Repairing Type Errors in Functional Programs

Bruce James McAdam

Doctor of Philosophy
Laboratory for Foundations of Computer Science
Division of Informatics
University of Edinburgh
2002

Abstract

Type systems for programming languages can be used by compilers to reject programs which are found to be potentially unsound and which may, therefore, fail to execute successfully. When a program is rejected the programmer must repair it so that it can be type-checked correctly and then executed safely. Diagnostic error messages are essential to help the programmer repair the program.

Hindley-Milner type systems give the programmer a great deal of flexibility (polymorphism and implicit typing) while still ensuring type safety. As a consequence of this flexibility repairing mistakes can be difficult and programmers have previously observed that the type error messages produced by compilers are not helpful enough.

This thesis examines the problem of producing more helpful error messages for ill-typed programs written in programming languages with a Hindley-Milner typing discipline. Three main results are described. Firstly, type inference algorithms which infer types in different orders are described, and the ability of these to produce more meaningful error messages is investigated. Secondly, the results of several other authors on helping explain type inference are condensed into a single generalisation. Thirdly, error messages which suggest concrete changes to the program to remove errors are produced using the theory of linear isomorphisms. This theory is implemented as an extension to the MLj compiler. Finally, extensions to Hindley-Milner are explored, taking the type system of MLj as an example.

Acknowledgements

I would like to thank Stephen Gilmore, my supervisor, for the help and support he has given me while completing this Ph.D.

I am also grateful to Andrew Kennedy and Nick Benton of Microsoft Research who supervised me during a three month internship at their company; and to my examiners, Peter Sestoft and Ian Stark, for their comments on this thesis.

During the first three years of this Ph.D., I was supported my an EPSRC studentship.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Bruce James McAdam*)

Table of Contents

Chapter 1

Introduction

This thesis is an investigation of a problem in the area of programming language *pragmatics*. In order to make programming languages more useful and 'safer', semanticians have devised *type systems* to ensure that certain sorts of programming error are detected before execution. For example type systems can ensure that pointers are not confused with immediate (or unboxed) values and that functions are only applied to valid arguments. This prevents programs from producing erroneous results (e.g. if a pointer is treated as a numeric value then the wrong answer will be obtained in calculations) and also prevents programs from crashing (e.g. if a numeric value is used as an pointer then a memory exception may occur). Well-designed type systems allow the production of high quality software, and can prevent catastrophic failure of software in safety critical applications. The pragmatic issue in question is making it easy to correct programs which the type system rejects.

This introduction starts by looking at a problem which prevents certain type systems gaining wider acceptability (Section 1.1). The aims of the thesis are then laid out (Section 1.2). Finally the structure of the rest of this thesis can be found in Section 1.3.

1.1 The Problem

Most programming language compilers perform several types of semantic analysis on programs before they are translated into machine code. This thesis is about one such analysis: *type inference*. In this analysis the compiler establishes information about the types of data used in the program and checks that the program is acceptable according to the type semantics. This analysis can reject a program as "incorrect", in which case the programmer will need to correct the program and will appreciate some help in doing so. To see what sort of help programmers might appreciate, consider how spell-checkers work. These are among the most useful programs in popular use today. They can point out where you made spelling mistakes, and suggest how to correct them. A spell-checker would be neither useful nor popular if it only pointed out that you had made a mistake without suggesting how to correct it, or if it gave information which looked at the problem from the perspective of how it worked rather than from the writer's perspective, e.g.

```
rype-checker
  ^
Spelling error at character 3.
Cannot follow "ry" with "p"
```

This however is the style of response produced for programmers by compilers.

The error messages typically produced are based on how the *process* of type inference and type checking failed, and not on either what the mistake is or how it can be corrected. These messages can be used to repair mistakes but not as easily as might be possible. Informal discussion with people who have learnt to program with strongly typed languages and then entered industry to program with other languages suggests that the difficulties in repairing errors is seen as "more trouble than it is worth," and this may explain why such systems have not been more widely adopted. The proliferation of papers on the topic and the intensity of discussion at conferences suggests that experienced programmers also have difficulty with error messages. It is therefore important to investigate this topic both to help existing users and encourage wider adoption of strong type systems.

1.2 Aims

The general aim of this thesis is to investigate forms of information which can help programmers repair type errors in their programs. Algorithms for producing these forms of information are given. Also of interest is the ease or difficulty of integrating these new algorithms with existing compilers so that the work can be put to use.

