec{z}\bar{1} = \bar{N}_1),\ldots,\lambda\vec{z}.(y\vec{z}\bar{k} = \bar{N}_k)] : \tau_= \end{aligned}$$

for some type $\tau_=$. Again, by proposition 17, this holds if and only if $\tau_M = (\tau_{M_1} \to \ldots \to \tau_{M_k} \to \tau_0) \to \tau_0$ and

$$\begin{aligned} &\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}\} \supset \bar{M}_i : \tau_{M_i}, i \in \{1,\ldots,k\}, \\ &\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}, \vec{z} : \vec{\tau^{(i)}}\} \supset y\vec{z}\bar{i} : \tau_{N_i}, i \in \{1,\ldots,k\}, \text{and} \\ &\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}, \vec{z} : \vec{\tau^{(i)}}\} \supset \bar{N}_i : \tau_{N_i} \end{aligned}$$

for some types $\tau_{M_1},\ldots,\tau_{M_k},\tau_{N_1},\ldots,\tau_{N_k}$ and suitable types $\vec{\tau^{(i)}} = \tau^{(i)}_1 \ldots \tau^{(i)}_m$. Note that, w.l.o.g.,

$$\{y : \forall\vec{\alpha}.\vec{\tau} \to \tau_M, \vec{x} : \vec{\tau}, \vec{z} : \vec{\tau^{(i)}}\} \supset y\vec{z}\bar{i} : \tau_{N_i}, i \in \{1,\ldots,k\}$$

holds if and only if

$$\{y : \forall\vec{\alpha}.\vec{\tau} \rightarrow \tau_M, \vec{x} : \vec{\tau}, \vec{z} : \vec{\tau^{(i)}}\} \supset y : \tau^{(i)} \rightarrow (\alpha_1 \rightarrow \ldots \alpha_k \rightarrow \alpha_i) \rightarrow \tau_{N_i}$$

Since the type of any occurrence of y must be a substitution instance of the type of y in the type assumption, it follows that τ_{N_i} must be a type instance of τ_{M_i}. It is easy to check that this is also sufficient. Since neither M_i nor N_i contain y or any of the z's, for any i, we can summarize that a necessary and sufficient condition for

$$\begin{array}{l} \{\} \supset \mathbf{fix}\, y.\lambda\vec{x}.K[\bar{M}_1, \ldots, \bar{M}_k], \\ \quad [\bar{M}_0 = \bar{N}_0, \lambda\vec{z}.(y\vec{z}\bar{1} = \bar{N}_1), \ldots, \lambda\vec{z}.(y\vec{z}\bar{k} = \bar{N}_k)] : \tau \end{array}$$

to be FMM-typable is that for some $\vec{\tau} = \tau_1 \ldots \tau_k$ and for some monotypes $\tau_{M_0}, \tau_{M_1}, \ldots, \tau_{M_k}, \tau_{N_0}, \tau_{N_1}, \ldots, \tau_{N_k}$ we have

$$\begin{array}{l} \{\vec{x} : \vec{\tau}\} \supset \bar{M}_i : \tau_{M_i}, 0 \leq i \leq k \\ \{\vec{x} : \vec{\tau}\} \supset \bar{N}_i : \tau_{N_i}, 0 \leq i \leq k \\ \tau_{M_0} = \tau_{N_0} \\ \tau_{M_i} = \tau_{N_i}, 1 \leq i \leq k \end{array}$$

Proof: (Proof of theorem continued)

With this lemma it is sufficient to show that whenever there is a solution of S then the above constraints can be satisfied, and vice versa.

($\Rightarrow$) Assume there is a solution σ of S. Let A_0 be a type environment that maps every variable in $\vec{x}$ into a distinct type variable. (Any other type environment that is injective on $\vec{x}$ will also do.) Now define $\tau_{M_i} = \tau(A, \sigma(M_i))$ for $0 \leq i \leq k$ where τ is the canonical type mapping from section 4.2.2, and let $\tau_i = \sigma(A_0)(x_i)$. We have

$$\begin{array}{l} \sigma(A_0) \supset \bar{M}_i : \tau(\sigma(A_0), M_i) \\ \sigma(A_0) \supset \bar{N}_i : \tau(\sigma(A_0), N_i) \end{array}$$

By lemma 10, we have $\tau(\sigma(A_0), M_i) = \tau(A_0, \sigma(M_i))$ and $\tau(\sigma(A_0), N_i) = \tau(A_0, \sigma(N_i))$. Since σ is a semi-unifier of S it furthermore follows for every i that there is a ρ_i such that $\rho_i(\sigma(M_i)) = \sigma(N_i)$. It is easy to show that the canonical type mapping τ above is monotonic (with respect to term subsumption) in its second argument.

($\Leftarrow$) Recall the function $\nu : M \rightarrow M$, which normalizes type expressions. Given types as required such that

$$\begin{array}{l} \{\vec{x} : \vec{\tau}\} \supset \bar{M}_i : \tau_{M_i}, 0 \leq i \leq k \\ \{\vec{x} : \vec{\tau}\} \supset \bar{N}_i : \tau_{N_i}, 0 \leq i \leq k \\ \tau_{M_0} = \tau_{N_0} \\ \tau_{M_i} = \tau_{N_i}, 1 \leq i \leq k \end{array}$$

by lemma 12 we know that

$$\begin{array}{l} \{\vec{x} : \nu(\vec{\tau})\} \supset \bar{M}_i : \nu(\tau_{M_i}), 0 \leq i \leq k \\ \{\vec{x} : \nu(\vec{\tau})\} \supset \bar{N}_i : \nu(\tau_{N_i}), 0 \leq i \leq k \end{array}$$

are also derivable. Following the proof of lemma 14, we can define a substitution σ (on terms). In the previous step we saw that τ is monotonic in its second argument. This argument can be strengthened to show, for injective A, $M_1 \leq M_2 \Leftrightarrow \tau(A, M_1) \leq \tau(A, M_2)$.

This completes the proof of the theorem

Corollary 19 *The following three problems are polynomial-time equivalent:*

1. *Typability in the Mycroft Calculus;*
2. *(nonuniform) semi-unifiability;*
3. *typability in the Flat Mycroft Calculus.*

Proof:

The steps (1) $\Rightarrow$ (2) and (2) $\Rightarrow$ (3) are proved in theorems 8 and 12; (3) $\Rightarrow$ (1) is trivial since FMM-typability instances are a subclass of MM-typability instances.

This corollary stands in contradiction to a statement by Mycroft who suggests prohibiting nested polymorphically typed **fix**-definitions "due to the exponential cost of analysing nested **fix** definitions" [85]. Indeed nesting does not make things any worse than they already are in a single **fix** definition.

Corollary 20

1. *The Milner Calculus is polynomial-time reducible to the Flat Mycroft Calculus.*
2. *The Flat Mycroft Calculus is PSPACE-hard.*

Proof:

1. By theorem 8 the Milner Calculus is polynomial-time reducible to semi-unification, which in turn is polynomial-time reducible to the Flat Mycroft Calculus by theorem 12.
2. By (1) and the PSPACE-hardness result of Kanellakis and Mitchell [53] for the Milner Calculus.

4.3 Type Inference in B and Semi-Unification

The programming language B [75] (now called ABC) has a polymorphic typing rule for recursive definitions and relies on type *inference* to determine the type correctness of programs. In contrast, Hope [8] also has a polymorphic typing rule for recursively defined functions, but mandates that their types be explicitly declared.

Even though there is no type inference system that specifies "logically" type correctness in B, it is clear from algorithm AA in [74] that the type computed in a recursive definition is the principal type in a "Flat" Milner-Mycroft style typing system. In fact, AA can be viewed as a variant of Mycroft's semi-algorithm for computing principal types in the Milner-Mycroft Calculus [85]. AA is provably nonterminating, and Meertens proceeds to refine it by adding a criterion reminiscent of our extended occurs check, but actually of much broader applicability, that guarantees termination of the resulting algorithm. Meertens argued that the absence of higher-order functions, nesting, and recursive types in B permitted uniform termination of his type inference algorithm.

Higher-order functions and their typing requirements usually create syntactic and semantic problems due to the fact that they are nonmonotonic in their domain types. This is of significance in the Second Order λ-calculus, but not in the Milner-Mycroft Calculus (since argument types cannot be *required* to be polymorphic). In the previous sections we have seen that nesting of definitions does not greatly change basic questions of type inference. This suggests that the type inference problem in B is actually no simpler than type inference in the Flat Mycroft Calculus and the complete Milner-Mycroft Calculus, in contrast to Meertens' general considerations. Indeed we shall show that semi-unification can be reduced to type inference for a small subset of B, which substantiates that neither higher-order functions nor nested definitions greatly influence the type inference problem. Since B has a syntactically simpler type system than MM the converse reduction is immediate. This implies that Meertens' uniformly terminating algorithm either proves decidability of semi-unification or it is not correct. In fact we shall show that Meertens' algorithm errs on the safe side. There are cases where the algorithm flags type-incorrectness while in fact there is a derivable typing for it (and AA would compute it).

In subsection 4.3.1 we introduce Pure B and its typing system. In subsection 4.3.2 we show that type inference in B and semi-unification are polynomial time equivalent. We also explain Meertens' termination criterion in terms of a criterion for our semi-unification algorithm (without extended occurs check) and give an example, both as a semi-unification problem and as a Pure B program, that shows where that criterion errs.

4.3.1 Pure B

Pure B is only a minute subset of B, yet big enough to capture the power of the polymorphic typing rule for recursive definitions. Pure B programs are given by the following grammar.

$$\begin{array}{ll} p ::= & \text{HOW'TO } x \text{ OF } x' : c \\ c ::= & x \text{ OF } e \mid cc \mid \text{PUT } e \text{ in } e \\ e ::= & x \mid (e, e) \end{array}$$

where x ranges over a predefined lexical category of identifiers. Even though the semantics of a language, as we have seen, is not all necessary to explain B's typing discipline, it helps to get an intuition for it. HOW'TO x OF x' : c defines a procedure x with formal (variable) parameter x'. The body of the procedure is the command c. A command is either the application of a procedure to an expression, xe, or a sequence of commands. An expression is either a variable or a pair of expressions. The PUT command copies its first argument to the second argument by pattern matching. This is all we need in Pure B to encode semi-unification. B, of course, has more complicated control structures and data types that make

Let A range over type environments; x over variables; e, e' over λ-expressions; t over type variables; τ, τ' over monotypes; σ, σ' over polytypes. The following are type inference axiom and rule schemes for Pure B.

Name	Axiom/rule
(TAUTD)	$A\{x : \tau\} \supset x : \tau$
(TAUTP)	$A\{x : \forall \vec{t}.\tau \to \text{unit}\} \supset x : \tau[\vec{\tau}/\vec{t}] \to \text{unit}$
(PUT)	$\dfrac{\begin{array}{l} A \supset e : \tau \\ A \supset e' : \tau \end{array}}{A \supset \text{PUT } e \text{ IN } e'}$
(PAIR)	$\dfrac{\begin{array}{l} A \supset e : \tau \\ A \supset e' : \tau' \end{array}}{A \supset (e, e') : (\tau, \tau')}$
(APPL)	$\dfrac{\begin{array}{l} A \supset x : \tau \to \text{unit} \\ A \supset e : \tau \end{array}}{A \supset x \text{ OF } e}$
(SEQ)	$\dfrac{\begin{array}{l} A \supset c \\ A \supset c' \end{array}}{A \supset cc'}$
(PROG)	$\dfrac{\begin{array}{l} A\{x : \forall \vec{\alpha}.\tau \to \text{unit}\} \supset c \\ A\{x : \forall \vec{\alpha}.\tau \to \text{unit}\} \supset x' : \tau \end{array}}{\supset \text{HOW'TO} x \text{ OF } x' : c}$

Table 4.4: Type inference axioms and rules for Pure B

it usable in practice.

The type expressions in Pure B are defined by

$$\begin{array}{rcl} \tau & ::= & t \mid (\tau, \tau') \\ \sigma & ::= & \tau \to \text{unit} \mid \forall t.\sigma \end{array}$$

Once again, we shall say type expressions derivable from τ are monotypes, and type expressions derivable from σ are polytypes. The type expression $\tau \to$ unit denotes the type of procedures whose argument is of type τ. Note that procedures are strictly first-order, since they can only take inputs whose types are built up from type variables and pairing and that only procedures can be polymorphic.[10] The typing rules of Pure B are given in Table 4.4.

A Pure B program p is typable if $\supset p$ is derivable in the type inference system for Pure B.

[10] This is, in general, an inessential restriction since polymorphic "data" such as "nil" in Pascal can be treated as nullary polymorphic functions.

4.3.2 Equivalence of Pure B and Semi-Unification

Recall the reduction of semi-unification to the Flat Milner-Mycroft Calculus. To facilitate the encoding of a term inequality in Pure B we need a representation of first-order terms by Pure B expressions whose types correspond to these terms, and an encoding of equations between the types representing terms (the PUT command will do). Term subsumption inequalities can be represented by the polymorphic typing rule for Pure B procedures. Indeed these are all the ingredients we need, and they are readily available in the type system for Pure B.

Consider, as usual, first-order terms over the ranked alphabet $\mathcal{A}_2$. Define the encoding function $\rho : T(\mathcal{A}_2, V) \rightarrow E$, where E denotes the Pure B expressions, as follows.

$$\begin{aligned} \rho(x) &= x, \text{if } x \in V \\ \rho(f(M,N)) &= (\rho(M), \rho(N)), \text{otherwise} \end{aligned}$$

We shall denote $\rho(M)$ simply by $\bar{M}$. Given a term inequality $M \leq N$ the Pure B program

```
HOW'TO $p$ OF $x$:
    PUT $x$ in $\bar{M}$
    $p$ OF $\bar{N}$
```

where x and p are identifiers not occurring in M or N, is typable if and only if $M \leq N$ is solvable. More generally, the program

```
HOW'TO $p$ OF $x$:
    PUT $x$ IN $(\bar{M}_1, \ldots, \bar{M}_k)$
    PUT $\bar{M}_0$ IN $\bar{N}_0$
    $p$ OF $(\bar{N}_1, y_1^{(2)}, \ldots, y_1^{(k)})$
    ...
    $p$ OF $(y_i^{(1)}, \ldots, y_i^{(i-1)}, \bar{N}_i, y_i^{(i+1)}, \ldots, y_i^{(k)})$
    ...
    $p$ OF $(y_k^{(1)}, \ldots, y_k^{(k-1)}, \bar{N}_k)$
```

with p, x, and additional Pure B variables $y_i^{(j)}, i \neq j$ not occurring in any of the M_i or N_i, is typable if and only if the SEI $S = (M_0 = N_0, M_1 \leq N_1, \ldots, M_k \leq N_k)$ is solvable.

Theorem 13 *Typability in Pure B and semi-unification are polynomial-time equivalent.*

Proof:

If we choose the encodings described above then the reductions can be easily adapted from the general reductions from MM to semi-unification, and from semi-unification to FMM.

Meertens' non-terminating algorithm AA computes the principal type for Pure B in the sense that it computes a type expression σ for the procedure p defined in HOW'TO p OF x: c such that this definition is typable and for any other type derivation the type of p, σ', will be such that $\sigma \sqsubseteq \sigma'$ in the generic instance preordering of chapter 2.

Instead of explaining algorithm AA and the refinement that results in a uniformly terminating algorithm we shall translate the termination criterion for AA into a termination criterion for our algorithm A and explain its effects in terms of semi-unification. For this, we assume the reader is familiar with the material in chapter 6. The independent sources of every arrow graph in an execution of algorithm A are nodes that are already present in the "initial" arrow graph that represents a given SEI; i.e., the independent sources of any node are "original" nodes. Meertens' "second circularity check" [74, p. 272] can be translated into a circularity check for algorithm A as follows. For any arrow graph G_i in an execution $(G_1, \ldots, G_i, \ldots)$ of algorithm A, let us define a directed graph $CV_i = (N, E_i)$ where N is the set of original nodes (i.e., the nodes in G_1) and $(n, n') \in E_i$ if and only if there are nodes m, m' in G_i such that n is a source of m, n' is a source of m' and m is a parent of m'. If, for some G_i, the digraph CV_i contains a proper cycle (i.e., nodes $n_1, \ldots, n_k, k \geq 2$, such that $n_1 = n_k, (n_i, n_{i+1}) \in E_i$ for $1 \leq i \leq k-1$, and the nodes $n_1, \ldots, n_{k-1}$ are pairwise distinct), then terminate the execution and signal unsolvability. Clearly, this criterion subsumes our extended occurs check since every time the occurs check is applicable and reduction to $\Box$ takes place, this circularity check is also applicable and unsolvability — reduction to $\Box$ — is indicated.

This algorithm is sound in the sense that whenever it produces a normal arrow graph that is not $\Box$, the input SEI is solvable. Furthermore, by analogy with Meertens' proof of termination, algorithm A with the extended occurs check replaced by the above circularity check is uniformly terminating. Unfortunately, though, the circularity check is too restrictive, and the resultant algorithm is incomplete: there are solvable SEI's that "trigger" the circularity check. A simple example is the SEI

$$S = (g(x) = y, g(z) \leq x, g(z) \leq y)$$

where g is a unary functor.[11] It is clearly solvable, the substitution $v = \{x \mapsto g(z'), y \mapsto g(g(z'))\}$ being a most general semi-unifier, yet, since z is source of both z' and $g(z')$ in this semi-unifier, the CV-graph contains a proper loop from (the node containing) z to z itself.

This SEI can be translated into a Pure B program via the encoding above and submitted for type checking by the B type inference system. According to the typing discipline described in [74] and partially formalized by the typing rules for Pure B in Table 4.4, the resulting program should be considered type correct, but the type inference algorithm with the "second circularity check" should flag a type error. At present we have not reconfirmed this with the locally available B interpreter.

[11]Recall from chapter 4, section 4.2 that we can claim the existence of a functor of any arity in any nonlinear alphabet.

Chapter 5

Algebraic Properties of Semi-Unifiers

In chapter 4 we saw that semi-unification characterizes type inference both in the Mycroft Calculus and the Flat Mycroft Calculus. Since semi-unification has a simpler definitional structure than any of the typing problems we will shift our attention in designing algorithms to semi-unification. This is justified since every algorithm for semi-unification yields a type inference algorithm, and vice versa. Since SEI's have potentially many solutions it is not *a priori* clear which one of the solutions an algorithm should compute. Naturally we expect to find an analog of the principal typing property for semi-unification: that every solvable SEI has a most general semi-unifier that is unique in some sense. In this chapter we shall see in which sense there are indeed unique most general semi-unifiers — and in which there are not.

A correct treatment of the algebraic structure of semi-unifiers — solutions of term inequalities — is trickier than is apparent at first sight. This is evidenced by technically incorrect treatments and statements in the literature [88,15]. In this chapter we present some results on the algebraic structure of semi-unifiers. Our main goal is to convince the reader that, in the same fashion in which *strong* equivalence classes of idempotent substitutions (see below for definitions) characterize the solutions of term equations and vice versa (see theorem 17), the *weak* equivalence classes of all substitutions characterize the solutions of term inequalities and vice versa (see theorem 18). In particular, we cannot replace "strong" by "weak" in this statement. Two substitutions σ_1 and σ_2 are strongly equivalent if there are substitutions α and α' such that $\alpha \circ \alpha' = \iota$, where ι denotes the identity substitution, and $\alpha \circ \sigma = \sigma'$. Strong equivalence is the preferred formalization of the common phrase "equivalent up to renaming of variables" [14,63]. We will show that, unlike term equations, term inequalities do *not* have most general solutions that are unique modulo *strong* equivalence. A natural, weaker notion of equivalence, however, admits unique most general solutions and, more generally, induces a complete lattice onto the set of all solutions of term inequalities.

5.1 Generality of Substitutions

Henceforth let W denote an arbitrary, but fixed subset of V.