1.2.1 Type System for Investigation

As stated earlier, this thesis is about pragmatic aspects of type systems. It is important though to consider not only how the systems are used and how programmers work, but also to consider the semantics of the type system. For this reason the type system to be investigated is the Hindley-Milner type system. This has a number of features which make it attractive to this work.

- The type system and its type inference algorithms have been proven to be sound. There is a large body of theory relating to the type system.
- Type inference is decidable, and in practice takes a reasonable length of time (in contrast to stronger dependent type systems).
- Many desirable features are exhibited by the system. These include polymorphism, implicit typing (and type inference) and higher order functions.
- Programs for this type system can often be completely devoid of type annotations (though the syntax of immediate values may reveal their type, for example in SML `1` is an integer, whereas `1.0` is real). This makes the problem of producing error messages harder as there is less information to guide this process.
- The type system is used in several multipurpose programming languages with extensive libraries and support. These have been used to write demanding applications such as compilers and theorem provers, and also parallel and distributed programs on a wide variety of platforms, low-level operating-system oriented programs and graphical applications.
- The system has yet to gain wide acceptability in the programming community (compared, for example, with the success of object oriented languages) and this may be due in part to the difficulties of fixing type errors.
- The system has influenced the design of programming languages which do not use it directly.

1.2.2 Inclusion of Proofs

It has not been considered necessary to rigorously prove all of the propositions in this thesis. This is because the aim of the thesis is to look at ways of helping people, rather than to create a new body of theory. Even if an algorithm is incomplete in some way, or even unsound for a small subset of its domain, in most cases it is still likely to be of help to programmers. That said, the work is grounded in theory and attempts have been made to state the unproven theorems which algorithms should satisfy in order to be useful, and the possible forms of proofs of these have been proposed.

1.2.3 Implementation

All of the algorithms in this thesis have been implemented in some way, whether a ‘toy’ implementation or in a full compiler. This gives some assurance of correctness in the absence of a proof. Examples of the output of these are provided as a way of letting the reader gauge whether or not they are likely to be of use and deserving of further testing with real users.

1.2.4 Testing

The aim of this thesis has not been to produce a production implementation of any algorithm. The intention has been to devise forms of information which are likely to help programmers, and to justify these logically. Details of the user interface have not been considered, and hence there has not been any testing with users. Before integrating any of the proposals in this thesis into a product, user testing to establish the best way to present the information produced should take place. We do however know that all the methods in this thesis can be used to help repair mistakes in programs (for logical reasons).

1.3 Structure of Thesis

The introductory part of this thesis includes a look at the current state of technology in type inference and the error messages produced by compilers in current use (Chapter 2). It then looks at related work in improving error messages (Chapter 3).

There are three chapters containing the main findings. The first of these, Chapter 4, looks at alternative type inference algorithms which look at programs in different orders in order to produce different error messages.

Chapter 5 introduces a data structure which can record information about the types in programs, whether or not the programs are correctly typed. This data structure can be used to extract different forms of useful information.

The third chapter of findings (Chapter 6) is about generating error messages which suggest ways of repairing programs. This is analogous to the example of a spell checker at the beginning of this introduction. The chapter is based on the theory of type isomorphisms.

After these main findings on Hindley-Milner types systems are described, Chapter 7, looks at an extension of this with subtyping. MLj integrates Standard ML with the class hierarchies of Java. This introduces new forms of type error. Ways of dealing with these errors are proposed and discussed (without any implementation).

In keeping with the spirit of this thesis, as a bridge between theory and practice, following the conclusions in Chapter 8 there are two appendices. One has proofs of theorems, and the other has extracts from an implementation.

Chapter 2

Current Technology

Before looking at new ways to improve type error messages and help programmers debug their mistakes, we will look at the advice produced by current compilers. First we will describe the standards against which error messages should be compared (Section 2.1). Then messages from popular compilers will be shown and compared against the standards (Section 2.2). Lastly, the algorithms used to produce these messages will be given and examined (Section 2.3).

2.1 Guidelines for Error Messages

In a recent article on error messages for the World-Wide-Web, Jakob Nielsen claims that "the guidelines for creating effective error messages have been the same for 20 years" [Nie01] and that the attributes that good error messages must have are to be

Explicit in their statement that an error has occurred, and in the stating the consequences of the error.

Human-readable so that they can be understood by their intended reader and do not have the feel of being produced for other software to read.

Polite particularly in not implying that the user is solely at fault and in not implying that the user is stupid.

Precise in their statement of the problem and in any additional advice they give.

Constructive by giving advice about how the problem can be rectified.

For type errors these guidelines have several consequences. It should be stated explicitly that a type error has prevented compilation, in contrast to warning messages which do not prevent compilation (e.g. a warning that pattern matching is incomplete). A clear statement that the program has not been recompiled will prevent programmers in an edit-recompile-test cycle from going on to test the dynamic behaviour without realising that their latest change is not reflected in the compiled program.

The benefits of making error messages readable by humans should be clear. In particular, for type error messages this means not referring to the internal representation of types and syntax. Errors should also be given names rather than abstract identifiers ("Type Error" rather than "Error Number 8"). Politeness refers more to the phrasing of the message than its content. Precision demands that extra information is given, e.g. where did the types come from, where in the program was the error found. These three aspects of messages — readability, precision and politeness — are highly dependant on the intended user of the compiler, for example the theorem proving community are typically familiar with type inference algorithms and will understand terms such as "occurs error", whereas in education it cannot even be taken for granted that the programmer understands that type inference takes place.

Giving constructive advice is the most demanding of the guidelines. Nielson states that for a "Page not found" error on the World-Wide-Web this could include advice such as "check the spelling of the URL", or even automatically suggest similarly spelt valid URLs. By analogy this means that for a type checker the messages could suggest a particular change to the program or make a generic suggestion such as "check that the function is curried," or "check that real and integer arithmetic are not mixed."

2.2 Current Error Messages

In this section we will look at a selection of error messages from a selection of compilers and see whether they address the guidelines described above.

2.2.1 Three Compilers

Consider this program which has a type error (in Standard ML [MTHM97])

```
map ([1, 2, 3], Int.toString)
```

The problem is that `map` has type `('a -> 'b) -> 'a list -> 'b list` — which means that it takes a function and then a list as curried arguments — but has been supplied with its arguments in uncurried form and in the wrong order. It is difficult for current compilers to explain the problem because of the curried type and the polymorphism. Here we will see how different compilers react to this program. The compilers shown are all currently in popular use.

Moscow ML[1] Following is the transcript of an interactive session including the program above (The programmer's input follows the "-" prompt.)

```
Moscow ML version 2.00 (June 2000)
Enter `quit();' to quit.
- map ([1, 2, 3], Int.toString);
! Toplevel input:
! map ([1, 2, 3], Int.toString);
! ^^^
! Type clash: expression of type
!   ('a -> 'b) -> 'a list -> 'b list
! cannot have type
!   'c * 'd -> 'a list -> 'b list
-
```

This error message suffers from several deficiencies

- It does not explicitly state that compilation was not possible (this is indicated by "Type clash", but some users may not understand this expression).
- The message does not make it clear what the two types given refer to. It should be precise and state that the first is the inferred type of the function, and the second is implied by its context.
- There is no constructive advice about how to repair the problem.

The error message's strengths lie in being human readable: it has a neatly phrased description and highlights the expression containing the problem.

[1] Moscow ML version 2.00, available from `http://www.dina.kvl.dk/~sestoft/mosml.html`.

Standard ML of New Jersey[1] Here is a transcript from an interactive session.

```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000
- map ([1, 2, 3], Int.toString);
stdIn:17.1-17.30 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z -> 'Y
  operand:         int list * (int -> string)
  in expression:
    map (1 :: 2 :: <exp> :: <exp>,Int.toString)
-
```

The types listed in this message are different from those in the Moscow ML message because a different inference algorithm is used, but the problems are similar

- The message lacks human readability by quoting the internal representation of the list rather than the original syntax.
- It is unclear where the "operator domain" came from, and it is possible that some programmers will not understand this term.

Each of these compilers has a different type inference algorithm, but the output from each could be improved. A new problem can be seen by comparing the two compilers. Their error messages are sufficiently different that a programmer used to one platform could have difficulty using the other. For example Moscow ML quotes the type of the function, whereas New Jersey quotes the type of its domain. A Moscow ML user failing to read the message correctly might think that New Jersey ML is claiming that `map` has the type `'Z -> 'Y`.

MLj[2] This compiler does not have interactive sessions. The following file was used to test it

```
structure Test = struct
  val _ = map ([1, 2, 3], Int.toString)
end
```

And this message was reported

[1] Standard ML of New Jersey version 110.0.7. Copyright Lucent Technologies. Available from http://cm.bell-labs.com/cm/what/smlnj/index.html.