Definition 10 *(Generality, W-equivalence, strong equivalence)*

The preordering $\leq_W$ *of* generality *on* $\mathcal{S}^\omega$ *over* W *is defined by*

$$\sigma_1 \leq_W \sigma_2 \Leftrightarrow (\exists \rho \in \mathcal{S}^\omega)\,(\rho \circ \sigma) \mid_W = \sigma_2 \mid_W .$$

The equivalence relation $\cong_W$ on $\mathcal{S}^\omega$ over W is defined by

$$\sigma_1 \cong_W \sigma_2 \Leftrightarrow \sigma_1 \leq_W \sigma_2 \wedge \sigma_2 \leq_W \sigma_1.$$

for all $\sigma_1, \sigma_2 \in \mathcal{S}^\omega$. We write $\sigma_1 <_W \sigma_2$ if $\sigma_1 \leq_W \sigma_2$, but $\sigma_1 \not\cong_W \sigma_2$. For any $\sigma \in \mathcal{S}^\omega$, $[\sigma]_W$ denotes the $\cong_W$-equivalence class of σ in $\mathcal{S}^\omega$. The equivalence relation $\cong_W$ is called W-equivalence*; if $W = V$, it is called* strong equivalence.

If $\sigma_1 \leq_W \sigma_2$ we say that σ_1 is *at least as general* as σ_2 on W. E. g., for $\sigma_1 = \{x \mapsto f(x)\}, \sigma_2 = \{x \mapsto f(y)\}, W \subset V - \{y\}$, the substitution σ_1 is at least as general as σ_2 on V, but σ_2 is only at least as general as σ_1 on W, not on V. Consequently, σ_1 and σ_2 are W-equivalent, but not strongly equivalent. For $M \leq N$ there is a unique most general substitution ρ, called the *quotient substitution* of N over M such that $\rho(M) = N$. We shall denote ρ by N/M.

Solutions of SEI's (semi-unifiers and unifiers) are closed with respect to "reasonable" substitution equivalences. More precisely we have

Proposition 21 *If $V(S) \subset W \subset V$ then for any σ_1 and σ_2 such that $\sigma_1 \cong_W \sigma_2$ we have*

1. $\sigma_1 \in \mathbf{U}(S) \Leftrightarrow \sigma_2 \in \mathbf{U}(S)$
2. $\sigma_1 \in \mathbf{USU}(S) \Leftrightarrow \sigma_2 \in \mathbf{USU}(S)$
3. $\sigma_1 \in \mathbf{SU}(S) \Leftrightarrow \sigma_2 \in \mathbf{SU}(S)$

Thus the solutions of any SEI S are closed with respect to equivalence relation $\cong_W$ as long as W contains at least all variables occurring in S, and every unifier/uniform semi-unifier/semi-unifier can viewed as (a representative) of a whole equivalence class of solutions.

As in the case of terms, the preordering $\leq_W$ induces a partial order on $\mathcal{S}^\omega/_{\cong_W} = \{[\sigma]_W \mid \sigma \in \mathcal{S}^\omega\}$, denoted also by $\leq_W$. Since the definitions of term subsumption $(T^\Omega, \leq)$ and of generality of substitutions $(\mathcal{S}^\omega, \leq_W)$ appear analogous we can ask whether an analog of theorem 4 holds for substitutions. The answer to this question is three quarters positive, one quarter negative: the analog of theorem 4, part 1, holds (see theorem 14), but the analog of part 2 holds if W is co-infinite (with respect to V) (see theorem 16) or $\mathcal{A}$ is linear (see theorem 15). These results are presented below.

Eder proved that $(\mathcal{S}^\omega, \leq_V)$ is Noetherian [26, corollary 2.19]. Although it is not an immediate consequence that $(\mathcal{S}^\omega, \leq_W)$ is Noetherian where W is *any* subset of V, Eder's proof can be easily adapted to take care of this case, too.

Theorem 14 *$(T^\omega/_{\cong_W}, \leq_W)$ is Noetherian for any $W \subset V$.*

Proof: See [26,45,46].

As already indicated the analog of theorem 4, part 2 holds if $\mathcal{A}$ is linear.

Theorem 15 *If $\mathcal{A}$ is linear then $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ is a complete lattice.*

Here and later we shall make use of Huet's anti-unification algorithm [46,63]. The recursive algorithm **mscai** on $T \times T$ is defined recursively in Figure 5.1.

It is easy to show that **mscai**(M, N) computes a most specific common anti-instance [63, theorem 5.8].

Let Φ be a bijection between $T \times T$ and V.

$$\mathbf{mscai}(M,N) = \left\{ \begin{array}{l} f(\mathbf{mscai}(M_1,N_1),\ldots,\mathbf{mscai}(M_k,N_k)), \\ \quad \text{if } M = f(M_1,\ldots,M_k), N = f(N_1,\ldots,N_k) \\ \Phi(M,N), \text{otherwise} \end{array} \right.$$

Figure 5.1: Anti-unification algorithm

Proof: (Proof of theorem)

In view of theorem 14 it is sufficient to show that $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ is a lower semi-lattice. We shall only show this here for $W = V$, which is the most interesting case anyway.

For a finite set of variables X the notation $\vec{X}$ shall denote a term with a "new" $|X|$-ary functor whose arguments are the distinct elements of X in some order determined by X. For finite X, for example, $\sigma(\vec{X})/\vec{X}$ is another way of writing $\sigma|_X$.

Let σ_1, σ_2 be two proper substitutions. We will first construct a substitution σ and then show that σ is a lower bound and that any other lower bound is at least as general as σ.

Define $Z = \mathbf{dom}\sigma_1 \cup \mathbf{dom}\sigma_2, X = \{x \in Z : \sigma_1(x) = \sigma_2(x)\}, Y = \{x \in Z : \sigma_1(x) \neq \sigma_2(x)\}$, and let $\bar{M} = \mathbf{mscai}(\sigma_1(\vec{Y}), \sigma_2(\vec{Y}))$. Since $\mathcal{A}$ is linear we have for the variables in the range of σ_1 (or σ_2) under X, $V' = V(\sigma_1(X))$, that $|V'| \leq |X|$ and consequently, with $V'' = (X \cup Y) - V'$, that $|V''| \geq |Y|$. Thus there is $\bar{M}'$ such that $\bar{M}' \cong \bar{M}$ and $V(M') \subset V''$. Now we can construct σ as

$$\sigma(x) = \left\{ \begin{array}{ll} \sigma_1(x), & x \in X \\ (\bar{M}'/\vec{Y})(x) & \text{otherwise} \end{array} \right.$$

Notice that $\sigma(\vec{Z}) \leq \sigma_1(\vec{Z})$ and thus $\rho_1 = \sigma_1(\vec{Z})/\sigma(\vec{Z})$ is well-defined. Furthermore, $\mathbf{dom}\,\rho_1 \subset Z$. Since $\rho_1 \circ \sigma = \sigma_1$ this shows that σ is a lower bound of σ_1 with respect to $\leq_V$. Similarly σ is a lower bound of σ_2.

To see that σ is a greatest (most specific) lower bound consider another lower bound σ' of σ_1, σ_2. Define $Z' = Z \cup \mathbf{dom}\,\sigma'$. It is easy to see that $\sigma(\vec{Z'})$ is a most specific common anti-instance of $\sigma_1(\vec{Z'})$ and $\sigma_2(\vec{Z'})$. Consequently $\delta = \sigma(\vec{Z'})/\sigma'(\vec{Z'})$ is well-defined. Furthermore we have, for $x \in X$, $\sigma'(x) = \sigma(x)$ or $V(\sigma'(x)) \subset Z'$, and, for $x \in Z' - X$, it is always the case that $V(\sigma'(x)) \subset Z'$ since otherwise the domain of either σ_1 or σ_2 would have to contain an element from outside Z' (being that σ' is a lower bound of both of them by assumption); but this cannot be as Z' contains the domains of both σ_1 and σ_2 by construction. This, in turn, implies that $\mathbf{dom}\,\delta \subset Z'$ and thus $\delta \circ \sigma' = \sigma$, which is to say $\sigma' \leq_V \sigma$.

For nonlinear $\mathcal{A}$ this structure theorem fails in a major way if W is co-finite: $\mathcal{S}^\omega/_{\cong_W}$ is neither a lower nor an upper semi-lattice under the partial order $\leq_W$ if $|V - W| < \infty$. This shall be proved in the following two propositions.

Proposition 22 *For nonlinear $\mathcal{A}$, if W is co-finite, $|V - W| < \infty$, then there is a pair of substitutions $\sigma_1, \sigma_2 \in \mathcal{S}$ with two minimal upper bounds $v_1, v_2 \in \mathcal{S}$ with respect to $\leq_W$ such that $v_1 \not\cong_W v_2$.*

Proof: Eder [26] shows that the pair of substitutions

$$\{x \mapsto f(x, f(y, z)), y \mapsto f(x, f(y, z)), z \mapsto f(x, f(y, z))\}$$

and

$$\{x \mapsto f(f(x, y), z), y \mapsto f(f(x, y), z), z \mapsto f(f(x, y), z)\}$$

has an infinite set of minimal upper bounds, but no least upper bound with respect to $\leq_V$.

A simple generalization of Eder's pair will do the trick. Let W be a co-finite set. Without loss of generality we can assume that $V - W = \{w_1, \ldots, w_n\}$ for some n and that $\{x_1, \ldots, x_{n+1}, y_1, \ldots, y_{n+1}, z_1, \ldots, z_{n+1}\}$ is a subset of W. Now with $\rho_i = \{x_i \mapsto f(x_i, f(y_i, z_i)), y_i \mapsto f(x_i, f(y_i, z_i)), z_i \mapsto f(x_i, f(y_i, z_i))\}$ and $\sigma_i = \{x_i \mapsto f(f(x_i, y_i), z_i), y_i \mapsto f(f(x_i, y_i), z_i), z_i \mapsto f(f(x_i, y_i), z_i)\}$ consider the substitutions

$$\rho = \cup_{i \in \{1, \ldots, n+1\}} \rho_i$$

and

$$\sigma = \cup_{i \in \{1, \ldots, n+1\}} \sigma_i$$

.[1]

The minimal upper bounds of ρ and σ are the substitutions

$$\cup_{i \in \{1, \ldots, n+1\}} \quad \begin{array}{l} \{x_i \mapsto f(f(s_i, t_i), f(u_i, v_i)), \\ y_i \mapsto f(f(s_i, t_i), f(u_i, v_i)), \\ z_i \mapsto f(f(s_i, t_i), f(u_i, v_i))\} \end{array}$$

for pairwise distinct variables $W' = \{s_1, t_1, u_1, v_1, \ldots, s_{n+1}, t_{n+1}, u_{n+1}, v_{n+1}\}$. Consider one such minimal upper bound, say τ_1. Simple counting shows that there must be some variable $w \in W'$ such that

$$w \notin \{w_1, \ldots, w_n, x_1, \ldots, x_{n+1}, y_1, \ldots, y_{n+1}, z_1, \ldots, z_{n+1}\}.$$

Thus w is in W. If we consider another minimal upper bound, τ_2, with range variables

$$V(\tau_2(\{x_1, \ldots, x_{n+1}, y_1, \ldots, y_{n+1}, z_1, \ldots, z_{n+1}\}))$$

disjoint from

$$V(\tau_1(\{x_1, \ldots, x_{n+1}, y_1, \ldots, y_{n+1}, z_1, \ldots, z_{n+1}\})),$$

then it is clear that $\tau_1 \not\leq_W \tau_2$ because $w \notin V(\tau_2(\mathbf{dom}\,(\tau_2)))$.

[1] More formally, $\rho = \rho_1 \circ \ldots \circ \rho_{n+1}$ and $\sigma = \sigma_1 \circ \ldots \circ \sigma_{n+1}$. Since the order of composition is insignificant the informal set union operation on the canonical representations of the ρ_i's and σ_i's is well-defined.

This shows that $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ is not an upper semi-lattice for $|V - W| < \infty$. We can also show that it fails to be a lower semi-lattice.

Proposition 23 *For nonlinear $\mathcal{A}$, if W is co-finite, $|V - W| < \infty$, then there is a pair of substitutions $\sigma_1, \sigma_2 \in \mathcal{S}$ with two maximal lower bounds $\upsilon_1, \upsilon_2 \in \mathcal{S}$ with respect to $\leq_W$ such that $\upsilon_1 \not\cong_W \upsilon_2$.*

Proof:

We shall only treat the case $W = V$. The general case is a generalization analogous to the previous proof.

Let $y_1, \ldots, y_4, z_1, \ldots, z_4$ be eight pairwise distinct variables and let f be an arbitrary functor with arity 2.[2] Consider

$$\begin{aligned} \sigma_1 \quad = \quad & \{x_1 \mapsto f(f(y_1, y_2), f(y_3, y_4)), \\ & x_2 \mapsto f(f(y_1, y_2), f(y_3, y_4)), \\ & x_3 \mapsto f(f(y_1, y_2), f(y_3, y_4))\} \end{aligned}$$

and

$$\begin{aligned} \sigma_2 \quad = \quad & \{x_1 \mapsto f(f(z_1, z_2), f(z_3, z_4)), \\ & x_2 \mapsto f(f(z_1, z_2), f(z_3, z_4)), \\ & x_3 \mapsto f(f(z_1, z_2), f(z_3, z_4))\}. \end{aligned}$$

Both

$$\begin{aligned} \upsilon_1 \quad = \quad & \{x_1 \mapsto f(x_1, f(x_2, x_3)), \\ & x_2 \mapsto f(x_1, f(x_2, x_3)), \\ & x_3 \mapsto f(x_1, f(x_2, x_3))\} \end{aligned}$$

and

$$\begin{aligned} \upsilon_2 \quad = \quad & \{x_1 \mapsto f(f(x_1, x_2), x_3), \\ & x_2 \mapsto f(f(x_1, x_2), x_3), \\ & x_3 \mapsto f(f(x_1, x_2), x_3)\} \end{aligned}$$

are maximal lower bounds since, at first sight maybe somewhat unexpectedly,

$$\{x_1, x_2, x_3 \mapsto f(f(v_1, v_2), f(v_3, v_4))\}$$

does not form a lower bound of σ_1 or σ_2 for any variables $v_1, \ldots, v_4$. Clearly, υ_1 and υ_2 are not equivalent under $\cong_V$.

[2]Note that there must be at least one functor with arity at least 2 since we assume that (F, a) is nonlinear in this section; w. l. o. g. we can assume that F contains a functor with arity exactly 2.

The reason for this "misbehavior" of $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ for co-finite W is due to the fact that we cannot "hide" enough variables from "consideration" under $\leq_W$ if there is not enough "room" in $V - W$. For subsets W of V that leave "enough" variables hidden in $V - W$ — for co-infinite W's — the partial orders $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ have indeed a lattice structure. The proof of this is a consequence of the more general theorem 18 proved in section 5.2.

Theorem 16 *For nonlinear $\mathcal{A}$ the following statements are equivalent.*

- $(\mathcal{S}^\omega/_{\cong_W}, \leq_W)$ *is a complete lattice.*
- W *is co-infinite; that is,* $|V - W| = \infty$.

Henceforth we shall deal almost exclusively with nonlinear alphabets. As we have already seen the theory of substitutions and (semi-)unifiers is very different for linear and nonlinear alphabets. In fact, the case of term inequalities over linear alphabets is algebraically and computationally much simpler than for nonlinear alphabets. It is treated in [15] under the name *prefix inequalities.*

5.2 The Structure of Semi-Unifiers

It is often quoted that most general unifiers are unique "up to renaming of variables". As pointed out in [63] there are several *distinct* notions of what this innocuous-looking little phrase can be taken to mean. The most commonly used notion is strong equivalence (i.e., equivalence modulo $\cong_V$). While different notions lead to a slightly different structure of unifiers for a given system of equations, they all admit the existence of most general unifiers, though most general unifiers with respect to one notion (e.g., [113]) are not necessarily most general with respect to another equivalence.

The fact that there are most general unifiers under any of the different notions of renaming may have prompted Chou to write that, similarly, "it is evident" that the most general semi-unifier of an SEI is unique modulo strong equivalence, if it exists at all [15, page 11]. The breakdown in the analogy of the structure of $T/_{\cong}$ and $S/_{\cong_V}$ (see theorem 16 and the discussion before it), however, already suggests that this claim may not be true in general, and, indeed, it is incorrect.[3] A weaker notion of equivalence (see, e.g. [113, chapter 4]), however, admits the existence of most general semi-unifiers and an equivalent to the main structure theorem for unifiers.

5.2.1 Strong Equivalence

Strong equivalence, $\cong_V$, corresponds to renaming of substitutions by composition of permutation substitutions; i.e., by substitutions α for which there is α^{-1} such that $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = \iota$. Two substitutions σ_1 and σ_2 are strongly equivalent if and only if there is such a permutation substitution α such that $\alpha \circ \sigma_1 = \sigma_2$. Strong equivalence has attracted a lot of attention because of its close connection to idempotent substitutions, which in turn are strongly related to systems of equations.

In this subsection the terms "minimal" and "most general" always refer to $\leq_V$.

Strong Equivalence and Unifiers

We recapitulate the most important result on the structure of unifiers modulo strong equivalence from [26] (see also [63]). Note that every SEI has a minimal unifier .[4] This follows immediately from theorem

[3] We feel tempted to say that, in view of theorem 16, uniqueness of most general unifiers with respect to strong equivalence is a "lucky coincidence"; or, less dramatically, a very specific property of unification that cannot simply be "transferred" to other problems; or, in more neutral terms, an outgrowth of the fact that the theory of unifiers can be viewed as a representation theory for idempotent substitutions, which indeed form a lattice with respect to $\leq_V$ [26, theorem 4.9].

[4] A unifier σ of an SEI S is minimal if for every other unifier σ' of S it holds that $\sigma' \leq \sigma \Rightarrow \sigma \leq \sigma'$.

14. We call a minimal unifier σ of S a *most general* unifier of S if for all unifiers υ of S there is a substitution ρ such that $\rho \circ \sigma = \upsilon$.

A substitution σ is *idempotent* if it satisfies $\sigma \circ \sigma = \sigma$. The set of proper idempotent substitutions is denoted by $\mathcal{IS}(\mathcal{A}, V)$ (or just $\mathcal{IS}$), and the set of all idempotent substitutions is denoted by $\mathcal{IS}^{\omega}(\mathcal{A}, V)$ (or simply $\mathcal{IS}^{\omega}$). The significance of idempotent substitutions and their relation to unification is summarized in the main structure theorem of systems of equations.

Theorem 17 *1. Every system of equations S has a most general unifier that is idempotent, and for every idempotent substitution σ there is a system of equations S' such that σ is a most general unifier of S' (with respect to $\leq_V$).*

2. $((\mathcal{IS}^{\omega} \cap \mathbf{U}(S))/_{\cong_V}, \leq_V)$ is a complete lattice for every system of equations S.

Proof: By refinement of the proof of theorem 4.9 in [26].

Since there are substitutions that are not strongly equivalent to any idempotent substitution, we have as a consequence of theorem 17 that there are substitutions in $\mathcal{S}$ that are not most general unifiers. For example, $\{z_1 \mapsto f(z_1), \ldots, z_n \mapsto f(z_n)\}$ is not strongly equivalent to any idempotent substitution.

Part 1 of theorem 17 expresses not only that every system of equations has a most general unifier, but that there is always an idempotent most general substitution. An instance of the theorem is Eder's original structure theorem for idempotent substitutions.

Corollary 24 *$(\mathcal{IS}^{\omega}/_{\cong_V}, \leq_V)$ is a complete lattice.*

Proof: Consider $S = ()$ in theorem 17.

Strong Equivalence and Semi-Unifiers

The set of idempotent unifiers of any system of equations forms a lattice. The fact that every system of equations has an idempotent most general unifier justifies in some sense the restriction of consideration to idempotent substitutions and unifiers, as is done from the outset in [104].

In this subsection we show that idempotent substitutions and strong equivalence fail to capture the structure of semi-unifiers in a major way; namely,

1. for any SEI S neither $\mathbf{U}(S)$ nor $\mathbf{USU}(S)$ nor $\mathbf{SU}(S)$ induce a lower or upper semi-lattice (under $\leq_V$).

2. there are systems of equations and inequalities that have a most general semi-unifier, but no idempotent one;

3. there are systems of equations and inequalities with no most general semi-unifier;

Proposition 25 *For nonlinear $\mathcal{A}$ neither $(\mathbf{U}(S)/_{\cong_V}, \leq_V)$ nor $(\mathbf{USU}(S)/_{\cong_V}, \leq_V)$ nor $(\mathbf{SU}(S)/_{\cong_V}, \leq_V)$ forms a lower or upper semi-lattice for any SEI S.*

Proof: Almost directly from the proofs of propositions 22 and 23.