[2] MLj Version 0.2, Copyright Persimmon IT Inc.. Available from http://www.dcs.ed.ac.uk/home/mlj/.

```
  Error at 2.31-56:type mismatch
    argument type: 'ac list*(int->string)
[for 'ac={Int8.int,int,Int64.int,Int16.int}]
    expected: 'ab->'aa
```

This message looks different from the previous two.

- The message lacks human-readability by not quoting the relevant source code. This should be particularly important in MLj as the source code is not already visible (because it is a batch compiler rather than interactive). The location "2.31-56" is designed to be read by other software (e.g. the sml-mode for the text editor Emacs) rather than a person.
- Readability is further reduced by showing the internal MLj representation of types which includes an idiosyncratic representation for the types of numbers (to deal with overloading). This is unlikely to be understood by many ML programmers.
- Again, it is not clear how the quoted types were obtained.

2.2.2 Location and Cascades

One characteristic of type error messages is that a small mistake early in the program can lead to a large number of error messages later on. Consider this program

```
val one = "1"
val two = one + one
val three = two + one
val four = three + one
val five = four + one
```

New Jersey SML reports

```
stdIn:18.17 Error: overloaded variable not defined at type
  symbol: +
  type: string
stdIn:19.19 Error: overloaded variable not defined at type
  symbol: +
```

```
  type: string
stdIn:20.20 Error: overloaded variable not defined at type
  symbol: +
  type: string
stdIn:21.19 Error: overloaded variable not defined at type
  symbol: +
  type: string
```

The simple mistake in the first line (using a string instead of an integer) has caused four identical messages.

As well as producing several mistakes for one mistake, none of the error messages produced actually refer to the mistake: they all have the wrong location.

Moscow ML avoids the problem of cascades by announcing only one error and then stopping type inference

```
! Toplevel input:
!   val two = one + one
!                 ^
! Overloaded + cannot be applied to argument(s) of type string
```

2.3 Current Algorithms

To understand why current error messages have the characteristics they have, let us have a look at two type inference algorithms. These will be given for the simple language in Figure 2.1 in which expressions have the types in Figure 2.2 as given by the semantics in Figure 2.3 (in the style of [MTHM97]). Note that tuple types are written with angle brackets, $\langle \ldots \rangle$, this allows us to have unary tuples and a unit type.

Type inference requires a type environment, Γ, which is a finite map from identifiers to type schemes. Γ can have mappings removed from it (Γ_x is Γ with any term for x removed) and can be augmented with new mappings, for example after a declaration. $\Gamma + (x : \sigma)$ is the same as $\Gamma_x \cup \{x : \sigma\}$. $\Gamma(x)$ is any σ such that $(x : \sigma) \in \Gamma$.

Type schemes are obtained from types by *closing* a type under a type environment. $\overline{\Gamma}(\tau)$ (the closure of τ under Γ) is the type scheme $\forall \alpha_1 \cdots \alpha_n . \tau$ where $\alpha_1 \cdots \alpha_n$ are the type variables occurring in τ but which do not occur free in Γ. In particular, closing a type under a type environment

Figure 2.1 An ML-like language.

$$
\begin{array}{rcll}
e & ::= & x & \text{identifiers} \\
 & \mid & \lambda p.e & \lambda\text{-abstraction} \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{let-declaration} \\
 & \mid & e_0\ e_1 & \text{function application} \\
 & \mid & \langle e_1, \ldots, e_n \rangle & \text{tuple expression} \\
p & ::= & x & \text{identifier patterns} \\
 & \mid & \langle p_1, \ldots, p_n \rangle & \text{tuple patterns}
\end{array}
$$

Figure 2.2 Types, type schemes, environments and substitutions.

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{type variable} \\
 & \mid & (\tau_1, \ldots, \tau_n)c & \text{type from constructor } c \\
 & \mid & \tau \to \tau' & \text{function type} \\
 & \mid & \langle \tau_1 \times \ldots \times \tau_n \rangle & \text{tuple type} \\
\sigma & ::= & \forall \alpha_1 \ldots \alpha_n . \tau & \text{type scheme} \\
\Gamma & ::= & \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} & \text{environment} \\
S & ::= & \{\alpha_1 \mapsto \tau_1, \ldots \alpha_n \mapsto \tau_n\} & \text{substitution}
\end{array}
$$

with no free type variables results in every type variable in the type being universally quantified. (Some papers denote closure with $clos_\Gamma(\tau)$.)

If some of the bound type variables in a scheme, σ, can be instantiated to give σ' then $\sigma > \sigma'$.

The language lacks several of the features of Standard ML

- Module system
- Explicit type annotations and constraints
- Overloading
- Recursive declarations

Though these features are relevant to producing error messages, they will be omitted as they differ from language to language. The core features in the figures are consistent across a range

Figure 2.3 Type semantics.

First for expressions, $\Gamma \vdash e : \tau$.

$$\frac{\Gamma(x) > \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_0 : \tau' \to \tau \qquad \Gamma \vdash e_1 : \tau'}{\Gamma \vdash e_0 e_1 : \tau}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1 \times \ldots \times \tau_n \rangle}\ \textsc{Tup}$$

$$\frac{\Gamma \vdash p : (\Gamma', \tau_0) \quad \Gamma' \vdash e : \tau_1}{\Gamma \vdash \lambda p.e : \tau_0 \to \tau_1}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma + (x : \overline{\Gamma}(\tau_0)) \vdash e_1 : \tau_1}{\Gamma \vdash \ \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 : \tau_1}\ \textsc{Let}$$

The following are for patterns, $\Gamma \vdash p : (\Gamma', \tau)$ (Γ' describes the identifiers in the pattern).

$$\frac{}{\Gamma \vdash x : (\Gamma + (x : \tau), \tau)}\ \textsc{VarPat}$$

$$\frac{\Gamma_0 \vdash p_1 : (\Gamma_1, \tau_1) \quad \Gamma_1 \vdash p_2 : (\Gamma_2, \tau_2) \quad \ldots \quad \Gamma_{n-1} \vdash p_n : (\Gamma_n, \tau_n)}{\Gamma_0 \vdash \langle p_1, \ldots, p_n \rangle : (\Gamma_n, \langle \tau_1 \times \ldots \times \tau_n \rangle)}\ \textsc{TupPat}$$

of languages including Standard ML, MLj, O'Caml, Haskell.

2.3.1 Substitutions and Unification

In addition to the types, type schemes and type environments seen in the semantics, type inference algorithms make extensive use of *substitutions*. A substitution is a map with finite support from type variables to types. Substitutions are denoted by a set of mappings, $\{\alpha_1 \mapsto \tau_1, \cdots \alpha_n \mapsto \tau_n\}$. A substitution represents a means of refining types. If we know that a certain type (containing type variables) is associated with an expression, and that a substitution is also associated with it then we can apply the substitution to the type to refine it.

The set of type variables which a substitution acts on is called its support, $\mathrm{supp}(S) = \{\alpha : \alpha \neq S\alpha\}$. Substitutions also have free variables, $FV(S) = \{\alpha : \exists \beta \in \mathrm{supp}(S).\alpha \in FV(S\beta)\}$.

Substitutions can be composed. $S_1 S_0$ is the substitution which has the same effect as applying first S_0 and then S_1.

All substitutions in type inference are idempotent, i.e. $SS\tau = S\tau$. This is achieved by ensuring $\mathrm{supp}(S) \cap FV(S) = \{\}$. This restriction prevents us from getting 'infinite' or recursive types

(types which contain themselves). Some languages allow a limited form of recursive type (which will not be discussed in this thesis). S_1S_0 is idempotent iff both components are idempotent and $FV(S_1) \cap \text{supp}(S_0) = \{\}$.

We define an ordering on types: $\tau > \tau'$ iff $\exists S : S\tau = \tau'$. We say that if $\tau > \tau'$ then τ' is a substitution instance of τ. Also, a type environment, Γ, has a substitution instance, Γ', iff $\exists S : S\Gamma = \Gamma'$ (where $S\Gamma$ satisfies $\forall x : (S\Gamma)(x) = S(\Gamma(x))$).

Unification, function U, returns the most general substitution which when applied to each of its argument types will produce the same type. For example $U(\mathsf{int} \rightarrow \alpha,\ \beta \rightarrow \mathsf{real}) = \{\alpha \mapsto \mathsf{real},\ \beta \mapsto \mathsf{int}\}$. A survey of applications and techniques for unification can be found in [Kni89]. Type inference fails if no unifier exists. Type inference fails (and mistakes are detected) because of a failure to unify. Implementations then produce error messages indicating a problem with the subexpression of current interest and mention the un-unifiable types.