Proposition 26 *For nonlinear $\mathcal{A}$ there is an infinite family of SEI's $S_1, \ldots, S_i, \ldots$ such that, for all $i \in \mathcal{N}$, S_i has uniform and nonuniform minimal semi-unifiers σ_{i1} and σ_{i2}, but $\sigma_{i1} \not\cong_V \sigma_{i2}$.*

Proof:

Consider $S_i = (f(x_1, \ldots, x_i) \leq y)$. The substitutions

$$\sigma_{i1} = \{y \mapsto f(u_1, \ldots, u_i)\}$$

and

$$\sigma_{i2} = \{y \mapsto f(v_1, \ldots, v_i)\}$$

are minimal semi-unifiers of S_i since only for $\rho = \{\}$ we have $\rho <_V \sigma_{i1}$ or $\rho <_V \sigma_{i2}$ and $\{\}$ is not a semi-unifier of S_i. But there is no substitution $\alpha \in S$ such that $\alpha \circ \sigma_{i1} = \sigma_{i2}$ or $\alpha \circ \sigma_{i2} = \sigma_{i1}$.

Proposition 27 *There is an infinite family of SEI's $S_1, \ldots, S_i, \ldots$ such that, for all $i \in \mathcal{N}$, S_i has a most general uniform and nonuniform semi-unifier, but no idempotent one.*

Proof:

Consider $S_i = (f(y_1) \leq z_1, \ldots, f(y_i) \leq z_i)$. The substitution

$$\sigma_i = \{z_1 \mapsto f(z_1), \ldots, z_i \mapsto f(z_i)\}$$

and its $\cong_V$-equivalent substitutions are the only most general uniform and nonuniform unifiers of S_i. As we remarked earlier there is no idempotent substitution amongst them.

The reason why $\mathcal{S}^\omega, \mathbf{U}(S), \mathbf{USU}(S), \mathbf{SU}(S)$ fail to be lattices under $\leq_V$ are intuitively rather pathological and cast some doubt on the appropriateness of choosing strong equivalence as the "proper" notion of renaming on substitutions for semi-unification.

5.2.2 Weak Equivalence

In this section we define an equivalence relation on substitutions relative to systems of equations and inequalities that is properly weaker than strong equivalence. We will show that this relation, weak equivalence, ties general substitutions and systems of equations and inequalities together just as strong equivalence ties idempotent substitutions and systems of equations together (theorem 17).

Definition 11 *(Weak equivalence)*
Substitutions σ_1 and σ_2 are called weakly equivalent *with respect to SEI S (or simply S-*equivalent*) if $\sigma_1 \cong_{V(S)} \sigma_2$.*

A k-ary *context* $C[]$ is a term with k "holes" in it such that $C[M_1, \ldots, M_k]$ is the (complete) term with the terms $M_1, \ldots, M_k$ in place of the holes in $C[]$. More formally, a k-ary context is a term $C[] \in T(\mathcal{A}, V \cup MV)$ where MV is a k-element set $\{y_1, \ldots, y_k\}$ of *meta-variables* disjoint from V and F. For subsitution $\sigma : V \cup MV \mapsto T(\mathcal{A}, V), \sigma = \{y_1 \mapsto M_1, \ldots, y_k \mapsto M_k\}$ the result of applying σ to C is denoted by $C[M_1, \ldots, M_k]$.

Recall that $(T^\Omega/_\cong, \leq)$ is a complete lattice with $\wedge$ and $\vee$ denoting the infimum and supremum operator, respectively.

Lemma 28 *There is an operation $\wedge : T \times T \mapsto T$ such that*

1. *$[M \wedge N] = [M] \wedge [N]$ for all $M, N \in T$.*
2. *$C[M_1, \ldots, M_k] \wedge C[M'_1, \ldots, M'_k] = C[M_1 \wedge M'_1, \ldots, M_k \wedge M'_k]$ for all k, k-ary contexts C, and terms $M_1, \ldots, M_k$ and $M'_1, \ldots, M'_k$.*

Proof:

1. Consider the anti-unification algorithm in Figure 5.1 and define $M \wedge N = \mathbf{mscai}(M, N)$. (See [46]; see also [63].)
2. The definition of $\wedge$ has the property $f(M) \wedge f(N) = f(M \wedge N)$ for every functor f. The result follows by structural induction on $C[]$.

For every operation that satisfies lemma 28, part 1, the following proposition holds.

Proposition 29 *For all terms $M_1, M_2, N_1, N_2 \in T$ such that $M_1 \leq M_2$ and $N_1 \leq N_2$ it holds that $M_1 \wedge N_1 \leq M_2 \wedge N_2$.*

For any SEI S, we call a (uniform) semi-unifier σ of S a *most general* (uniform) semi-unifier of S if for all (uniform) semi-unifiers υ of S there is a substitution ρ such that $(\rho \circ \sigma) \mid_{V(S)} = \upsilon \mid_{V(S)}$. Similarly, from now on a unifier of S will be called most general if it is minimum with respect to $\leq_{V(S)}$ instead of $\leq_V$ as in the previous section.

Now we are ready to prove the main theorem of this section.

Theorem 18 *1. Every system of equations and inequalities S has a most general (uniform) semi-unifier, and for every substitution σ there is a system of equations and inequalities S' such that σ is a most general (uniform) semi-unifier of S.*

2. $(\mathbf{SU}(S)/_{\cong_{V(S)}}, \leq_{V(S)})$ (as well as $(\mathbf{USU}(S)/_{\cong_{V(S)}}, \leq_{V(S)})$ and $\mathbf{U}(S)/_{\cong_{V(S)}}$) is a complete lattice for every system of equations and inequalities S.

As an immediate consequence we have

Corollary 30 *Every solvable SEI S has a most general idempotent semi-unifier.*

Proof: (Proof of corollary)

Take a most general semi-unifier σ of S. If $V(S) = \{x_1, \ldots, x_n\}$ define $\sigma' = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\} \circ \sigma$ where $x'_1, \ldots, x'_n$ are pairwise distinct variables not occurring in S. Then σ' is idempotent and a most general semi-unifier of S.

The theorem can be strengthened and still holds if we replace $\cong_{V(S)}$ (weak equivalence) and $\leq_{V(S)}$ by $\cong_W$ and $\leq_W$, respectively, where W is any co-infinite subset of V containing $V(S)$. This strengthened version of theorem 18, part 2, implies theorem 16 (let $S = ()$).

Proof: (Proof of theorem)

For part 2, since every complete semi-lattice is automatically a complete lattice and since every Noetherian lower semi-lattice is a complete lower semi-lattice, it is sufficient to show that $(\mathbf{SU}(S)/_{\cong_{V(S)}}, \leq_{V(S)})$ is a lower semi-lattice.

Let σ_1 and σ_2 be semi-unifiers of S. Let $x_1, \ldots, x_k$ be the set $V(S)$ of variables occurring in S. Denote $\sigma_1(x_i)$ by M_i and $\sigma_2(x_i)$ by N_i for $1 \leq i \leq k$. Now define $\sigma = \{x_1 \mapsto M_1 \wedge N_1, \ldots, x_k \mapsto M_k \wedge N_k\}$ with $\wedge$ defined as in lemma 28.

First we show that σ is a semi-unifier of S. Without loss of generality (see proof of proposition 1) we can assume that S consists of one equation and n inequalities. There are contexts $C_0, C_1, \ldots, C_n$ and $C'_0, C'_1, \ldots, C'_n$ such that S is equal to

$$\{\ C_0[x_1,\ldots,x_k] \ = \ C'_0[x_1,\ldots,x_k]\ \} \qquad \text{(equation)}$$

$$\left\{ \begin{array}{rcl} C_1[x_1,\ldots,x_k] & \leq & C'_1[x_1,\ldots,x_k] \\ & \ldots & \\ C_n[x_1,\ldots,x_k] & \leq & C'_n[x_1,\ldots,x_k] \end{array} \right\} \quad \text{(inequalities)}$$

By assumption both σ_1 and σ_2 are semi-unifiers of S, and so

$$\begin{array}{rcl} C_0[M_1,\ldots,M_k] & = & C'_0[M_1,\ldots,M_k] \\ C_1[M_1,\ldots,M_k] & \leq & C'_1[M_1,\ldots,M_k] \\ & \ldots & \\ C_n[M_1,\ldots,M_k] & \leq & C'_n[M_1,\ldots,M_k] \end{array}$$

holds as well as

$$\begin{array}{rcl} C_0[N_1,\ldots,N_k] & = & C'_0[N_1,\ldots,N_k] \\ C_1[N_1,\ldots,N_k] & \leq & C'_1[N_1,\ldots,N_k] \\ & \ldots & \\ C_n[N_1,\ldots,N_k] & \leq & C'_n[N_1,\ldots,N_k] \end{array}$$

By proposition 29 this implies that

$$\begin{array}{rcl} C_0[M_1,\ldots,M_k] \wedge C_0[N_1,\ldots,N_k] & = & C'_0[M_1,\ldots,M_k] \wedge C'_0[N_1,\ldots,N_k] \\ C_1[M_1,\ldots,M_k] \wedge C_1[N_1,\ldots,N_k] & \leq & C'_1[M_1,\ldots,M_k] \wedge C'_1[N_1,\ldots,N_k] \\ & \ldots & \\ C_n[M_1,\ldots,M_k] \wedge C_n[N_1,\ldots,N_k] & \leq & C'_n[M_1,\ldots,M_k] \wedge C'_n[N_1,\ldots,N_k] \end{array}$$

holds, and by lemma 28, part 2, we conclude that

$$\begin{array}{rcl} C_0[M_1 \wedge N_1,\ldots,M_k \wedge N_k] & = & C'_0[M_1 \wedge N_1,\ldots,M_k \wedge N_k] \\ C_1[M_1 \wedge N_1,\ldots,M_k \wedge N_k] & \leq & C'_1[M_1 \wedge N_1,\ldots,M_k \wedge N_k] \\ & \ldots & \\ C_n[M_1 \wedge N_1,\ldots,M_k \wedge N_k] & \leq & C'_n[M_1 \wedge N_1,\ldots,M_k \wedge N_k] \end{array}$$

holds true. This, in turn, shows that σ is a semi-unifier of S.

We now show that any other semi-unifier σ' that is a lower bound of both σ_1 and σ_2 is also a lower bound of σ. Define $\sigma'(x_i) = L_i$ for $1 \leq i \leq k$. Since σ' is a lower bound of σ_1 (with respect to $\leq_{V(S)}$) it holds that $[L_1,\ldots,L_k] \leq [M_1,\ldots,M_k]$ for some arbitrary functor $[\ldots]$ written in bracket-notation; similarly, $[L_1,\ldots,L_k] \leq [N_1,\ldots,N_k]$. Consequently, $[L_1,\ldots,L_k] \leq [M_1,\ldots,M_k] \wedge [N_1,\ldots,N_k]$ and, by lemma 28, part 2, we have $[L_1,\ldots,L_k] \leq$

$[M_1 \wedge N_1, \ldots, M_k \wedge N_k]$; i.e., there is a substitution ρ such that $\rho([L_1, \ldots, L_k] = [M_1 \wedge N_1, \ldots, M_k \wedge N_k]$. But this immediately implies $\rho(\sigma'(x_i)) = \sigma(x_i)$ for $1 \leq i \leq k$, and thus $\sigma' \leq_{V(S)} \sigma$.

Part 2 implies one half of part 1, that every system of equations and inequalities has a most general semi-unifier. Conversely, let σ be an arbitrary substitution. If $\sigma = \omega$ then clearly σ is a most general semi-unifier of $\{f(x) = x\}$. If $\sigma = \{x_1 \mapsto M_1, \ldots, x_k \mapsto M_k\}$ let $\alpha = \{x_1 \mapsto x'_1, \ldots, x_k \mapsto x'_k\}$ where $x'_1, \ldots, x'_k$ are pairwise distinct variables disjoint from $x_1, \ldots, x_k$. Now define $S = \{\alpha(M_1) \leq x_1, \ldots, \alpha(M_k) \leq x_k\}$. Clearly, σ is a most general semi-unifier of S.

There are more constructive proofs of the uniqueness of most general semi-unifiers modulo weak equivalence, but they do not yield the powerful structure theorem 18. The algorithmic specifications (functional, rewriting, and graph-theoretic) for computing most general semi-unifiers in chapter 6, for example, can be turned independently into proofs of the existence of most general semi-unifiers. In fact their proofs of correctness constitute alternative proofs, although additional care is necessary since the specifications may not be uniformly terminating.

5.3 The Structure of Typings and Typing Derivations

It is interesting that the main structure theorem for semi-unifiers, theorem 18, yields a "simultaneous" proof of the principal typing property of CH, DM, MM, and FMM via the reduction in theorem 8 of chapter 4. Something even stronger can be said about typings and their derivations in the syntax-oriented versions of our type disciplines since the reduction in theorem 8 translates every typing derivation into a solution of the corresponding semi-unification problem instance.

Consider a substitution on monotypes, $S : TV \rightarrow M$. S can be applied to a polytype σ by simultaneously replacing only the free variables in σ all the while renaming bound type variables in σ to avoid capture of (necessarily free) type variables from S. Such a substitution can thus be extended to type assignments, $S(A)(x) = S(A(x)), x \in \mathbf{dom}\, A$, to typings, $S(A \supset e : \sigma) = S(A) \supset e : S(\sigma)$ and to whole derivation trees. We can also extend the generic instance preordering on polytypes of chapter 2, $\sigma_1 \sqsubseteq \sigma_2$, to type assignments by $A \sqsubseteq A' \Leftrightarrow (\forall x \in \mathbf{dom} A)\, A(x) \sqsubseteq A'(x)$. Finally, we define the relation $(A \supset e : \sigma) \leq (A' \supset e' : \sigma')$: it holds if and only if there is a substitution S such that

1. $S(A) \sqsubseteq A'$,
2. $e = e'$,
3. $S(\sigma) \sqsubseteq \sigma'$.

Finally for two proof trees (in a fixed typing calculus), P and P', we define $P \leq P'$ if P and P' are structurally isomorphic and there is a substitution S such that $(A \supset e : \sigma) \leq (A' \supset e' : \sigma')$ holds for every corresponding pair of typings $(A \supset e : \sigma) \in P$ and $(A' \supset e' : \sigma') \in P'$. Clearly $\leq$ defines a preorder that induces canonically a partial order, also denoted by $\leq$.

Corollary 31 *Let X = CH, DM, MM, or FMM, and let X' denote the syntax-oriented version of X. For any expression $e \in \Lambda$ that is X-typable,*

1. *the set of derivable typings for e in X, respectively X', forms a complete lattice w.r.t. the partial order $\leq$ on typings;*
2. *the set of all proof trees for e in X' forms a complete lattice w.r.t. the partial order $\leq$ on proof trees.*

Proof: Because of theorem 5 part (2) implies part (1). An inspection of the proofs of theorems 6 and 7 reveals that proof trees for e and solutions of the canonical system of equations and inequalities $SEI(e)$ are in a one-one correspondence and the composition of the two reductions is strongly monotonic in the sense that if P and P' are derivations for e and S and S' are the corresponding semi-unifiers of $SEI(e)$ then $P \leq P' \Leftrightarrow S \leq S'$.

The first part of this corollary implies that there is a least typing $A \supset e : \sigma$ for every typable e. This can be read as a generalized principal typing property since it is not relative to a fixed type assignment [23]. The second part may have practical applications in an incremental compiler: it should be quite practical to maintain the principal type information with any well-defined program fragment and perform a "meet"-operation with new type information once it is available. The corollary certifies that this is always possible (see also [74]).

Chapter 6

Algorithmic Specification of Most General Semi-Unifiers

In chapter 4 we showed that semi-unification is at the heart of polymorphic type inference in the Mycroft Calculus. In chapter 5 we saw that every system of equations and inequalities (SEI) has a most general semi-unifier, which is unique up to weak equivalence. In this chapter we address the problem of computing most general semi-unifiers. It appears natural to expect that in order to solve the decision problem of semi-unifiability it is essentially necessary to compute most general semi-unifiers since they represent the least commitment to substitution decisions. It is interesting then to see that Kapur *et al.* achieve a polynomial-time algorithm for *uniform* semi-unification by exploiting a property that makes it possible to "abandon" most general (uniform) semi-unifiers and compute a more specific semi-unifier. This is possible because the more specific semi-unifier is guaranteed to exist if and only if the most general semi-unifier exists, which is the case if and only if there is any (uniform) semi-unifier at all. This property does not hold for two or more inequalities, and hence computing most general semi-unifiers seems the best approach for obtaining a correct decision algorithm for semi-unification. The functional problem of semi-unification — computing a most general semi-unifier — is of independent importance in its application in type inference. In ML, for example, a program that is submitted for type checking is annotated with type information, its principal type. We would also like to have complete type information for all the program fragments making up the whole program. This amounts to computing the most general semi-unifier of the SEI encoding the typing constraints of the program and printing it out as an annotation of the program.

We present three algorithmic specifications for computing the most general semi-unifier of an SEI in this chapter. The first one is a functional specification that is proved partially correct by fixed point induction. The second one is an SEI-rewriting specification whose partial correctness follows from a soundness and completeness theorem that shows that the class of solutions is invariant under rewritings. The third specification is a graph-theoretic version of the SEI-rewriting specification. Its encoding of SEI's by *arrow graphs*, which are term graphs with some additional structure, not only saves execution time and space over the SEI-rewriting formulation, it also seems more appropriate for analyzing its termination properties since all these specifications make use of a basically "nonlocal" failure criterion called the *extended occurs check*. These three specifications can be viewed as manifestations (or implementations) of one abstract algorithm. We conjecture that this algorithm is uniformly terminating, and that thus semi-unification and the Mycroft Calculus are decidable.

We have been implicitly suggesting that it is acceptable to talk about type inference and semi-unification interchangeably. This informality may be unwarranted if the reduction of type inference to semi-unification, as in chapter 4, needs to be done "off-line", as a proper preprocessing step to semi-unification, since this would be very undesirable in an interactive environment. Fortunately, it is quite easy to see that this reduction can be done "on-line", just as lexical, syntax, and semantic analysis in compilers can usually be "jammed" to a large degree. This enables compilers to operate in interactive and incrementable environments. Since a "direct" syntax-oriented type inference algorithm, based on algorithm A, is quite easy to obtain, yet raises a different set of issues that are more practical than those addressed in this thesis, we shall refrain from delving into details and only present algorithmic specifications for semi-unification.

6.1 Functional Specification

We now provide a functional specification of a most general semi-unifier of an SEI S, which we prove partially correct. W.l.o.g. we may assume that SEI's have at most one equation and at most one inequality per inequality group and that the SEI's are over alphabet $\mathcal{A}_2$. We start with some definitions and notational conventions used later.

Definition 12 *1. A* k-dimensional constraint mapping *R is a sequence $(R_1, \ldots, R_k)$ of finite maps from V to T that are undefined almost everywhere.*[1] *The* domain *$D(R_i)$ of R_i is the set of variables x for which $R_i(x)$ is defined.*[2] *A component of a constraint mapping can be applied to a term $\tau \in T$ by recursively applying it to the subterms of τ as long as it is defined for every variable occurring in τ; otherwise, the result is undefined.*

2. Let $R = (R_1, \ldots, R_k)$ and $R' = (R'_1, \ldots, R'_k)$ be constraint mappings; let $\tau_1, \tau_2 \in T$ be terms. A substitution U is an R-compatible semi-unifier in the i-th dimension (R-compatible unifier) of τ_1 and τ_2 via R' *if*[3]

(a) $R'_i(U(\tau_1)) = U(\tau_2)$(respectively, $U(\tau_1) = U(\tau_2)$)

(b) $(\forall j \in \{1, \ldots, k\})(\forall x \in D(R_j))R'_j(U(x)) = U(R_j(x))$

In the mutually recursive function specifications V (Figure 6.1) and U (Figure 6.2) we use quite standard notational conventions from both ALGOL-like and functional languages. Some notations are specific to our applications domain, though. $R_i\{x : \tau_2\}$ means the same thing for constraint mappings as it does for type environments: it denotes the constraint mapping identical to R with the only (possible) difference that $R_i\{x : \tau_2\}(x)$ is τ_2 no matter whether $R_i(x)$ is defined or undefined. The function "new" takes two arguments, a term τ_1 and a set of variables Φ. It returns a term τ'_1 that is obtained from τ_1 by replacing all variables in τ_1 with variables *not* in Φ; for convenience, it also returns the set of new variables thus introduced.[4] The operator $\circ$ denotes functional composition.