In most compilers implemented in languages with imperative features (such as Standard ML), instead of storing explicit substitutions, a type variable, α, is represented as a reference which can refer to a type, τ, to implement the substitution term $\alpha \mapsto \tau$. The main motivation for this is speed, especially in generating elaborated syntax trees containing types. It has been suggested [PW93] that this style of implementation is more efficient for large projects (though [Tof89] suggests that explicit substitutions are equally efficient for smaller implementations). There do not appear to have been any studies of whether using explicit substitutions will cause any appreciable slow-down on today's computers.

2.3.2 Two Algorithms for Type Inference

The two most common type inference algorithms are W: the classic bottom-up algorithm [DM82] and M, a top-down algorithm in use in Moscow ML and Objective Caml [LY98]. Bottom-up inference (Figure 2.4) answers the question "What type does expression e have in environment Γ, and what restrictions must be made to the types in the environment for this to be the case?"(It takes an expression and environment and returns a type and substitution of types for type variables in the environment). Top-down type inference (Figure 2.5) answers the question "What restriction must be made to type variables for expression e to have type τ in environment Γ?" (It takes an expression, environment and target type and returns a substitution).

W satisfies two theorems, which are proved in [Dam85], similar theorems can be shown for M [LY98].

Theorem 1 (Soundness of W) *If $W(\Gamma, e)$ succeeds with (S, τ) then there is a derivation of $S\Gamma \vdash e : \tau$.*

Theorem 2 (Completeness of W) *Given (Γ, e) let Γ' be an instance of Γ and η be a type scheme such that $\Gamma' \vdash e : \eta$. Then*

1. *$W(\Gamma, e)$ succeeds*
2. *If $W(\Gamma, e) = (P, \pi)$ then for some R: $\Gamma' = RP\Gamma$, and η is a generic instance of $R\overline{P\Gamma}(\pi)$.*

Theorem 3 (Soundness of M) *Let e be an expression and Γ be a type environment. If there exists a type τ such that $M(\Gamma, e, \tau) = S$ then $S\Gamma \vdash e : S\tau$.*

Theorem 4 (Completeness of M) *Let e be an expression, Γ be a type environment. If there exists a type τ and a substitution P such that $S\Gamma \vdash e : P\tau$ then $M(\Gamma, e, \tau) = S$ is defined, and there exists a substitution R such that $P|_{new} = (RS)|_{new}$ where new is the set of variables used by $M(\Gamma, e, \tau)$ and $|_V$ is defined by $S|_V = \{\alpha \mapsto \tau \in S : \alpha \notin V\}$.*

Hybrid algorithms in which some recursive calls look like those of W and others look like those of M are also possible [LY00].

2.3.3 Generating Messages in W and M

A type error is typically announced when unification fails. This occurs in W at application expressions, tuples and some other syntactic forms not in the language shown here (such as modules with signature constraints). In M unification can fail at λ-expressions, variables, tuples and, as with W, at other expressions present in real languages but not shown here which must have a particular form of type.

The error messages shown from real compilers all announce the expression being inspected when unification failed, and the types (or parts of them) which failed to unify. Variations occur because the types can be quoted before or after the attempted unification.

Type inference can continue after unification has failed by giving the expression being examined a ‘best guess’ type (such as a new type variable). This allows many type error messages to be produced, which is useful if the programmer is able to fix several of them before recompiling. It has the disadvantage that one mistake may cause a cascade of error messages, which can confuse the programmer.

Figure 2.4 Bottom-up algorithm W.

$W(e,\Gamma) = (S,\tau)$ if $S\Gamma \vdash e : \tau$.