The function V takes five arguments: a set of variables Φ, an index i between 1 and k, a k-dimensional constraint mapping R, and two terms τ_1 and τ_2. U takes the same arguments except for the index. Both functions return a substitution and a (new) constraint mapping. In both cases the first argument, Φ, is only there for technical reasons to facilitate a "true" functional specification (and the correctness proof of the following lemma). For all practical purposes, a LISP-like "gensym" function used inside of the function "new" would be sufficient (and preferable). For simplicity both V and U are formulated for the functors f and c with arities 2 and 0, respectively.

Without going into too much detail we interpret the definitions of V and U as the least fixed points over suitable flat domains or, more prosaically, by any one of a number of computation rules (c. f. [71]).

[1] Recall that $T = T(\mathcal{A}_2, V)$.

[2] Note the difference between the domain of a substitution, which is defined everywhere, and a component of a constraint mapping, which is undefined almost everywhere.

[3] The order of τ_1 and τ_2 is significant for the definition of R-compatible semi-unifiers, but not for unifiers.

[4] Any function with these properties will do in place of "new". In fact, "new" encapsulates the nondeterminism of the problem (most general semi-unifiers are only unique up to weak equivalence) in this deterministic specification.

```
$V(\Phi, i, R, \tau_1, \tau_2) =$

**if** $\tau_1 = x$ (variable) **then**
    **if** $R_i(x)$ is undefined **then**
        $(\{\}, R_i\{x : \tau_2\})$
    **else**
        $U(\Phi, R, R_i(x), \tau_2)$
    **fi**
**elseif** $\tau_2 = y$ (variable) **then**
    **if** $(\exists v \in R^*(y))\tau_1$ contains $v$ **then**
        ERROR: occurs-check
    **else**
        **let** $(\tau_1', \Phi') = \text{new}(\Phi, \tau_1)$ **in**
        **let** $(U_1, R_1) = V(\Phi \cup \Phi', i, R, \tau_1, \tau_1')$ **in**
        **let** $(U_2, R_2) = U(U_1(\Phi), R_1, U_1(y), U_1(\tau_1'))$ **in**
            $(U_2 \circ U_1, R_2)$
    **fi**
**elseif** $\tau_1 = \tau_2 = c$ (constant) **then**
    $(\{\}, R)$
**elseif** $\tau_1 = f(v_1, \psi_1), \tau_2 = f(v_2, \psi_2)$ **then**
    **let** $(U_1, R_1) = V(\Phi, i, R, v_1, v_2)$ **in**
    **let** $(U_2, R_2) = V(U_1(\Phi), i, R_1, U_1(\psi_1), U_1(\psi_2))$ **in**
        $(U_2 \circ U_1, R_2)$
**else**
    ERROR: functor clash
**fi**


$R^*(y) =$

**the least** $X \mid \{y\} \subset X$ **and**
$(\forall x \in X)(\forall i \in \{1, 2, \ldots k\})R_i(x)$ is a variable
    $\Rightarrow R_i(x) \in X$
```

Figure 6.1: Functional specification of V

```
$U(\Phi, R, \tau_1, \tau_2) =$

**let** $(\tau_1, \tau_2) =$
    **if** $\tau_2 = y$ (variable) **then**
        $(\tau_2, \tau_1)$
    **else**
        $(\tau_1, \tau_2)$ **in**

**if** $\tau_1 = x$ (variable) **then**
    **if** $x = \tau_2$ **then**
        $(\{\}, R)$
    **elseif** $\tau_2$ contains $x$ **then**
        ERROR: occurs-check
    **else**
        $(U_t, R_t) := (\{x : \tau_2\}, \{x : \tau_2\}(R));$
        **for** $i = 1$ **to** $k$ **do**
            **if** $R_i(x)$ is defined **then**
                $(U_{t'}, R_{t'}) := V(U_t(\Phi), i, R_t, U_t(\tau_2), U_t(R_i(x)));$
                $(U_t, R_t) := (U_{t'} \circ U_t, R_{t'});$
            **fi**
        **rof**;
        $(U_t, R_t)$
**elseif** $\tau_1 = \tau_2 = c$ (constant) **then**
    $(\{\}, R)$
**elseif** $\tau_1 = f(\upsilon_1, \psi_1), \tau_2 = f(\upsilon_2, \psi_2)$ **then**
    **let** $(U_1, R_1) = U(\Phi, R, \upsilon_1, \upsilon_2)$ **in**
    **let** $(U_2, R_2) = U(U_1(\Phi), R_1, U_1(\psi_1), U_1(\psi_2))$ **in**
        $(U_2 \circ U_1, R_2)$
**else**
    ERROR: functor clash
**fi**
```

Figure 6.2: Almost-functional specification of U

```
$W(S) =$
            (Assume $S = (\tau_0 = \tau'_0, \tau_1 \leq \tau'_1, \ldots, \tau_k \leq \tau'_k)$)

$(U_t, R_t) := (\{\}, (\underbrace{\{\}, \ldots, \{\}}_{k\text{times}}));$
$(U_t, R_t) := U(\mathbf{vars}(S, R_t, \tau_0, \tau'_0);$
for $i = 1$ to $k$ do
      $(U_{t'}, R_{t'}) := V(U_t(\Phi), i, R_t, \tau_i, \tau'_i);$
      $(U_t, R_t) := (U_{t'} \circ U_t, R_{t'});$
rof;
return $U_t$;
```

Figure 6.3: Pseudo-functional specification of most general semi-unifier

Lemma 32 *Let $\tau_1, \tau_2 \in T$ be terms. Let Φ be a recursive subset of V containing all variables occurring in τ_1 and τ_2 such that $V - \Phi$ is infinite. Let R be a constraint mapping.*

1. *If $V(\Phi, i, R, \tau_1, \tau_2)$ terminates with an error, then there is no R-compatible semi-unifier in the i-th dimension of τ_1 and τ_2. If $(U', R') = V(\Phi, i, R, \tau_1, \tau_2)$ terminates without error, then U' is a Φ-maximal R-compatible semi-unifier in the i-th dimension (via R') of τ_1 and τ_2; that is,*
   (a) $R'_i(U'(\tau_1)) = U'(\tau_2)$
   (b) $(\forall i \in \{1, \ldots, k\})(\forall x \in D(R_i))R'_i(U(x)) = U(R_i(x))$
   (c) *For any R-compatible semi-unifier T in the i-th dimension of τ_1 and τ_2 there is a substitution S such that* $(\forall x \in \Phi)S(U'(x)) = T(x)$

2. *If $U(\Phi, R, \tau_1, \tau_2)$ terminates with an error, then there is no R-compatible unifier of τ_1 and τ_2. If $(U', R') = U(\Phi, R, \tau_1, \tau_2)$ terminates without error, then U' is a Φ-maximal R-compatible unifier (via R') of τ_1 and τ_2; that is,*
   (a) $U'(\tau_1) = U'(\tau_2)$
   (b) $(\forall i \in \{1, \ldots, k\})(\forall x \in D(R_i))R'_i(U(x)) = U(R_i(x))$
   (c) *For any R-compatible unifier T of τ_1 and τ_2 there is a substitution S such that* $(\forall x \in \Phi)S(U'(x)) = T(x)$

The proof of this lemma is by simultaneous computational induction over the definitions of V and U. Its details are truly tedious, but they are available as a manuscript [35]. The constraint mapping R passed as an argument to V and U encodes the inequational constraints encountered during the course of the computation. Any further substitution has to be compatible with these constraints in the sense that they must preserve these inequational constraints. This "preservation" of constraints is captured in the notion of R-compatible semi-unifiers and unifiers. The lemma states that V and U return the most general semi-unifiers and unifiers, respectively, that are compatible with the input constraints R. From this lemma we obtain immediately a routine W (see figure 6.3) that computes a most general semi-unifier for every "normal form" SEI S with at most one equation and one inequality per inequality group.

Theorem 19 *Let S be a system of equations and inequalities consisting of singleton sets only.*

If $W(S)$ does not terminate or terminates with an error then S has no solution. If $U' = W(S)$ terminates without error then U' is a most general semi-unifier of S.

This specification has already appeared in [38]. We have also implemented it in SETL [108] and tested it on several examples.

6.2 SEI-Rewriting Specifications

In this section we present basic, implementable rewriting specifications for most general semi-unifiers. The first is a natural and straightforward extension of the rewriting specification for most general unifiers from [39], which was expounded by Martelli and Montanari and used as the starting point for the development of efficient unification algorithms [72]. This system is in general, though, nonterminating. The second rewriting specification refines the first one by adding an "extended" occurs check. It is conjectured to be uniformly terminating.

6.2.1 The Naive Rewriting Specification

The first specification, given in Figure 6.4 is straightforward, and similar versions can be found in the literature (e.g., [15]). This rewriting system preserves semi-unifiers in a sense that we shall make precise below.

Definition 13 *Let $\Rightarrow$ be a reduction relation on systems of equations and inequalities.*

1. *The relation $\Rightarrow$ is* sound *if for every S, S' such that $S \Rightarrow S'$ and for every semi-unifier σ' of S' there is a semi-unifier σ of S such that $\sigma |_{V(S)} = \sigma' |_{V(S)}$ (and thus $\sigma \cong_{V(S)} \sigma'$).*
2. *The relation $\Rightarrow$ is* complete *if for every S, S' such that $S \Rightarrow S'$ and for every semi-unifier σ of S there is a semi-unifier σ' of S' such that $\sigma |_{V(S)} = \sigma' |_{V(S)}$ (and thus $\sigma \cong_{V(S)} \sigma'$).*

Informally speaking, soundness expresses that a reduction step does not add semi-unifiers, and completeness means that no semi-unifiers are lost in a reduction step.

Proposition 33 *The reduction relation defined by the naive rewriting system (in Figure 6.4) is sound and complete.*

Proof: Induction on the number of rewriting steps.

Any SEI S is in *normal form* with respect to a reduction relation $\Rightarrow$ if there is no S' such that $S \Rightarrow S'$. If an SEI is in normal form with respect to the naive rewriting system or the canonical rewriting system below it is easy to extract a most general semi-unifier from it.

Proposition 34 *Let S be a system of equations and inequalities in normal form with respect to the reduction relation defined by the naive (canonical) rewriting system in Figure 6.4.*

If $S = \{x_1 = M_1, \ldots, x_k = M_k, y_1 \leq N_1, \ldots, y_l \leq N_l\}$ then the substitution $\sigma = \{x_1 \mapsto M_1, \ldots, x_k \mapsto M_k\}$ is a most general idempotent semi-unifier of S.

Proof: By inspection.

To determine a most general semi-unifier of an SEI S we can apply the naive rewriting system to it and if it terminates in a normal form S' we can extract a most general semi-unifier of S'. If $S' = \square$ then S is unsolvable; otherwise there is a most general semi-unifier σ' of S' according to proposition 34. As a result of proposition 33 the restriction $\sigma' |_{V(S)}$ (or σ' itself) is a most general semi-unifier of S.

Given an SEI S with k sets of inequalities we initially tag all the inequality symbols with distinct "colors" $1, \ldots, k$ to indicate to which group of inequalities they belong. This is done by superscripts of the inequality symbol; e.g., $\leq^{(1)}$ Then nondeterministically choose an equation or inequality and take a rewriting action depending on its form.[a]

1. $f(M_1, \ldots, M_k) = f(N_1, \ldots, N_k)$:

   Replace by the equations $M_1 = N_1, \ldots, M_k = M_k$.

2. $f(M_1, \ldots, M_k) = g(N_1, \ldots, N_l)$ where f and g are distinct functors:

   Replace current SEI by $\Box$ (functor clash).

3. $f(M_1, \ldots, M_k) = x$:

   Replace by $x = f(M_1, \ldots, M_m)$.

4. $x = f(M_1, \ldots, M_k)$ where x occurs in at least one of $M_1, \ldots, M_k$:

   Replace current SEI by $\Box$ (occurs check).

5. $x = M$ where x does not occur in M, but occurs in another equation or inequality:

   Replace x by M in all other equations or inequalities.

6. $x = x$:

   Delete it.

7. $f(M_1, \ldots, M_k) \leq^{(i)} f(N_1, \ldots, N_k)$:

   Replace by inequalities $M_1 \leq^{(i)} N_1, \ldots, M_k \leq^{(i)} M_k$.

8. $x \leq^{(i)} M$ and $x \leq^{(i)} N$:

   Delete one of the two inequalities and add the equation $M = N$.

9. $f(M_1, \ldots, M_k) \leq^{(i)} x$:

   Add the equation $x = f(x'_1, \ldots, x'_k)$ where $x'_1, \ldots, x'_k$ are new variables not occurring anywhere else.

[a] Without loss of generality we restrict ourselves to the minimal nonlinear alphabet $\mathcal{A} = (f, \{f \mapsto 2\})$. Recall that $\Box$ denotes the canonical unsolvable SEI.

Figure 6.4: Naive rewriting specification

(9.1) $f(M_1, \ldots, M_k) \leq^{(i_0)} x$ and there are variables $x_0, \ldots, x_n$ such that $x = x_0$, $x_i \leq^{(j_i)} x_{i+1}$ are inequalities in the current SEI for $0 \leq i \leq n-1$ and some colors $i_1, \ldots, i_{n-1}$, and there exists an i such that x_n occurs in M_i:

Replace current SEI by $\Box$ (extended occurs check).

(9.2) $f(M_1, \ldots, M_k) \leq^{(i_0)} x$ and there is no sequence of variables $x_0, \ldots, x_n$ such that $x = x_0$, $x_i \leq^{(j_i)} x_{i+1}$ are inequalities in the current SEI for $0 \leq i \leq n-1$ and some colors $j_1, \ldots, j_{n-1}$, and x_n occurs in some M_i:

Add the equation $x = f(x'_1, \ldots, x'_k)$ where $x'_1, \ldots, x'_k$ are new variables not occurring anywhere else.

Figure 6.5: Extended occurs check

6.2.2 The Canonical Rewriting Specification

There are systems of equations and inequalities for which there is no finite rewriting derivation in the naive rewriting system; that is, no sequence of rewriting steps such that after a finite number of steps no more rewritings are possible. Consider, for example, the system $S_0 = \{f(x, g(y)) \leq f(y, x)\}$. It is easy to see that there is always at least one rule applicable.

The main reason for nontermination is that the last inequality rule, rule (9), introduces new variables every time it is executed. Replacing it with the deceivingly pleasing rule [97]

$f(M_1, \ldots, M_k) \leq x$:

Add the equation $x = f(M_1, \ldots, M_k)$.

indeed eliminates the nontermination problem of rewriting derivations, but also its completeness. To see this, consider, for example, the system $S_1 = (f(g(y), g(y)) \leq f(x, g(g(y))))$. There is a derivation that would lead us to claim, incorrectly, that S_1 has no semi-unifiers.

If we reconsider system S_0 it is easy to see that it is unsolvable. This is due to the fact that the inequalities

$$\begin{aligned} g(y) &\leq x \\ x &\leq y \end{aligned}$$

are not uniformly or nonuniformly solvable. If we denote the length of a term M by $|M|$, then any solution M_1 for x and M_2 for y would have to satisfy the numeric inequalities $|M_1| \leq |M_2|$ and $|M_1| \geq |g(M_2)| \geq |M_2| + 1$, which is clearly impossible. We can catch this case by refining rule (9) with an "extended" occurs check. More precisely, let us call the rewriting system with rule (9) replaced by the rules in Figure 6.5 the *canonical* rewriting system.

Proposition 35 *The reduction relation defined by the rewriting system in Figure 6.4 with rule (9) replaced by the rules (9.1) and (9.2) from Figure 6.5 is sound and complete.*

Proof: See discussion of system S_0.

For any reduction relation $\Rightarrow$ with a notion of normal form, an *effective (one-step) normalizing strategy* is a polynomial-time computable function F such that if $F(S) = S$ then S is a normal form and, otherwise, if $F(S) = S'$ then $S \Rightarrow S'$; and furthermore, if $S \overset{*}{\Rightarrow} S'$, S' a normal form, then $F^n(S) = F^{n+1}(S)$ for some $n \in \mathcal{N}$.

Even though there are still infinite rewriting derivations possible in the canonical rewriting system we conjecture that there is an effective normalizing strategy for the canonical rewriting system.

Conjecture 1 *There exists an effective normalizing strategy for the canonical rewriting system such that the strategy admits only finite rewriting derivations.*

In fact we believe that any strategy that executes rule (9.2) only if there are no other rules applicable satisfies this conjecture (see chapter 7).

An immediate consequence of this conjecture is the decidability of semi-unification.

Conjecture 2 *The set of all solvable systems of equations and inequalities is decidable.*

6.3 Graph Rewriting Specification

It is probably easier to analyze the extended occurs check in both the functional specification and the SEI-rewriting specification in a graph-theoretic setting since it is a (syntactically) nonlocal condition. This formulation is a good starting point for both the analysis of termination properties, for practical implementations, and for optimizations for subcases of general semi-unification, such as uniform semi-unification.

6.3.1 Arrow graphs

Recall that term graphs are (nonunique) representations for sets of terms. Arrow graphs are term graphs with additional structure to represent SEI's.

Definition 14 *A* (k-colored) arrow graph *G is a sextuple $(N, N_F, E, L, A, \sim)$ where $|G| = (N, N_F, E, L)$ is a term graph (over $\mathcal{A}_2$), $A = (A_1, \ldots, A_k)$ is a k-tuple, $A_i \subset N \times N$, for $1 \leq i \leq k$; the elements of A_i are called* arrows*; and $\sim$ is an equivalence relation on N.*

We can think of an arrow in A as colored by $1, \ldots, k$ indicating to which A_i it belongs. We may write $m \xrightarrow{i} n$ for $(m, n) \in A_i$ whenever A and A_i are understood from the context.

An *arrow graph representation* of SEI $S = (M_0 = N_0, M_1 \leq N_1, \ldots, M_k \ldots N_k)$ is a an arrow graph G whose underlying term graph, $|G|$, represents all the terms occurring in S; G contains arrows $m_i \xrightarrow{i} n_i$ if $[m_i] = M_i, [n_i] = N_i$ for $1 \leq i \leq k$; and $\sim$ in G is the smallest equivalence relation containing $m_0 \sim n_0$ if $[m_0] = M_0, [n_0] = N_0$. In other words, the colored arrows encode inequalities, and the equivalence relation encodes equations.

Let $\mathcal{A} = \mathcal{A}_2$ be the usual ranked alphabet.

Definition 15 *An* interpretation *I of an arrow graph $G = (N, N_F, L, E, A, \sim)$ is a mapping of nodes to first-order terms (with variables) over the ranked alphabet $\mathcal{A}$. I is* valid *if there exist* quotient substitutions *$R_1, \ldots, R_k$ such that*

1. $(\forall n \in N_F, n_1, n_2 \in N)\ L(n) = f, E(n) = (n_1, n_2) \Rightarrow I(n) = f(I(n_1), I(n_2))$;
2. $(\forall n, n' \in N)\ n \sim n' \Rightarrow I(n) = I(n')$;
3. $(\forall n, n' \in N, 1 \leq i \leq k)\ n \xrightarrow{i} n' \Rightarrow R_i(I(n)) = I(n')$.

It is easy to see that an SEI S has a semi-unifier if and only if G, an arrow graph representation of S, has a valid interpretation.