$$
\begin{array}{rcl}
W(\Gamma, x) & = & \textbf{let } \forall\alpha_1\cdots\alpha_n.\tau = \Gamma(x) \\
& & \textbf{in } (\{\}, \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}\tau) \text{ (for new } \beta_1\cdots\beta_n) \\
W(\Gamma, e_0 e_1) & = & \textbf{let} \\
& & \quad (S_0, \tau_0) = W(\Gamma, e_0) \\
& & \quad (S_1, \tau_1) = W(S_0\Gamma, e_1) \\
& & \quad \tau_0' = S_1\tau_0 \\
& & \quad V = U(\tau_0', \tau_1 \to \beta) \text{ (for new } \beta) \\
& & \textbf{in } (V S_1 S_0, V\beta) \\
W(\Gamma, \lambda x.e) & = & \textbf{let} \\
& & \quad (\Gamma', \tau) = W_{pat}(\Gamma, p) \\
& & \quad (S, \tau') = W(\Gamma', e) \\
& & \textbf{in } (S, S(\tau \to \tau')) \\
W(\Gamma, \langle e_1, \ldots, e_n\rangle) & = & \textbf{let} \\
& & \quad (S_1, \tau_1) = W(\Gamma, e_1) \\
& & \quad \vdots \\
& & \quad (S_n, \tau_n) = W(S_{n-1}\ldots S_1\Gamma, e_n) \\
& & \quad S = S_n \ldots S_1 \\
& & \textbf{in } (S, \langle S\tau_1 \times \ldots S\tau_n\rangle) \\
W(\Gamma, \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1) & = & \textbf{let} \\
& & \quad (S_0, \tau_0) = W(\Gamma, e_0) \\
& & \quad (S_1, \tau_1) = W((S_0\Gamma) + (x : \overline{S_0\Gamma}(\tau_0)), e_1) \\
& & \textbf{in } (S_1 S_0, \tau_1)
\end{array}
$$

$$
\begin{array}{rcl}
W_{pat}(\Gamma, x) & = & (\Gamma + (x : \beta), \beta) \text{ (for new } \beta) \\
W_{pat}(\Gamma, \langle p_1, \ldots, p_n\rangle) & = & \textbf{let} \\
& & \quad (\Gamma_1, \tau_1) = W_{pat}(\Gamma, p_1) \\
& & \quad \vdots \\
& & \quad (\Gamma_n, \tau_n) = W_{pat}(\Gamma_{n-1}, p_n) \\
& & \textbf{in } (\Gamma_n, \langle \tau_1 \times \ldots \times \tau_n\rangle)
\end{array}
$$

Figure 2.5 Top-down algorithm M.

$M(\Gamma, e, \tau) = S$ if $S\Gamma \vdash e : S\tau$.

$$
\begin{array}{rcl}
M(\Gamma, x, \tau) & = & \textbf{let } \forall \alpha_1 \cdots \alpha_n . \tau' = \Gamma(x) \\
 & & \textbf{in } U(\{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}\tau', \tau) \text{ (for new } \beta\text{s)} \\
M(\Gamma, e_0 e_1, \tau) & = & \textbf{let} \\
 & & \quad S_0 = M(\Gamma, e_0, \beta \to \tau) \text{ (for new } \beta\text{s)} \\
 & & \quad S_1 = M(S_0\Gamma, e_1, S_0\beta) \\
 & & \textbf{in } S_1 S_0 \\
M(\Gamma, \lambda p.e, \tau) & = & \textbf{let} \\
 & & \quad S_0 = U(\tau, \beta \to \beta') \text{ (for new } \beta\text{s)} \\
 & & \quad (\Gamma', S_1) = M_{pat}(S_0\Gamma, p) \\
 & & \quad S_2 = M(\Gamma', e, S_1 S_0 \beta') \\
 & & \textbf{in } S_2 S_1 S_0 \\
M(\Gamma, \langle e_1, \ldots e_n \rangle, \tau) & = & \textbf{let} \\
 & & \quad S = U(\tau, \langle \beta_1 \times \ldots \times \beta_n \rangle) \text{ (for new } \beta\text{s)} \\
 & & \quad S_1 = M(S\Gamma, e_1, S\beta_1) \\
 & & \quad \vdots \\
 & & \quad S_n = M(S_{n-1} \ldots S_1 S, e_n, S_{n-1} \ldots S_1 S \beta_n) \\
 & & \textbf{in } S_n \ldots S_1 S \\
M(\Gamma, \textsf{let } x = e_0 \textsf{ in } e_1, \tau) & = & \textbf{let} \\
 & & \quad S_0 = M(\Gamma, e_0, \beta) \text{ (for new } \beta) \\
 & & \quad S_1 = M((S_0\Gamma) + (x : \overline{(S_0\Gamma)}(S_0\beta)), e_2, S_0\tau) \\
 & & \textbf{in } S_1 S_0
\end{array}
$$