Let $G = (N, N_F, E, L, A, \sim)$. Apply the following rules (depicted also in Figure 6.7) until convergence:

1. If there exist nodes m and n labeled with a functor f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \sim n$ then merge the equivalence classes of m_1 and n_1 and of m_2 and n_2.
2. If there exist nodes m and n labeled with a functor f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \xrightarrow{i} n$ then place arrows $m_1 \xrightarrow{i} n_1$ and $m_2 \xrightarrow{i} n_2$.
3. If there exist nodes m_1, m_2, n_1, and n_2 such that
   (a) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $n_1 \xrightarrow{i} n_2$ then merge the equivalence classes of m_2 and n_2;
   (b) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $m_2 \sim n_2$ then place an arrow $n_1 \xrightarrow{i} n_2$.
4. (a) (Extended occurs check) If there is an path consisting of arrows of any color (arrow path) from n_1 to n_2 and n_2 is a proper descendant of n_1, then reduce to the improper arrow graph $\Box$.
   (b) If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with functor f and has children m_1, m_2, n is not equivalent to a functor labeled node, and there is an arrow $m \xrightarrow{i} n$ then create new nodes n', n'_1, n'_2 (each initially in their own equivalence class) and label n' with functor f, label n'_1 and n'_2 with new variables x' and x'', respectively; make n'_1, n'_2 the children of n'; and merge the equivalence classes of n and n'.

Figure 6.6: Algorithm A

6.3.2 Algorithm A

Algorithm A in Figure 6.6 applies the closure rules depicted in Figure 6.7, repeatedly rewriting the initial arrow graph representation of an SEI S until the arrow graph does not change any more.

An equivalence relation on the nodes of a term graph can be interpreted as a substitution relative to a system of (equivalence class) representatives as long as the equivalence relation is a structural equivalence, i.e., satisfies closure rule 1 and does not trigger rule 4a. This correspondence has been widely used in graph-theoretic formulations of unification algorithms (c. f. [89]), and we will refrain from making it precise here.

Proposition 36 *Let G be an arrow graph representation of SEI S. If algorithm A (Figure 6.6) terminates on input G with an arrow graph $G' \neq \Box$, then the resulting equivalence relation represents a most general semi-unifier of S. If $G' = \Box$ then S is unsolvable.*

Proof:

Every graph rewriting step corresponds to SEI rewriting steps in the canonical rewriting system in Figure 6.4 with the extended occurs check rules from Figure 6.5 replacing rule (9) on the terms represented by the term graph $|G|$, and vice versa. By proposition 35 the canonical SEI-rewriting system computes a most general semi-unifier.

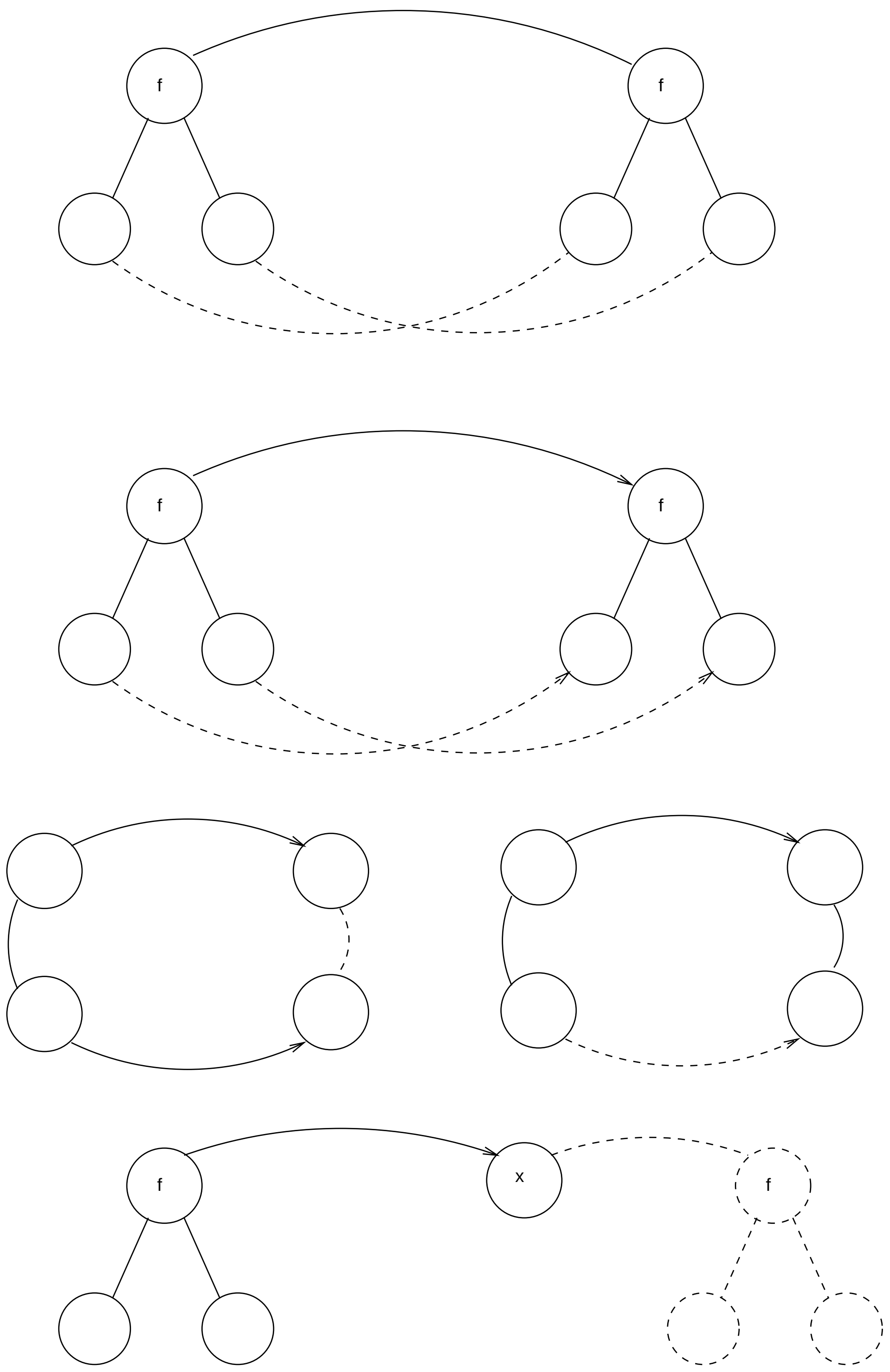

Figure 6.7: Closure rules

In contrast to Mycroft's original type inference algorithm for the Mycroft Calculus there are no known inputs that lead to nontermination of algorithm A. Nonetheless it is currently unknown whether algorithm A terminates uniformly or whether there is any uniformly terminating algorithm for semi-unifiability at all. Since we conjecture that both questions have an affirmative answer, we believe that key to the establishment of this result is an in-depth investigation of the deep structure of sequences of arrow graphs that arise from the (nondeterministic) execution of algorithm A. For this reason we call a sequence $\mathcal{G} = (G_1, \ldots, G_i, \ldots,)$ of arrow graphs an *execution (of algorithm A)* if every component in the sequence is derived from its predecessor by one of the rewriting rules in Figure 6.6, respectively Figure 6.7. Some elementary approaches and preliminary results are reported in chapter 7.

6.4 Arithmetic Compression for Uniform Semi-Unification

For uniform semi-unification we will show that it is possible to establish decidability. In fact algorithm A (Figure 6.6) terminates uniformly for every input in exponential time and space if the initial arrow graph is only 1-colored (see below). By a form of "arithmetic" compression it is possible to compute most general uniform semi-unifiers in polynomial space, as shown in this chapter. The decidability of uniform semi-unification has also been discovered by Pudlák [96]. If it is only desired to decide uniform semi-unifiability it is possible to simplify the algorithm and run it in polynomial time by a result of Kapur *et al.* [54].

6.4.1 An exponential time algorithm for uniform semi-unification

It can be shown that algorithm A terminates in exponential time for *uniform* semi-unification under a deterministic rewriting strategy we shall describe below. It is inspired by *normalized* executions, in which rule 4 of algorithm A is only executed when none of the other rules are applicable. What permits a relatively simple termination proof (and the exponential upper bound) is that, for arrow graphs of one color, for every node without an outarrow there can be at most one "new" node created by execution of rule 4. This property does not hold for arrow graphs with two or more colors.

Proposition 37 *The algorithm "solve" in Figure 6.8 is an exponential-time uniform semi-unification algorithm.*

Proof:

Let us say a *discrepancy* in an arrow graph is a node where rule 4 can be applied; i.e., it is a functor node n with an outarrow to variable node n' that is *not* equivalent to any functor node. We associate with every arrow graph G of one color the triple $(e_{w/o}, w, e)$, called the *characteristic* of G, where

1. $e_{w/o}$ is the number of equivalence classes in G that has *no* node with an outarrow (i.e., for no node n in the equivalence class is there an arrow $n \xrightarrow{1} n'$ for any n');
2. w is the number of equivalence classes that contain only variable nodes at least one of which is reachable from a discrepancy via an arrow path (discrepancy weight);
3. e is the number of equivalence classes.

These triples are lexicographically well-ordered.

The procedure "solve" in Figure 6.8 implements a specific strategy for applying the closure rules of algorithm A. In particular, rules 1 and 3, which merge equivalence classes, are always applied exhaustively after any of the other steps as a "normalization" step. Furthermore, when rule 4 is applicable at some discrepancy n then it is clear that it can be applied recursively at every descendant of n after execution of rule 2 at node n, until the variable leaves of n are reached; this is accomplished by the procedure "copy". Since every new node created by copy(n) is *not* a descendant of n, it is easy to see that an invocation of copy(n) creates k new nodes, if n has k descendants (descendant equivalence classes) with no outarrow at the time copy(n) is called.

Let us call exhaustive application of rules 1 and 3 a *normalization* step. We call merging an equivalence class *with* an outarrow and an equivalence class *without* an outarrow a *skewed merge.*

Now note that the exhaustive application of rules 1 and 3, if applicable at least once, decreases the number of equivalence classes at least by one. Furthermore, if normalization contains a skewed merge, then the discrepancy weight, w, may be increased, but the number of equivalence classes *without* an outarrow, $e_{w/o}$, is decreased by at least one. If normalization contains *no* skewed merge, the number of equivalence classes reachable from any discrepancy does not increase and, consequently, the discrepancy weight does not increase, either. In all cases the total number of equivalence classes does not increase.

Application of rule 2 with subsequent normalization leads to consideration of two cases: either the tail of the new arrow propagated has already an outarrow, or it does not. In the first case, clearly $e_{w/o}$ is decreased by one. In the second case e is properly decreased, and there are two possibilities to consider depending on whether the normalization contains a skewed merge. If a skewed merge occurs, $e_{w/o}$ is properly decreased. If no skewed merge occurs, it can be seen that the discrepancy weight is not increased.

Finally, a discrepancy n is *minimal* if there is no discrepancy n', a sequence of nodes $n_1, \ldots, n_k$ such that $n' = n_1, n = n_k$ and $n_i \xrightarrow{1} n_{i+1}$ or n_i is a child of n_{i+1} for $1 \leq i \leq k-1$ with the additional constraint that there is a j such that n_j is a child of n_{j+1} (see chapter 7). If there is a discrepancy, but no minimal one, then reduction to $\Box$ is performed, since this corresponds to a "preemptive" application of the extended occurs check. If a minimal discrepancy exists, then rule 4b is applicable at a minimal discrepancy (there may be several minimal discrepancies). Instead of applying rule 4b only once algorithm "solve" applies routine "copy", which is an exhaustive application of rule 4b to the original discrepancy and, recursively, all its descendants, facilitated by intermediate applications of rule 2 and 3. Application of "copy" to the children of a minimal discrepancy terminates in time $O(e_{w/o})$ and decreases the discrepancy weight by one. Although e is properly increased, $e_{w/o}$ is not.[5]

This shows that every iteration through the loop strictly decreases the characteristic of the rewritten arrow graph. Consequently the procedure solve terminates uniformly. Furthermore, since w is bounded by e, and e is only increased by execution of "copy", it can be seen that e at most doubles every time $e_{w/o}$ decreases by one. Clearly every iteration of the loop in "solve" is executed in polynomial time with respect to the size (number of nodes) of the arrow graph before the iteration. This shows that solve(G) terminates in exponential time; i.e., in time $O(2^{cn^k})$, for some c, k, where n is the number of nodes in G.

[5]The fact that $e_{w/o}$ is not increased by application of "copy" is at heart of why this termination proof works for *uniform semi-unification*, but not for general semi-unification.

Recall the closure rules of algorithm A, Figure 6.6, also depicted in Figure 6.7.

```
solve($G$) =
    repeat
        apply rules 1 and 3 exhaustively;
        if rule 2 is applicable then
            apply it (once);
        else if rule 4 is applicable then
            if there is a minimal discrepancy $n$ then
                $(L(n) = f, E(n) = (n_1, n_2))$
                create a new functor node $n'$, $L(n') = f$,
                with children copy($n_1$) and copy($n_2$);
                place an arrow from $n$ to $n'$;
            else
                reduce to $\square$ (extended occurs check);
            end if
        end if
    until no more rules are applicable;


copy($n$) =
    if $n$ has an outarrow to some node $n'$ then
        return $n'$;
    else if $n$ is equivalent to a functor node $n'$,
            $L(n') = f, E(n') = (n'_1, n'_2)$ then
        create new functor node $n''$, $L(n'') = f$,
        with children copy($n'_1$) and copy($n'_2$);
        return $n''$;
    else
        create new variable node $n'$, $L(n') = x'$, where $x'$ is a new variable;
        return $n'$;
    end if
```

Figure 6.8: Exponential-time uniform semi-unification algorithm

Let $G = (N, N_F, L, E, C)$ be an interaction graph. G is normal if it satisfies the following closure rules.

1. For $n, n_1, n_2, n', n'_1, n'_2 \in N$ such that $E(n) = (n_1, n_2), E(m) = (m_1, m_2)$, if $(l, l') \in C(n, n')$ then $(l, l') \in C(n_1, n'_1)$ and $(l, l') \in C(n_2, n'_2)$.
2. For $n_1, n_2, n_3, n_4 \in N$, if $(l_{01}, l_{10}) \in C(n_0, n_1), (l_{23}, l_{32}) \in C(n_2, n_3), (l_{02}, l_{20}) \in C(n_0, n_2)$ then $(l_{10} + (l_{02} - l_{01}), l_{32} + (l_{20} - l_{23})) \in C(n_1, n_3)$ if the differences above are nonnegative.
3. If $(l, l') \in C(n, n')$ then $(l', l) \in C(n', n)$.
4. $(0, 0) \in C(n, n)$.
5. If $(l, l') \in C(n, n')$ then $(l + 1, l' + 1) \in C(n, n')$.

Figure 6.9: Consistency rules for uniform semi-unification

6.4.2 Interaction Graphs

Notice that a 1-colored arrow graph G will be transformed by algorithm A into a possibly exponentially bigger arrow graph with many "new" nodes (introduced by rule 4 in Figure 6.6). Let us call the nodes that are in the input graph G "original" nodes and all other nodes that are added by A "new" nodes. If we consider all arrow paths in the "evolved" graph after a number of graph rewriting steps it can be seen from the closure rules that almost all the relevant information about arrow paths in an execution of algorithm A can be computed from other information about arrow paths. For the most part, it is sufficient to consider only arrow paths from original nodes to original nodes. Since there may be arrow graphs from two original nodes to a common new node we have to consider, more generally, all the possible ways in which arrow graphs from two original nodes "merge" together, if at all. Consequently it is not necessary to explicitly construct new nodes, only *all relevant information* about arrow paths from pairs of original nodes. Since we assume only 1-colored graphs the arrow paths in question are completely characterizable by their starting point, end point and their *length*. Since the length can be stored in space $O(\log n)$ where n is the length itself, this representation of arrow paths yields a space compression due to this "arithmetization" of arrow paths. Indeed we can thus devise an algorithm that computes a most general uniform semi-unifier in polynomial space. The details are below.

Definition 16 *(Interaction graph)*
An interaction graph (of degree 1) *is a term graph over $\mathcal{A}_2$ with an additional* consistency *mapping $C : N \times N \rightarrow 2^{\mathcal{N} \times \mathcal{N}}$. A* normal *interaction graph is an interaction graph whose consistency sets satisfy the rules in Figure 6.9.*

An *interaction graph representation* of an SEI S is very similar to an arrow graph representation (for uniform semi-unification problems). In particular, both inequalities and equations can be encoded in a single consistency mapping. Specifically, an interaction graph representation of $S = (M_0 = N_0, M_1 \leq N_1)$ is an interaction graph G whose underlying term graph, $|G|$, represents all the terms occurring in S; and the consistency mapping in G is the smallest C such that $(0, 0) \in C(m_0, n_0), (1, 0) \in C(m_1, n_1)$ if $[m_i] = M_i, [n_i] = N_i, 0 \leq i \leq 1$.
Let $\mathcal{A} = \mathcal{A}_2$ be the usual ranked alphabet.

Definition 17 *An* interpretation *I of an interaction graph $G = (N, N_F, L, E, C)$ is a mapping of nodes to first-order terms (with variables) over the ranked alphabet $\mathcal{A}$. I is* valid *if there is a quotient substitution*

R such that

1. $(\forall n \in N_F, n_1, n_2 \in N)\ L(n) = f, E(n) = (n_1, n_2) \Rightarrow I(n) = f(I(n_1), I(n_2))$;
2. $(\forall n, n' \in N, l, l' \in \mathcal{N})\ (l, l') \in C(n, n') \Rightarrow R^l(I(n)) = R^{l'}(I(n'))$.

It is clear that for every interaction graph G with consistency mapping C there is a unique smallest normal interaction graph $\bar{G}$ with the same term graph as G and a consistency mapping $\bar{C}$ that contains C. It is also easy to check that I is a valid interpretation for G if and only if I is a valid interpretation for $\bar{G}$, and SEI S has a uniform semi-unifier if and only if an interaction graph representation G of S has a valid interpretation.

Now, a somewhat more complicated analog of the extended occurs check of algorithm A, applied to a normal interaction graph $\bar{G}$, determines whether there is a valid interpretation for $\bar{G}$ and, consequently, whether the SEI that $\bar{G}$ represents has a uniform semi-unifier.

For $n' \in N_F, n, n'_1, n'_2 \in N, E(n') = (n'_1, n'_2)$ we say (n'_1, l') (respectively (n'_2, l') is a *direct left (right) descendant* of (n, l) with respect to C if $(l, l') \in C(n, n')$. The transitive closure of this relation defines *proper descendancy*, and the reflexive-transitive closure defines *descendancy.*

Theorem 20 *Let G be an interaction graph representing an SEI S over $\mathcal{A}_2$, and let $\bar{G}$ be the smallest normal interaction graph containing G, where the consistency mapping in $\bar{G}$ is $\bar{C}$. Then S is uniformly semi-unifiable if and only if for no $n \in N$ and $l, d \in \mathcal{N}$, $(n, l + d)$ is a proper descendant of (n, l) in $\bar{C}$.*

Proof:

First we shall prove that if there is $n \in N$ and $l \in \mathcal{N}$ such that (n, l) is a proper descendant of $(n, 0)$ in $\bar{C}$, then $\bar{G}$ has no valid interpretation. Assume I is a valid interpretation with quotient substitution R. If (n', l') is a direct descendant of (n, l) in $\bar{C}$, then $|R^{l'}(I(n'))| < |R^l(I(n))|$, and, by induction, this holds also if (n', l') is a proper descendant of (n, l) in $\bar{C}$. If $(n, l+d)$ is a proper descendant of (n, l) this means that $|R^d(R^l(I(n)))| < |R^l(I(n))|$; but this is manifestly impossible since applying a substitution to a term cannot make the (tree) size of a term smaller. Consequently there cannot be a valid interpretation for $\bar{G}$.

Conversely, if there is no $(n, l + d)$ that is a proper descendant of (n, l), then we can define a valid interpretation for $\bar{G}$. Assume all variables in V are totally ordered in some fashion. We may assume that the underlying term graph of $\bar{G}$ has exactly one node labeled x for every variable x occurring in S. Thus the ordering on variables extends uniquely to nodes. We can also extend it lexicographically to node-number pairs, where the ordering on numbers is the standard arithmetic ordering.

Consider the function I defined in Figure 6.10. It is routine to check that $I(n) = I(n, 0)$ defines a well-defined valid interpretation since the closure properties of the consistency rules guarantee that the definition is well-defined, and the recursion must terminate if there is no (n, l) that has $(n, l + d), d \geq 0$, as a proper descendant.

6.4.3 A polynomial space algorithm for uniform semi-unification

The consistency mapping in an interaction graph maps pairs of nodes to infinite sets. In order to transform the closure rules for interaction graphs into an algorithm it is necessary to find a finite representation and effective means of manipulating it. We can consider a given set $C(n, n')$ and "close" it with respect to the consistency rules in the "trivial" term graph consisting only of nodes n and n' (and no edges or other nodes). In this sense every set of pairs of nonnegative numbers generates, independent of any term graph, a unique smallest set of pairs of numbers that are closed with respect to the consistency

```
I(n, l) =
    if $(n, l)$ has no proper descendant then
        ($n$ is variable labeled)
        let $(n', l')$ be the least $(n'', l'')$ such that
            $(n'', l'') \leq (n, l)$;
        return $y^{(l'')}$ (where $L(n) = y$)
    else
        let $(m', l'), (m'', l'')$ be direct left,
            respectively right, descendants of $(n, l)$;
        return $f(I(m', l'), I(m'', l''))$
    end if
```

Figure 6.10: Interaction graph interpretation

Let $C \subset \mathcal{N} \times \mathcal{N}$. C is *consistently closed* if the following closure rules are satisfied.

1. If $(l_1, l_2), (l_3, l_2), (l_3, l_4) \in C$ then $(l_1, l_4) \in C$.
2. If $(l, l') \in C$ then $(l + d, l' + d) \in C, d \geq 0$,.

Figure 6.11: Consistently closed relations

rules. We shall show that every such closed set can be represented by at most two pairs of numbers, and the consistency rules that involve the structure of a given term graph, namely rule 1 and rule 2 can be encoded by effective operations on such pairs of numbers. The details are below.

A binary relation C on the natural numbers is *consistently closed* if the closure rules in Figure 6.11 are satisfied.

For consistently closed relations we have the following proposition.

Proposition 38 *Let C be a consistently closed relation, and let $l_1, l_2, d, d' \in \mathcal{N}, d' \geq d > 0$. Then*

1. $(l_1, l_2), (l_1 + d, l_2) \in C \Rightarrow (l_1, l_2 + d) \in C$ *and* $(l_1, l_2), (l_1, l_2 + d) \in C \Rightarrow (l_1 + d, l_2) \in C$;
2. $(l_1, l_2), (l_1, l_2 + d), (l_1, l_2 + d') \in C \Rightarrow (l_1, l_2 + (d' - d)) \in C$;
3. $(l_1, l_2), (l_1, l_2 + d), (l_1, l_2 + d') \in C \Rightarrow (l_1, l_2 + \gcd(d, d')) \in C$.

Proof:

1. If $(l_1, l_2), (l_1 + d, l_2) \in C$, then $(l_1 + d, l_2 + d) \in C$ by rule 2 of the definition of consistently closed relations (Figure 6.11), and, by rule 1, $(l_1, l_2 + d)$. The other case is symmetric.
2. If $(l_1, l_2), (l_1, l_2 + d), (l_1, l_2 + d') \in C$, then $(l_1 + (d' - d), l_2 + d + (d' - d)) = (l_1 + (d' - d), l_2 + d') \in C$ by rule 2. Since $(l_1 + (d' - d), l_2 + d'), (l_1, l_2 + d'), (l_1, l_2) \in C$ we have $(l_1 + (d' - d), l_2) \in C$ by rule 1. The result follows by case 1 above.

3. Note that, by induction on Euler's gcd-algorithm, if for any property $P(d)$ over the natural numbers we have $(\forall d, d' \in \mathcal{N}, d' \geq d)\ P(d)$ and $P(d') \Rightarrow P(d'-d)$ then it also holds that $P(d)$and $P(d') \Rightarrow P(\gcd(d, d'))$. If we let $P(d) \equiv (l_1, l_2 + d) \in C$, then the result follows from case 2.

The significance of consistently closed relations is summarized in the following proposition.

Proposition 39 *Let $\bar{G}$ be a normal interaction graph with consistency mapping $\bar{C}$. Then $\bar{C}(n, n')$ is consistently closed for all nodes n, n'.*

Proof:

Closure property 2 is established by simple induction on d and rule 5 in the consistency rules for normal interaction graphs (Figure 6.9).

Property 1 is a special case of rule 2 in Figure 6.9. Consider $\bar{C}(n, n')$. If $(l_1, l_2), (l_3, l_2), (l_3, l_4) \in \bar{C}(n, n')$, then $(l_2, l_3) \in \bar{C}(n', n)$ by rule 3 of Figure 6.9. With $n_0 = n', n_1 = n, n_2 = n, n_3 = n', l_{10} = l_1, l_{01} = l_{02} = l_2, l_{20} = l_{23} = l_3, l_{32} = l_4$ it follows by rule 2 that $(l_{10}, l_{32}) = (l_1, l_4) \in \bar{C}(n, n')$.

Drawing on terminology from algebra, we shall say a relation B *spans* a consistently closed C if the smallest consistently closed relation containing B is C; we shall denote this by $\langle B \rangle = C$. If no set with cardinality smaller than B spans C, then B is a *basis* of C. A set B is *independent* if no proper subset of B is a basis.

The following theorem is at the heart of our uniform semi-unification algorithm.

Theorem 21 *1. Every consistently closed relation C has a basis of cardinality at most 2; i.e., there exist $l_1, l_2, l'_1, l'_2 \in \mathcal{N}$ such that $\langle (l_1, l_2), (l'_1, l'_2) \rangle = C$.*[6]

2. For every consistently closed relation C there exist unique $l, l', c \in \mathcal{N}, k \geq -c$ such that $C = \langle (l, l'), (l + k, l' + k + c) \rangle$.

Proof:

Part 1 follows immediately from part 2, of course.

Let C be consistently closed relation. If C is empty, then the empty set is a basis of C, and we are done. Otherwise, let l' be the smallest number with $(l'', l') \in C$ for some l'', and let l be the smallest l such that $(l, l') \in C$. If $\langle (l, l') \rangle = C$, we are done. Otherwise, let c be the smallest positive number such that $(l + k'', l' + k'' + c) \in C$ for some (possibly negative) integer k''. Let k be the smallest k such that $(l + k, l' + k + c) \in C$. Note that $k > -c$ by definition of l' and l. Clearly, $\langle (l, l'), (l + k, l + k + c) \rangle \subset C$.

We shall now show that $\langle (l, l'), (l + k, l + k + c) \rangle \supset C$. Let (l_1, l_2) be any pair in C. There exist unique integers k', c' such that $(l_1, l_2) = (l + k', l' + k' + c')$. There are three cases to consider: $c' = 0, c' > 0$, and $c' < 0$.

$c' = 0$: If $c' = 0$, then it must be that $k' \geq 0$. But then $(l_1, l_2) \in \langle (l, l') \rangle$ (by rule 2 of consistently closed relations) and $(l_1, l_2) \subset \langle (l, l'), (l + k, l' + k + c) \rangle$.

$c' > 0$: By construction of l' and l it must be that $k' > -c'$. Consequently $(k+c)+(k'+c') > 0, c + (k' + c') > 0, c' + (k + c) > 0$. By rule 2 for consistently closed relations we can conclude that $(l+(k+c)+(k'+c'), l'+(k+c)+(k'+c')), (l+k+c+(k'+c'), l'+k+c+(k'+$

[6]As is conventional, we shall elide the set former brackets in finite bases.

$c') + c), (l + k' + c' + (k + c), l' + k' + c' + (k + c) + c') \in C$. From the previous proposition we have $(l + (k + c) + (k' + c'), \mathrm{gcd}(c, c')) \in C$. By definition of c this implies that $c \leq \mathrm{gcd}(c, c')$ and thus $c' = ic$ for some $i \in \mathcal{N}$. With the looping rule we can show that $(l+k', l'+k'+c'+c) \in C$ and, consequently, $(l+k', l'+k'+c) \in C$. This shows that $k' \geq k$ by definition of k. And furthermore, since $(l + k', l' + k' + c') = (l + k + d, l' + k + d + ic)$ for some $d, i \in \mathcal{N}$ this shows that $(l + k', l' + k' + c') \in \langle (l, l'), (l + k, l' + k + c) \rangle$.

$c' < 0$: This case can be reduced to the case $c' > 0$, since, by the previous proposition, if $k \geq 0$ then $\{(l, l'), (l + k + c, l' + k)\}$ is also in $\langle (l, l'), (l + k, l' + k + c) \rangle$; if $k < 0$ then $(l + k, l' + c + k), (l + k + (-(c + k)) + c, l' + c + k + (-(c + k)))$ is another way of writing $(l + k, l' + c + k), (l, l')$ that is of the symmetric form with the "looping factor" c in the first component.

We can use this finite representation to construct a polynomial-space algorithm for computing most general uniform semi-unifiers as follows. Let G be an interaction graph representation of SEI S with consistency mapping C. Add pairs $(0, 0)$ into $C(n, n)$ for every node n. Maintain at most two number pairs per node pair. If the set of node pairs B is associated with (n, n'), whose left children are m and m', respectively, then take the number pairs B' associated with (m, m') and compute a new basis B'' of $\langle B \cup B' \rangle$ and associate it with (m, m'), replacing B'. This corresponds to "executing" rule 1 of Figure 6.9. A similar trick can be applied for rule 2. Since the remaining rules are independent of the structure of G, they are already taken care of by the fact that the number pairs associated with node pairs are interpreted as bases of consistently closed relations. The critical part that remains to be shown is how B'' is calculated from B and B'.

For three pairs $(l_1, l_2), (l_1', l_2'), (l_1'', l_2'')$ it is easy to check whether one of them is in the span of the other two. The only interesting case that has to be treated is if this is not the case. Then, w.l.o.g., $B = \{(l_1, l_2), (l_1 + k, l_2 + k + c), (l_1 + k', l_2 + k' + c'\}$ where $l_1, l_2, c, c' > 0, k > -c, k' > -c'$.

Proposition 40 *Let $B = \{(l_1, l_2), (l_1 + k, l_2 + k + c), (l_1 + k', l_2 + k' + c')\}$ where $l_1, l_2, c, c' > 0, k \geq -c, k' \geq -c'$.*

Then $B' = \{(l_1, l_2), (l_1 + k'', l_2 + k'' + c'')\}$, with $k'' = \min\{k, k'\}, c'' = \mathrm{gcd}(c, c')$, is a basis of $\langle B \rangle$.

Proof:

It is sufficient to show that $\langle B' \rangle = \langle B \rangle$. This is analogous to the proof of theorem 21.

This proposition shows that it is possible to compute a basis of a consistently closed relation spanned by three number pairs; of course, this construction can be applied repeatedly to calculate the basis of any finite set of number pairs. We shall denote the basis B above of $\langle B' \rangle$ by $b(B')$. We can now translate the closure rules of Figure 6.9 to operations on *bases* of consistently closed relations and arrive at the following theorem.

Theorem 22 *There is an algorithm A^1 that computes the most general uniform semi-unifier (in a suitable representation) of any SEI $S \in \Gamma(\mathcal{A}_2, V)$ in polynomial space.*

Proof:

Construct an initial interaction graph G for S. Apply the rewriting steps in algorithm A^1 in Figure 6.12 to G' until convergence. The biggest number occurring in any consistency set during its execution is bounded by 2^{2m^2}, where m is the number of nodes in G' (which does not change). This can be shown by observing that the rewrite rules guarantee that numbers only decrease unless the cardinality of some $C(n, n')$ is increased, in which case the biggest number can be at most doubled in the rewritten interaction graph. An increase of cardinality of one of these sets can happen at most $2m^2$ times. This shows that A^1 uses at most polynomial space during the first stage. The second stage — checking for a violation of the descendancy check — is a backtracking algorithm that also uses at most polynomial space. Consequently, algorithm A^1 executes in polynomial space.

We believe that the first stage of algorithm A^1 can be further improved to run in polynomial time, although we cannot see how to speed up the second stage without simplifying the interaction graph from the first stage further by normalizing it with respect the "inverse" rule below, which has been proposed by Kapur *et al.* [54] to arrive at a polynomial-time *decision* algorithm for uniform semi-unification. Note that our algorithm permits us to extract a most general uniform semi-unifier by "running" the interpretation "program" I in the proof of theorem 20.

The inverse of rule 3, specifically

> "If there exist nodes m_1, m_2, n_1, and n_2 such that $m_2 \sim n_2$, $m_1 \xrightarrow{i} m_2$ and $n_1 \xrightarrow{i} n_2$ then merge the equivalence classes of m_1 and n_1,"

is sound, but not complete in our sense. It appears to preserve semi-unifiability in the uniform case (one inequality), even though it does *not* preserve semi-unifiabilty for two or more inequalities and thus is not correct for nonuniform semi-unification. Now arithmetization of algorithm A', which consists of A and the new "inverse" rule above, yields a polynomial-time algorithm.

Theorem 23 *Uniform semi-unifiability is polynomial-time decidable.*

Proof: See [54].

(First stage) Apply the following operations to G' until G' does not change any more.

1. For $n, n_1, n_2, n', n'_1, n'_2 \in N$ such that $E(n) = (n_1, n_2), E(m) = (m_1, m_2)$,

$$\begin{aligned} C(n_1, n'_1) &:= b(C(n_1, n'_1) \cup C(n, n')) \\ C(n_2, n'_2) &:= b(C(n_2, n'_2) \cup C(n, n')) \end{aligned}$$

2. For $n_1, n_2, n_3, n_4 \in N$, if $(l_{01}, l_{10}) \in C(n_0, n_1), (l_{23}, l_{32}) \in C(n_2, n_3), (l_{02}, l_{20}) \in C(n_0, n_2)$,
   if d is the smallest natural number such that $l_{02} + d \geq l_{01}$ and $l_{20} + d \geq l_{23}$, then $C(n_1, n_3) := b(C(n_1, n_3) \cup \{(l_{10} + (l_{02} + d - l_{01}), l_{32} + (l_{20} + d - l_{23}))\})$.
3. $C(n', n) := b(C(n', n) \cup C^{-1}(n, n'))$
   where $C^{-1}(n, n') = \{(l_2, l_1), (l'_2, l'_1)\}$ if $C(n, n') = \{(l_1, l_2), (l'_1, l'_2)\}$.

(Second stage) Execute check$(n, 0)$, with an initially empty stack, for all nodes n and see whether an error is signaled. If so, the normal interaction graph after the first stage has no valid interpretation; if not, it has a valid interpretation.

```
check($n, l$) =
    if there is $(n, l')$ in stack then
        if $C(n, n) \neq \{(0, 0)\}$ or $l' \geq l$ then
            signal error and terminate;
        else
            return;
        end if
    else
        if there is $(n', l')$ such that $(l, l') \in C(n, n')$ and
                $L(n') = f, E(n') = (n'_1, n'_2)$ then
            push $(n, l)$ onto stack;
            check($n'_1, l'$);
            check($n'_2, l'$);
            pop $(n, l)$ off stack;
        else
            return;
        end if
    end if
```

Figure 6.12: Algorithm A[1]

Chapter 7

Decidability: Elementary Combinatorial Properties and Approaches

Semi-unification is, at present, not known to be decidable. There have been several attempts at proving its decidability (or decidability of one of the problems we have shown to be polynomial-time equivalent), but they all failed. In this chapter we introduce graph-theoretic notions that *may* simplify the analysis of the combinatorial properties of semi-unification and eventually lead to a proof of decidability. Semi-unification is widely believed to be decidable; in fact, we conjecture that algorithm A is uniformly terminating. In the first section of this chapter, we present a "normalization" of executions of algorithm A that may be helpful in getting good insight into this problem. In the second section we present a generalization of executions of algorithm A, called graph developments, that are simpler in the sense that they abstract from the specific effect of the rules that affect the equivalence relation in arrow graphs. Our feeling is that this generalized problem is still decidable and may indeed prove easier to solve than the more involved structure of executions of algorithm A.

7.1 Normalized Executions

The *satisfiability problem* for arrow graphs is the problem of deciding whether there is a valid interpretation for a given arrow graph. Since arrow graph representations can be constructed efficiently from SEI's it is clear that semi-unification is polynomial-time reducible to satisfiability of arrow graphs.

A k-colored arrow graph $G = (N, N_F, L, E, A, \sim)$ over $\mathcal{A}$ is *downward closed* if the following closure rules hold.

1. $(\forall m, n \in N_F, m_1, m_2, n_1, n_2 \in N$ if $L(m) = L(n) = f, E(m) = (m_1, m_2), E(n) = (n_1, n_2)$ then $m \sim n \Rightarrow m_i \sim n_i$ for $1 \leq i \leq 2$ and $m \xrightarrow{j} n \Rightarrow m_i \xrightarrow{j} n_i$ for $1 \leq i \leq 2, 1 \leq j \leq k$;

2. $(\forall m, m', n, n' \in N, 1 \leq j \leq k)\,(m \sim m', m \xrightarrow{j} n, m' \xrightarrow{j} n' \Rightarrow n \sim n')$ and $(m \xrightarrow{j} n, m \sim m', n \sim n' \Rightarrow m' \xrightarrow{j} n')$.

It is easy to see that every arrow graph G has a unique smallest downward closure, *closure(G)*, which is simply the arrow graph reached by repeatedly applying the above closure rules as rewrite rules until no longer possible.[1]

Downward closure preserves valid interpretations.

Proposition 41 *I is a valid interpretation of arrow graph G if and only if I is a valid interpretation of* closure(G).

We can factor out the equivalence relation $\sim$ in downward closed arrow graphs to arrive at an essentially equivalent, but simplified, arrow graph. Specifically, we define

$$G/_{\sim} = (N/_{\sim}, N_F/_{\sim}, L/_{\sim}, E/_{\sim}, A/_{\sim}, \iota)$$

where

1. $N/_{\sim}$ is the set of equivalence classes of $\sim$; $[n]_{\sim}$ denotes the equivalence class of $n \in N$;
2. $N_F/_{\sim}$ is the set of equivalence classes that contain some functor node;
3. $L/_{\sim}([n]_{\sim}) = f$ if $L(n') = f$ for some $n' \sim n$; otherwise $L/_{\sim}([n]_{\sim}) = x$ if x is the least variable contained in any $n' \sim n$ (w.r.t. to a given fixed total order on V);
4. $E/_{\sim}([n]_{\sim}) = ([n'_1]_{\sim}, [n'_2]_{\sim})$ if $n \sim n'$ and $E(n') = (n'_1, n'_2)$;
5. $([n]_{\sim}, [n']_{\sim}) \in (A/_{\sim})_i$ if and only if $(n, n') \in A_i$ for $1 \leq i \leq k$;
6. ι is the trivial equivalence relation on $N/_{\sim}$;

if the following three conditions are satisfied:

1. $(\forall n, n' \in N_F)\, n \sim n' \Rightarrow L(n) = L(n')$ (no functor clash);
2. The extended occurs check (rule 4a in Figure 6.6) is not triggered.

If either of these conditions is violated we define $G/_{\sim} = \Box$ where $\Box$ denotes a fixed arrow graph with no valid interpretation. We call any arrow graph with a trivial equivalence relation (i.e., only the identity pairs $(n, n), n \in N(G)$, are in the equivalence relation) *normalized.*

Proposition 42 *Let G be a downward closed arrow graph with equivalence relation $\sim$. Then $G/_{\sim} = \Box$ or*

1. *$G/_{\sim}$ is downward closed; and*
2. *$G/_{\sim}$ is normalized; and*
3. *any valid interpretation of G canonically induces a valid interpretation of $G/_{\sim}$ and vice versa.*

Proof:

(1) and (2) are trivial. For (3) we can verify that for any valid interpretation I of G, $I(n) = I(n')$ if $n \sim n'$, and consequently $I([n]_{\sim})$ is well-defined; conversely, a valid interpretation I of $G/_{\sim}$ extends to G by simply defining $I(n) = I([n]_{\sim})$.

Given any arrow graph G we denote by $\bar{G}$ the normalized, downward closed arrow graph defined by *closure(G)*$/_{\sim}$ where $\sim$ is the equivalence relation in *closure(G).*

[1]This can be made precise by defining arrow graph morphisms and proving uniqueness and minimality by induction on the depth — with respect to dag edges — of the arrow graph.

Proposition 43 *$\bar{G}$ is polynomial-time computable.*

Proof:

A simple adaptation of the union-find based unification algorithm [43,1] yields an algorithm that executes in time $O(kn\alpha(n,n))$ where α is an extremely slow-growing function (see [115]).

We can now define a reduction relation on normalized, downward closed arrow graphs simply by executing rule 4b (Figure 6.6) with subsequent exhaustive application of rules 1, 2, 3, and 4a, which corresponds to computing $\bar{G'}$ from G' after G has been transformed into G' by application of rule 4b at some discrepancy. We say G reduces to $\bar{G'}$ and write $G \Rightarrow \bar{G'}$.

Proposition 44 *Let G be an arrow graph, and let G' be defined as above. Denote the nodes of G with N, and the nodes of G' with N'. Then for any valid interpretation I of G there is a valid interpretation I' of G' such that $I' \mid_N = I$, and, conversely, for every valid interpretation I' of G', $I' \mid_N$ is a valid interpretation of G.*

Proof: Obvious

Note that $\Rightarrow$ defines a reduction relation on downward closed, normalized arrow graphs. The previous propositions guarantee that this reduction relation preserves valid interpretations. A sequence $(G_1, \ldots, G_i, \ldots)$ of downward closed, normalized arrow graphs is a *normalized execution* if $G_i \Rightarrow G_{i+1}$ for $i \geq 1$ and, if it is finite, its last element is irreducible. We say a downward closed, normalized arrow graph G is *solvable* if there exists a finite normalized execution $(G_1, \ldots, G_k)$ such that $G = G_1$ and $G_k \neq \Box$.

Proposition 45 *Semi-unification is polynomial-time reducible to arrow graph solvability.*

Proof:

By the correctness of algorithm A.

The reduction relation $\Rightarrow$ on downward closed, normalized arrow graphs effectively "collapses" the compound effect of *exhaustive* application of rules 1, 2, and 3 in Figure 6.7 of chapter 6. Note also that the exhaustive application of these rules can be done very efficiently since an extended occurs check — which subsumes the ordinary occurs check — is only done once, after rules 1, 2, and 3 are applied exhaustively. The propositions above follow immediately from the fact that algorithm A is just a graph-theoretic reformulation of the "canonical" SEI-rewriting system for computing most general semi-unifiers in section 6.2.

Our hope is that this reduction relation, maybe in connection with the combinatorial structure in the following section, is possible starting point for a *much* deeper understanding of the algebraic and combinatorial structure of executions of algorithm A that will eventually lead to a proof of uniform termination of A and, consequently, of decidability of semi-unification.

7.2 Graph Developments

Inspired by the construction of Kanellakis and Mitchell that shows that ML typing is PSPACE-hard [53] our intuition is that the reduction rules 1 and 3 incorporate the computational "intelligence" of algorithm A in that they "steer" the execution whereas rule 4 and, to a lesser degree, rule 2, simply create the necessary space resources. For this reason we first introduce the notion of (arrow) graph developments. Following, we show that every execution of algorithm A induces an arrow graph development, the finiteness of which can be "lifted back" to show that any execution sequence describing A is finite.

Let $G = (N, N_F, L, E, A)$ be a graph.[a] Define the reduction relation $\rightarrow_r$ by the following two rules.

1. If there exist $m, m', m'', n, n', n'' \in N$ such that $L(m) = L(n) = f, E(m) = (m', m''), E(n) = (n', n'')$ and $m \rightarrow n$, but $m' \not\rightarrow n'$ (or $m'' \not\rightarrow n''$), then

$$G \rightarrow_r G[A := A \cup (m', n')](\text{or } G \rightarrow_r G[A := A \cup (m'', n'')], \text{respectively}).$$

2. (a) If there exist $m, n \in N$ such that n is a proper descendant of m and there is a (possibly empty) arrow path from m to n, then

$$G \rightarrow_r \Box.$$

   (b) if rule 2a above does *not* apply and there exist $m, m', m'', n \in N$ such that $L(m) = f, E(m) = (m', m''), L(n) \in V$, then

$$\begin{aligned} G \quad \rightarrow_r \quad & G[N := N \cup \{n', n''\}, L := L\{n \mapsto f, n' \mapsto l', n'' \mapsto l''\}, \\ & E := E\{n \mapsto (n', n'')\}]. \end{aligned}$$

   where n' (or n'') is either an old node, $n' \in N$, or a new node, $n' \notin N$, and if n' is a new node then $l' = x'$ for a new variable x', otherwise $l' = L(n')$; similar for n''.

In all these cases the node m is the *hinge* of the rule application.

[a]We shall write $n \rightarrow m$ for $(n, m) \in A$.

Figure 7.1: Graph development Rules

In this section we shall consider arrow graphs without an equivalence relation and with arrows of only one color; i.e., they consist of a term graph and arrows (all of the same color) only. For convenience' sake we shall simply call them *graphs*. These graphs can be identified with arrow graphs that have only a trivial equivalence relation on their nodes. In this sense the notions of descendant, arrow path and so forth carry over from arrow graphs to graphs.

We shall now introduce a reduction relation, also denoted by $\rightarrow_r$, between graphs. It is defined by two rules given in Figure 7.1. The surface similarity of these rules with arrow graph reduction rules 2 and 4 in Figure 6.7 is not coincidental and will be made precise just below.

Graphs for which no rule is applicable (in particular $\Box$) are *normal graphs.*

Definition 18 *(Graph development)*

A graph development *is a (possibly infinite) sequence $\mathcal{G} = (G_1, \ldots, G_i, \ldots)$ of graphs where G_{i+1} is derived from G_i by application of one of the rules in Figure 7.1 for all $i \geq 1$ and, if G is finite then the last component in G is a normal graph.*

The limit graph $\lim\mathcal{G}$ *of a graph development $\mathcal{G} = (G_1, \ldots, G_i, \ldots)$ where $G_i = (N^i, N_F^i, L^i, E^i, A^i)$ is $\Box$ if G is finite and its last component is $\Box$; otherwise it is defined by (N, N_F, L, E, A) where*

$$\begin{aligned} N &= \{n : (\exists i) n \in N^i\} \\ N_F &= \{n : (\exists i) n \in N_F^i\} \\ L(n) &= \begin{cases} f, & if (\exists i) L^i(n) = f \\ x, & otherwise, \; and \; L^i(n) = x \, for \; some \; i \end{cases} \end{aligned}$$

$$E(n) \quad = \quad (n', n'') \; if (\exists i) \; E^i(n) = (n', n'').$$

The first component, G_1, of a graph development $\mathcal{G} = (G_1, \ldots, G_i, \ldots)$ is called the initial graph *of $\mathcal{G}$. A node in* $\lim \mathcal{G}$ *or in any of the graphs in G is an* original *(or* old*) node if it occurs in the initial graph of G; otherwise it is a* new node.

Every execution of algorithm A whose final, normal arrow graph is not $\Box$ defines a graph development in a canonical fashion. Consider the final arrow graph G of an execution and its equivalence relation. This equivalence relation can be "factored" out from every arrow graph in the execution leading up to G in almost the same way in which normalized arrow graphs are formed from downward-closed arrow graphs in section 7.1.

Let us now consider graph developments whose limit graph is not $\Box$. For any $\mathcal{G} = (G_1, \ldots, G_i, \ldots)$ we can define an equivalence relation on the nodes in G and a partial order on the resulting equivalence relations. For $\lim \mathcal{G} = (N, N_F, L, E)$ define $n \leq n'$ for $n, n' \in N$ if $n \rightarrow n'$ (in $\lim \mathcal{G}$) or $E(n') = (n, n'')$ or $E(n') = (n'', n)$ for some $n'' \in N$. We can take the reflexive-transitive closure of $\leq$ and then factor out the equivalence relation $\cong$, $n \cong n' \Leftrightarrow n \leq \ldots \leq n'$and $n' \leq \ldots \leq n$, which defines a partial order, also denoted by $\leq$, on equivalence classes of $\cong$. The equivalence class containing node n shall be denoted by $[n]$.

We call a graph development G *fair* if for every node in G that becomes a hinge for a rule application the corresponding rule is eventually executed.

Proposition 46 *Let G be a fair graph development with* $\lim \mathcal{G} = (N, N_F, L, E) \neq \Box$, *and let $\leq$ be the partial order on $\cong$-equivalence classes of N defined above.*

For all nodes $n, n', n'' \in N$, if $E(n) = (n', n'')$ then $[n'] < [n]$ and $[n''] < [n]$.

Proof:

It is clear by definition that $[n'] \leq [n]$ and $[n''] \leq [n]$. We need to show that $[n] \not\leq [n']$. Let us assume $[n] \leq [n']$. By definition of $\leq$ there is a sequence of nodes $(n_{00}, n_{01}, n_{10}, n_{11}, \ldots, n_{k0}, n_{k1}), k \geq 0$ such that $n_{00} = n, n_{k1} = n'$ and for $0 \leq i \leq k$ there is a (possibly empty) arrow path from n_{i0} to n_{i1} and for $0 \leq i \leq k-1$ the node n_{i1} is a child of $n_{(i+1)0}$. Since G is fair we can show by induction on the length of these sequences that there exists a proper descendant m of n_{k0} in G such that there is an arrow path from $n'(= n_{k1})$ to m. Consequently there is an arrow path from n_{k0} to m. But this means that n_{k0} must be a hinge for applying the "extended occurs check" rule 2a in a component of G. Since G is fair by assumption this means that the extended occurs check rule is applied at some point in G and consequently $\lim \mathcal{G} = \Box$. But this is in contradiction to our assumption that the limit graph is not $\Box$.

This proposition shows that proper $<$-inequalities hold between (the equivalence class) of a child and (the equivalence class of) its parent. This is critically due to the extended occurs check rule, rule 2a, since the notion of fairness mandates that every rule that can possibly be executed at some node eventually is. In fact it is easy to give a (necessarily infinite and unfair) graph development in which the resulting (infinite) limit graph has equivalence classes that contain a child and its parent.

This separation of equivalence classes along parent-child edges is crucial in the following lemma. For $G = (N, N_F, L, E, A)$ and $C \subset N$ we define $\mathrm{Env}_G(C) = \{(n, n') \in A \mid n \in C \text{or } n' in C\}$.

Lemma 47 *Let G be a fair graph development with* $\lim \mathcal{G} = (N, N_F, L, E, A) \neq \Box$ *and let* $G' = (N', N'_F, L', E', A')$ *be the initial graph of G.*

For any maximal equivalence class C in $\lim \mathcal{G}$ *we have*

1. *For all $n \in C$, $n \in N'$ and n is* not *a child in G' of any node $n' \in N'$.*
2. *$Env_{\lim \mathcal{G}}(C) = Env_{G'}(C)$*

Proof:

1. By assumption, C is a maximal equivalence class in $\lim\mathcal{G}$ with respect to $\leq$. If $n \in C$ is not in N', then it must have been introduced by rule 2b since this is the only rule that adds new nodes. But then n would have to be the child of some node n' in a component of G and, consequently, in $\lim\mathcal{G}$, which, by proposition 46, would contradict the assumption that the equivalence class, C, of n is maximal in $\lim\mathcal{G}$. If n were the child of a node n' in G' then, again, C could not be maximal since n would also be a child of n' in $\lim\mathcal{G}$.
2. By inspection of the graph development rules it is clear that the containment $\mathrm{Env}_{\lim \mathcal{G}}(C) \supset \mathrm{Env}_{G'}(C)$ holds. Assume it is a proper superset. Then a new arrow, with a node $n \in C$ at its head or at its tail, must have been introduced by rule 1 since this is the only rule that introduces new arrows. But this means that n has a parent in $\lim\mathcal{G}$ and, again, it follows by proposition 46 that C is not maximal contradicting our assumption.

This lemma guarantees that any group of nodes that turns out to be a maximal equivalence class in the limit graph of a fair graph development, *all* the arrows between them, and *all* the arrows whose head is one of these nodes are actually already present in the initial graph of the graph development. We cannot predict which group that will be by looking at the initial graph since rule 2b can wildly pick any old node for a child (thus making that node an element of a nonmaximal equivalence class), but the lemma guarantees that there *exists* one, no matter what (literally) unpredictable turns rule 2b takes. Note that we have not proved that the limit graph of a fair graph development actually has maximal equivalence relations.

Before we present the main theorem we need another lemma. In a graph G the *sources* of a node n is defined to be the set of all nodes n' in G such that there is a (possibly empty) arrow path from n' to n. The *independent sources* of n are all those sources of n that have only themselves as a source. Note that every finite graph development is necessarily fair.

Lemma 48 *Let G be a finite graph development with $\lim\mathcal{G} \neq \Box$ and initial graph G'. Then, for any node n in $\lim\mathcal{G}$, all independent sources of n are nodes in G'.*

Proof:

This can be shown by (finite) induction on the index of the component graphs in the graph development. If we insisted on "normalized" graph developments, in which rule 2b is only executed when none of the other rules is applicable (which is a good idea anyway since it simplifies the process of looking for hinges for the extended occurs check rule, rule 2a), this would indeed be straightforward to prove. Since there is a slight complication in "unnormalized" graph developments, we shall momentarily generalize the notion of a source. In rule 2b we say there is a "hop" from m' to n' and from m'' to n''. A node m in G_j is a *phantom source* of n if there is a sequence of nodes $(n_0, \ldots, n_k)$ such that $m = n_0, n = n_k$, and, for $1 \leq i \leq k$, there is a hop from n_{i-1} to n_i or $n_{i-1} \rightarrow n_i$ in G_j. Independent phantom sources are defined analogously to independent sources. We claim that for all graphs in a graph development the set of independent phantom sources is already contained in the initial graph.

The claim holds trivially for the empty prefix of a given graph development of length n.

1. If rule 1 is applied to get G_i from G_{i-1}, let us denote the tail of the new arrow by m and its head by n. The sources of m are added to the sources of every node n' that n is a source of. Since the independent sources of n' are then a subset of the independent sources of m and of the independent sources n' had before the rule application, by induction we can conclude that the independent sources of n' are contained in the initial graph G. (For the other nodes, not affected by this rule application, nothing changes.)
2. If rule 2b is applied and no new node is introduced the claim remains true trivially. If a new node n is introduced, then there is a hop from a node m already in G_{i-1} to n, and the claim remains true.

Since G is a finite graph development the claim holds true for the final graph of G. But, in the final graph, for every hop there is also a corresponding arrow, and consequently, the set of independent phantom sources is also the set of independent sources. This proves the lemma.

Let us now define the size of a graph, $|G|$, simply as the number of nodes it contains. (The size of $\Box$ is undefined.)

Lemma 49 *For every finite graph development G with* $\lim\mathcal{G} \neq \Box$ *and* $|\lim\mathcal{G}| = s$ *whose initial graph* G' *has size* $|G'| = t > 1$ *and that has a maximal equivalence class* E *of size* k *with a node* $n \in E$ *that is functor labeled in the initial graph of G there is a graph development* $\mathcal{G}_\infty$ *with* $\lim\mathcal{G}_\infty \neq \Box$ *and* $|\lim\mathcal{G}_\infty| = s - k$ *and the initial graph* G'_1 *of* $\mathcal{G}_\infty$ *has size* $|G'_1| = t - k$.

Proof:

Since G is finite, it is fair, and its limit graph has a maximal equivalence relation C. Now, by assumption there is an equivalence class E with a functor labeled node n. Let us only treat the case where $k = 1$; i.e., n is the only node in E. Since n is functor labelled in the initial graph of G, there are children n', n'' of n. Now, let N' and N'' be the independent source of n', respectively n'' in the limit graph. By lemma 48, all elements of N' and N'' are also in the initial graph of G. Place arrows from any independent source of n' to n' and from every independent source of n'' to n'' in the initial graph of G, possibly adding new arrows, and delete node n along with the edges to its children. This results in a new initial graph graph of size $s - 1$. Now we can "simulate" G on the smaller initial graph by simply copying the steps from G do not involve, directly or indirectly, node n and, otherwise, substituting steps involving some of the added arrows whenever node n is involved.

If we could establish a (recursive) lower bound (as a function of the size of the limit graph and possibly the size of the initial graph) on the size of the limit graph of some graph development with an initial graph that has fewer nodes than the initial graph of any given finite graph development, even in the case where all nodes in maximal equivalence classes are variable labeled in the initial graph, then we could prove, by induction, an upper bound on the size of any limit graph as a function of the size of the initial graph. This is so since any graph development on an initial graph of size 1 has a limit graph of size 1. This would establish decidability of semi-unification since the existence of an infinite execution of algorithm A induces an infinite graph development.

We might be tempted to "loosen" the notion of graph development even more by requiring an constant upper bound on the number of outarrows any node in a graph can have, but allowing arbitrary insertion of arrows, not only in the case of rule 1. But then it is fairly easy to construct an infinite graph development as long as at least two outarrows are permissible. Since a generalization of executions of algorithm A, as the notion of graph developments is, is only sensible if it admits a proof of termination by showing that only finite developments are possible this further generalization is useless.

Chapter 8

Implications for Programming Language Design

In this chapter we attempt to shed some light on a somewhat puzzling observation: that polymorphic type inference is theoretically intractable and, as such should be only marginally usable, yet experience with declaration-free polymorphic languages bears witness to its practical utility. In section 8.1 we offer some general considerations to suggest that the apparent practicality of type inference is not just a lucky coincidence, and in section 8.2 we briefly formalize some of our considerations.

8.1 Theoretical Intractability and Practical Utility of Polymorphic Type Inference

Some of the results of the previous chapters seem to suggest that polymorphic type inference (as modeled by the Mycroft Calculus) has no place in programming language design. After all, the type inference problem is at least PSPACE-hard, which is already beyond the point of what is conventionally considered tractable, and likely it is much harder than that: At present even the decidability question is not solved.

On the other hand, some theoretical results and preliminary practical experience suggest that this evaluation may yet be too pessimistic.

First of all, the principal typing property of the Mycroft Calculus guarantees a well-defined notion of what *the* typing of a program should be, and this notion can very intuitively be interpreted as the "most general" typing possible. Even though, at this time, the decidability of both the Mycroft Calculus and the (implicit) Second Order λ-calculus is open, the Mycroft Calculus has the appealing principal typing property, which is in contrast to the Second Order λ-calculus where no good notion of "principality" for a λ-expression is known.

Secondly, there is a relatively simple algorithm, algorithm A, for computing principal typings (in the more general sense of computing typing derivations in the "syntax-oriented" version of the Mycroft Calculus) that, due to the principal typing property, does not necessitate any backtracking or other complicated control mechanisms. This can be seen as a sign of "implementability" and as a preliminary indicator that the type inference problem may prove realistically usable since many problem instances will admit rapid computation of their principal types.

Thirdly, languages such as ML, Miranda, and B have been in use for several years now, and the type checking phases in these systems have been sufficiently efficient in actual usage to help promulgate, for about ten years, the myth that ML type checking is *theoretically* efficient in the sense that it was believed to have a worst-case polynomial running time of low degree. The fact that B's type inference algorithm is actually incomplete (with respect to B's typing discipline), but that this apparently hadn't been noticed, only corroborates our appraisal that type inference problems encountered in actual programming practice are of the kind that admit rapid computation of principal types or rapid detection of type errors. Of course, since the polymorphic languages in question are still used rather infrequently, it is too early to give much weight to these empirical observations. We shall attempt to argue, though, that the apparent practicality of polymorphic type inference in the face of theoretical infeasibility results is not a random phenomenon.

A conventional remedy for eliminating problems with type inference is to mandate explicit, fully typed declarations of variables, parameters and other basic syntactic units. Observe, for example, that type checking in the "explicit" Second Order λ-calculus is easy in the sense that there is a fast polynomial time algorithm for checking the type correctness of a fully typed λ-expression. Applying this sort of remedy to the Mycroft Calculus highlights, though, why type *checking* (with explicit type information embedded in the program) is no more "practical" than type *inference* (with no or only optional type information in the program).

The culprit for the theoretical intractability of the Mycroft Calculus (and the Milner Calculus) is the fact that the type information of a program can be extra-ordinarily bigger than the untyped program; in particular, it is at least exponentially bigger [53]. Now, writing a 200-line (untyped) program whose principal type is bigger (measured, for example, in terms of the "tree" size of the final arrow graph of the corresponding semi-unification problem[1]) than the number of atoms in the universe is no more impractical than writing the program *with* this typing information in the first place. Even though both these cases seem to have the same "intuitive" complexity they are treated very differently in conventional complexity analysis since the two input sizes are dramatically different.

This may be seen as a plea to measure complexity in terms of the sizes of the input program and its computed principal type. This would permit comparison of the efficiency of different (sound and complete) type inference algorithms by comparing their performance on typable inputs, even in the case where they don't terminate for some untypable inputs. Yet this is not quite satisfactory in explaining the apparent practicality of type inference. In particular it does not question the "legitimacy" of a short program that has a typing of inconceivable size.

We feel that the formalization of type inference in logical calculi has failed to take the *intensional* character of types and typings into account. *Types* and typings are generally viewed as abstractions of the *behavior* of programs and their parts, and, by analogy to types and program behaviors, *type descriptions* are meant to be abstractions of the *programs* themselves. If the complete inferred type information of a program is exponentially bigger than the (untyped) program itself, we think it unreasonable to say the type information is an *abstract* description of the program. Either the type description mechanism is inadequate for capturing the intended abstraction of behavior or the program at hand does *not* have a suitable abstract description of its behavior. The first explanation points toward a problem with the whole language, an issue that will have to be addressed by language designers. Given a fixed static

[1]Measuring the size of type information in this way can be justified as follows. When admitting — or requiring — explicit type information in programs, this type information is presented by type expressions of the kind we have used, and by no other mechanism that might conceivably encode type information in some other, possibly more compact way. Since the "size of the input" is usually counted as the number of symbols in the input (with or without taking bit-complexity into account), this amounts to determining the size of all explicit typing information in a program as the sum of the string sizes of the type expressions occurring in it. Since every part of a final arrow graph corresponding to (Mycroft Calculus) type inference for program e is represented in a type expression occurring in the full typing for e, this full typing information is at least as big as the *tree* size of the final arrow graph; i.e., the number of nodes of the final arrow graph once it is "blown up" into a tree (or forest). If the fully typed program can be written with type abbreviations of the sort **lettype** $t = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ **in** ..., then the type information can be represented in, asymptotically, the same space as the size of the final arrow graph. But this has the disadvantage that principal types are not necessarily the "smallest types", and then determining resource-bounded typability becomes difficult again.

typing discipline, however, and its implicit insistence that only behavior that is expressible in it should be considered desirable, the second explanation should be interpreted as saying that the program at hand has no "reasonable" abstract description of its behavior and thus should be considered unacceptable — type-incorrect.

Theoretical type inference calculi are motivated by *extensional* considerations: two descriptions are considered completely interchangeable if they *denote* the same semantic objects, regardless of any "syntactic" properties of the descriptions (such as the size). Consequently, the motivation of type descriptions as syntactic abstractions of programs (and not only program behavior) is lost in the formalization of "practical" type inference by typed λ-calculi. If we try to recapture some of this connection by requiring that a λ-expression e only be considered "effectively well-typed" whenever it is typable in the sense of the Mycroft Calculus *and* its principal type is at most polynomially bigger than e itself, then it is easy to see that effective well-typing is (theoretically) feasible. This is made precise in section 8.2.

Unfortunately, this does not explain the significance, if any, of the extended occurs check in algorithm A that we conjecture makes A a uniformly terminating algorithm for semi-unification. If resource bounds on the sizes of typings are given we could run A — or Meertens' algorithm AA or Mycroft's algorithm — either until a principal type is found or the resource bounds are exceeded. It appears that, in practice, this check will catch many typing errors early on without exhausting the possibly big resource bounds. As a matter of principle, it seems that the *requirement* of resource bounds in type systems is a bad idea[2], whereas they appear to be a good *property* of a type system. In other words, it is preferable to devise a syntax-directed type system whose axioms and rules guarantee resource-boundedness instead of explicitly imposing a *global* restriction that mandates explicit resource bounds. We think this is a problem worthy of attention in the type system design arena, but not so much in the area of programming language semantics. After all, static typing disciplines are semantically incomplete anyway (that is, there are programs that are considered statically type-incorrect even though they would never run into a type-incompatibility at run-time), and resource-bounded static typing systems are just "a tad more" incomplete.

If we consider, in general, (derivable) typings as "witnesses" to the fact that a program is well-typed, then typing problems whose witnesses are required to be polynomial-sized fall into two main complexity classes: P and NP.[3] This is so since we assume that any reasonable typing discipline has a polynomial time type checking problem for programs that are completely decorated with typing information. If we consider the "typing" problem[4] of determining whether there is an assignment of (polynomial-sized) type expressions to function definitions in a language with Ada-style overloading, but without explicit type declarations (Ada requires such explicit declarations), it can be shown that this problem is NP-complete [1, exercise 6.25], whereas the resource-bounded polymorphic type inference problem is in P. This lends some technical expression to the intuition that "overload resolution" as above is much harder than polymorphic type inference; also, in practical terms, since overload resolution requires a backtracking algorithm, polymorphic type inference should be expected to fare much better in practice than this liberal sort of overload resolution. Note also that overload resolution has no principal typing property.

8.2 Resource-Bounded Polymorphic Type Inference

Consider the type inference system in Table 8.1, which we shall call the *explicit* Mycroft Calculus.

We can define notions of typability and type inference as usual. Typed λ-expressions are defined by the grammar

$$e \quad ::= \quad x \mid \lambda x : \tau . e \mid (ee') \mid$$

[2]Imagine error messages of the sort "Well, so far everything was okay, but this type expression here is a little bit too big."

[3]We make the standard assumption that $P \neq NP$.

[4]Some people would not consider this overload resolution problem an example of a typing problem.

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over monotypes; σ, σ' over polytypes. The following are type inference axiom and rule schemes.

Name	Axiom/rule
(TAUT)	$A\{x : \sigma\} \supset x : \sigma$
(GEN)	$\dfrac{A \supset e : \sigma \qquad (\alpha \text{ not free in } A)}{A \supset e : \forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e : \forall\alpha.\sigma}{A \supset e : \sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x : \tau'.e : \tau' \to \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \to \tau \qquad A \supset e' : \tau'}{A \supset (ee') : \tau}$
(LET-P)	$\dfrac{A \supset e : \sigma \qquad A\{x : \sigma\} \supset e' : \sigma'}{A \supset \textbf{let}\ x : \sigma = e\textbf{in}\ e' : \sigma'}$
(FIX-P)	$\dfrac{A\{x : \sigma\} \supset e : \sigma}{A \supset \textbf{fix}\ x : \sigma.e : \sigma}$

Table 8.1: Type inference axioms and rules for explicit Mycroft Calculus

$$\begin{aligned}&\mathbf{let}\ x : \sigma = e' \ \mathbf{in}\ e \mid \\ &\mathbf{fix}\ x : \sigma . e\end{aligned}$$

where τ ranges over monotypes, and σ over polytypes, as usual. For every typed λ-expression e there is a unique underlying untyped λ-expression, $\bar{e}$, derived by erasing all mention of types in the typed λ-expression (and all colons); e is called a *typed version* of $\bar{e}$. Clearly, every typed λ-expression has a principal type in the explicit Mycroft Calculus with respect to a given type assignment. The following proposition should not come as a surprise.

Proposition 50 *There is a polynomial time algorithm for computing the principal type of a typed λ-expression or indicating untypability.*

We can now formally define a resource-bounded restriction of the Mycroft Calculus. Let p be a fixed polynomial of one variable, and let $|e|$ be the number of symbols in a typed or untyped λ-expression e, and let eMM stand for the explicit Mycroft Calculus. Define

$$MM^p = \{\bar{e} : \exists A, \sigma \mid eMM \vdash A \supset e : \sigma \text{ and } |e| \leq p(|\bar{e}|)\}$$

A simple way to think about this set is to recognize that, if $A \supset e : \sigma$ is derivable in eMM, then $A \supset \bar{e} : \sigma$ is derivable in MM. The second requirement encodes the fact that MM^p considers only those untyped λ-expressions type-correct that have a typed equivalent whose type information is at most polynomially bigger than the untyped λ-expression itself.

Theorem 24 *MM^p is polynomial-time decidable.*

Proof:

Execution of rule 4 in algorithm A only makes the *tree* size of the initial arrow graph properly bigger. Since the other rules cannot reduce the tree size of the arrow graph (note though, that they can reduce the number of equivalence classes in the arrow graph) and they can be executed at most polynomially many times with respect to the "current" arrow graph without forcing application of rule 4, and since the tree size of the arrow graph can be computed in time polynomial in the number of nodes in the arrow graph, it follows that rule 4 can be applied at most polynomially many times without exceeding the bound given by p. Consequently, computing the "principal" typed version of a λ-expression e can be done in polynomial time, and since every other typed version of e that satisfies the typing rules is at least as big as the principal one, this proves the theorem.

It would be interesting to see whether this theorem also holds true if (monomorphic) type abbreviations of the form **let type** $s = \tau$ **in** ... are allowed.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986. Addison-Wesley, 1986, Reprinted with corrections, March 1988.

[2] E. Allender. Personal communication. April 1988.

[3] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1984.

[4] C. Ben-Yelles. *Type-assignment in the Lambda-calculus.* PhD thesis, University College, Swansea, 1979.

[5] W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9(1):1–35, 1977.

[6] H. Boehm. Partial polymorphic type inference is undecidable. In *Proc. 26th Annual Symp. on Foundations of Computer Science*, pages 339–345, IEEE, Oct. 1985.

[7] W. Buettner. Unification in the data structure sets. In *Proc. 8th Int'l Conf. on Automated Deduction*, pages 470–488, Springer-Verlag, 1986. Lecture Notes in Computer Science, Vol. 230.

[8] R. Burstall, D. MacQueen, and D. Sannella. Hope: an experimental applicative language. In *Stanford LISP Conference 1980*, pages 136–143, 1980.

[9] L. Cardelli. A semantics of multiple inheritance. In *Int. Symp. on Semantics of Data Types*, pages 51–68, Springer-Verlag, 1984.

[10] L. Cardelli. A semantics of multiple inheritance. *Information and Computation (Information and Control)*, 76:138–164, 1988.

[11] L. Cardelli. Structural subtyping and the notion of power type. In *Title of book Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 70–79, ACM, Jan. 1988.

[12] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 202–212, ACM, Jan. 1989.

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

[14] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York and London, 1973.

[15] C. Chou. *Relaxation Processes: Theory, Case Studies and Applications.* Master's thesis, UCLA, February 1986. technical report CSD-860057.

[16] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. *INRIA Centre Sophia Antipolis*, RR No. 529, May 1986.

[17] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, 1984.

[18] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for lambda-terms. *Archive f. math. Logik und Grundlagenforschung*, 19:139–156, 1979.

[19] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL), Vol. 1 (Invited Lectures)*, pages 151–184, Springer-Verlag, Apr. 1985. Lecture Notes in Computer Science, Vol. 203.

[20] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[21] H. Curry and R. Feys. *Combinatory Logic.* Volume I of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1958.

[22] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[23] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[24] N. de Bruijn. A survey of the project AUTOMATH. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

[25] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[26] E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

[27] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proc. 2nd European Symp. on Programming*, pages 94–114, Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[28] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Symp. on Logic in Computer Sciene*, pages 61–70, IEEE, Computer Society, Computer Society Press, June 1988.

[29] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[30] W. Goldfarb. The undecidability of the second order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[31] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM POPL*, pages 119–130, 1978.

[32] R. Harper. *Introduction to Standard ML (Preliminary Draft).* Technical Report, University of Edinburgh, February 1986.

[33] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 194–204, June 1987.

[34] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 308–319, Springer-Verlag, Mar. 1987. Lecture Notes in Computer Science, Vol. 250.

[35] F. Henglein. *The Milner-Mycroft Calculus.* Technical Report (SETL Newsletter) 223, New York University, April 1988.

[36] F. Henglein. *A Polymorphic Type Model for SETL.* Technical Report (SETL Newsletter) 221, New York University, July 1987.

[37] F. Henglein. Simple type inference and unification. Oct. 1988. New York University, Computer Science Department, SETL Newsletter 232.

[38] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*, pages 184–197, ACM, ACM Press, July 1988.

[39] J. Herbrand. Recherches sur la theorie de la demonstration. In *Ecrits logiques de Jacques Herbrand*, PUF, Paris, 1968. thèse de Doctorat d'Etat, Université de Paris (1930).

[40] C. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot.* PhD thesis, MIT, 1971.

[41] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[42] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus.* Volume 1 of *London Mathematical Society Student Texts*, Cambridge University Press, 1986.

[43] J. Hopcroft and R. Karp. *An Algorithm for Testing the Equivalence of Finite Automata.* Technical Report TR-71-114, Dept. of Computer Science, Cornell U., 1971.

[44] W. Howard. The formulae-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, 1980.

[45] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[46] G. Huet. *Résolution d'equations dans des langages d'ordre 1, 2, ..., omega.* PhD thesis, Univ. Paris VII, Sept. 1976.

[47] G. Huet. A unification algorithm for typed *lambda*-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[48] H. Hussmann. Unification in conditional equational theories. In *European Conf. on Computer Algebra (EUROCAL)*, pages 543–553, Springer-Verlag, April 1985. Lecture Notes in Computer Science, Vol. 204; also Universitaet Passau technical report MIP-8502, Jan. 1985.

[49] J. Jaffar and J. Lassez. Constraint logic programming. 1986. IBM Manuscript.

[50] L. Jategaonkar and J. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proc. ACM LISP and Functional Programming Conf.*, pages 198–211, ACM, July 1988.

[51] G. Johnson and J. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proc. 13th Annual ACM Symp. on Principles of Programming Languages*, pages 44–57, ACM, Jan. 1986.

[52] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symosium on Programming, Lecture Notes in Computer Science, Vol. 300*, pages 131–144, Springer-Verlag, Nancy, France, March 1988.

[53] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing (extended abstract). In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, ACM, January 1989.

[54] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proc. Foundations of Software Technology and Teoretical Computer Science*, Jan. 1989.

[55] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In *Proc. 8th Int'l Conf. on Automated Deduction*, pages 489–495, Springer-Verlag, 1986. Lecture Notes in Computer Science, Vol. 230.

[56] A. Kfoury. Announcement to the types mailing list. May 1988.

[57] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, June 1989.

[58] A. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 58–69, ACM, ACM Press, Jan. 1988.

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

[60] R. Kowalski. *Logic for Problem Solving. Artificial Intelligence Series*, North-Holland, 1979.

[61] J. Krajicek and P. Pudlak. The number of proof lines and the size of proofs in first order logic. *Archive for Mathematical Logic*, 27(1):69–84, 1988.

[62] D. Lankford and D. Musser. A finite termination criterion. Unpublished draft, USC Information Sciences Institute, 1978.

[63] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, Morgan Kauffman, 1987.

[64] H. Leiß. On type inference for object-oriented programming languages. In *Proc. 1st Workshop on Computer Science Logic*, Springer-Verlag, Lecture Notes Computer Science, Vol 329, Oct. 1987.

[65] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98, ACM, Jan. 1983.

[66] S. MacLane and G. Birkhoff. *Algebra.* Macmillan, 1979. 2nd edition.

[67] D. MacQueen. Personal communication. June 1988.

[68] D. MacQueen. References and weak polymorphism. 1988. Standard ML of New Jersey, Documentation.

[69] D. MacQueen. Using Dependent Types to Express Modular Structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286, ACM, Jan. 1986.

[70] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[71] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[72] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[73] P. Martin-Lof. Constructive mathematics and computer programming. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184, Prentice-Hall, 1985.

[74] L. Meertens. Incremental polymorphic type checking in B. In *Proc. 10th ACM POPL*, pages 265–275, 1983.

[75] L. Meertens and S. Pemberton. Description of B. *SIGPLAN Notices*, 20(2):58–76, Feb. 1985.

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

[77] P. Mishra and U. Reddy. Declaration-free type checking. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 7–21, ACM, Jan. 1985.

[78] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

[79] J. Mitchell. Polymorphic type inference and containment. *Information and Control*, 76:211–249, 1988.

[80] J. Mitchell. Type inference and type containment. In *Proc. Int'l Symp. on Semantics of Data Types, LNCS 173*, pages 257–277, June 1984.

[81] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*, ACM, Jan. 1988.

[82] J. Mitchell and G. Plotkin. Abstract Types have Existential Type. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 37–51, ACM, Jan. 1985.

[83] G. Monteleone. Generalized conjunctive types. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 242–249, ACM, Jan. 1989.

[84] J. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.

[85] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[86] A. Mycroft and R. O'Keefe. A polymorphic type system for PROLOG. *Artificial Intelligence*, 23:295–307, 1984.

[87] D. Parker. Partial order programming. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, ACM, ACM Press, January 1989.

[88] D. Parker and R. Muntz. A theory of directed logic programs and streams. In *Proc. Int'l Conf. on Logic Programming*, pages 620–650, August 1988.

[89] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[90] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[91] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. 1988 ACM LISP and Functional Programming Conf.*, pages 153–163, ACM, July 1988.

[92] F. Pfenning and P. Lee. *LEAP: A Language with Eval and Polymorphism.* Technical Report ERGO-88-065, Carnegie-Mellon University, September 1988.

[93] G. Plotkin. *Lattice-Theoretic Properties of Subsumption.* Technical Report MIP-R77, Univ. of Edinburgh, 1970.

[94] G. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[95] D. Prawitz. An improved proof procedure. *Theoria*, 26:102–139, 1960.

[96] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[97] P. Purdom. Detecting looping simplifications. In *Proc. 2nd Conf. on Rewrite Rule Theory and Applications (RTA)*, pages 54–62, Springer-Verlag, May 1987.

[98] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 77–88, ACM, Jan. 1989.

[99] J. Reynolds. *Preliminary Design of the Programming Language Forsythe.* Technical Report CMU-CS-88-159, Carnegie-Mellon University, June 1988.

[100] J. Reynolds. Three approaches to type structure. In *Proc. TAPSOFT*, pages 97–138, Springer-Verlag, 1985.

[101] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, pages 408–425, Springer-Verlag, 1974.

[102] J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–152, 1970.

[103] J. Reynolds. Using category theory to design conversions and generic operators. In *Proc. Semantics-Directed Compiler Generation*, pages 211–258, Springer-Verlag, 1980. Lecture Notes in Computer Science, Vol. 94.

[104] J. Robinson. *Logic: Form and Function – The Mechanization of Deductive Reasoning.* North-Holland, New York, 1979.

[105] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[106] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[107] P. Ruzicka and I. Privara. An almost linear Robinson unification algorithm. In *Proc. Mathematical Foundations of Computer Science*, Springer Verlag Lecture Notes in Computer Science, Vol. 324, 1988.

[108] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[109] J. Siekmann. Universal unification. In *Proc. 7th Int'l Conf. on Automated Deduction*, pages 1–42, Springer-Verlag, 1984. Lecture Notes in Computer Science, Vol. 170, Springer-Verlag.

[110] M. Solomon. Type definitions with parameters. In *Proc. 5th Annual ACM Symp. on Priniciples of Programming Languages*, pages 31–38, ACM, Jan. 1978.

[111] R. Statman. Personal communication. April 1989.

[112] R. Statman. Personal communication. May 1988.

[113] L. Sterling and E. Shapiro. *The Art of PROLOG*. MIT Press, 1986.

[114] M. Stickel. A unification algorithm for associative-commutative functions. *J. Assoc. Comput. Mach.*, 28(3):423–434, July 1981.

[115] R. Tarjan. *Data Structures and Network Flow Algorithms*. Volume CMBS 44 of *Regional Conference Series in Applied Mathematics*, SIAM, 1983.

[116] M. Tofte. Type inference for polymorphic references. 1988. Submitted to Information and Computation.

[117] D. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, Dec. 1986.

[118] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76, ACM, Jan. 1989.

[119] M. Wand. Finding the source of type errors. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 38–43, IEEE, June 1986.

[120] M. Wand. A semantic prototyping system. *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices*, 19(6):213–221, June 1984.

[121] M. Wand. Type inference for objects with instance variables and inheritance. Nov. 1988. Manuscript.

[122] M. Wand. Type inference for record concatenation and multiple inheritance. Oct. 1988. Technical Summary.

[123] D. Warren, L. Pereira, and F. Pereira. Prolog — the language and its implementation compared with LISP. *SIGPLAN Notices*, 12(8):109–115, 1977.