$$
\begin{array}{rcl}
M_{pat}(x, \Gamma, \tau) & = & (\Gamma + (x : \tau), \{\}) \\
M_{pat}(\langle p_1, \ldots, p_n \rangle, \Gamma, \tau) & = & \textbf{let} \\
 & & \quad S = U(\tau, \langle \beta_1, \ldots, \beta_n \rangle) \text{ for new } \beta\text{s}) \\
 & & \quad (\Gamma_1, S_1) = M_{pat}(p_1, \Gamma, S\beta_1) \\
 & & \quad \vdots \\
 & & \quad (\Gamma_n, S_n) = M_{pat}(p_n, \Gamma_{n-1}, S_{n-1} \ldots S_1 S \beta_1) \\
 & & \textbf{in } (\Gamma_n, S_n \ldots S_1 S)
\end{array}
$$

Chapter 3

Related Work

There have been a number of works relating to type errors and helping programmers use types in their programs. These have largely operated in one of two ways: explaining how a type was inferred (Section 3.1) or suggesting a location where a mistake may have been made (Section 3.2). Also relevant to this thesis are interactive program development systems and user interfaces which offer some way to avoid or repair type errors (Section 3.3). In this chapter I also include a brief survey of other important works which provide background information to the thesis on type systems and type checking algorithms (Section 3.4).

3.1 Explaining Type Inference

One popular proposal for helping programmers use types is to offer a facility to explain how types were derived.

Helen Soosaipillai has produced an "explanation based" type checker for a subset of Standard ML [Soo90]. This can explain how types were derived (it works for correctly typed programs, not for type errors). The programmer operates it by asking "why" types were derived. For example if the compiler says that 1+2 is an int, the programmer can ask "why?" and be told that it is because 1 is an int, 2 is an int and + has type int $\times$ int $\rightarrow$ int. This design has not been integrated into any compiler.

Mike Beaven and Ryan Stansifer [BS93] describe a similar system, except that it explains type errors rather than correctly typed programs. Again the programmer can ask "why?" particular types were derived to get more information. The system has been implemented but not integrated into any compiler.

Dominic Duggan and Frederick Bent [DB96] devised a unification algorithm which collected explanations of why types were derived. The explanations take the form of a collection of locations in the program which were used to infer the type of some subexpression. This system has been added to a version of the Standard ML of New Jersey Compiler called SML/E (standing for "explanation")[1]. Dominic Duggan has also produced a definition for "correct type explanations" [Dug98], which is examined in more detail in Section 5.4.3.

Olaf Chitil too, has looked at a formal presentation of type explanations and has proposed that these be based on inference trees of principal typings [Chi01]. Based on this notion of type explanation he has implemented an algorithmic debugger [Sha83] to track down the location of mistakes in Haskell programs. This operates by asking programmers questions such as `reverse :: [b] -> [c]  Is intended type an instance?` when it believes that the expression should have that type. After a series of questions it will announce a source for the type discrepancies. The system is impressive to watch in action but seems to ask more questions than should be necessary. The example in Chitil's paper uses nine questions to track down the mistake in a five line program containing less than forty syntactic tokens. Other systems (described in the next section) can already propose a small set of possible locations for mistakes in a program. If the number of possible locations for a mistake is n then there is a set of n questions (possibly as few as $\lceil \lg n \rceil$ questions) which can distinguish the actual location. Chitil's system appears to exceed this. It has also been shown that questions about whether a particular expressions should have an instance of a particular type are hard to answer mentally [YM97].

My own introduction to the topic came from my supervisor, Stephen Gilmore's, paper [Gil95] which suggested annotating types with identifiers which were used to infer them. The resultant annotated types were called wide-types or deep-types. I extended this work by creating big-types [McA97] which incorporated the information in both wide- and deep-types.

3.2 Finding the Location of Mistakes

An orthogonal approach to explaining why a type could or could not be inferred is to suggest the location in the program of the expression where the programmer has made a mistake.

[1]SML/E is available from `http://guinness.cs.stevens-tech.edu/~dduggan/Public/Smle/index.html`

Gregory Johnson and Janet Walz dealt with this problem for ML by looking at unification [JW86]. They base their solution upon looking for maximum flow in graphs representing unification and aim to find the single anomaly (mistake) which prevents type checking and to avoid producing extra error messages by instantiating unknown types with the type they are most likely to be.

Mitchell Wand [Wan86] also modified unification so that it could keep track of which pieces of code were used to derive types and announce the two which were inconsistent.

Mikael Rittri advocates an interactive approach to finding the source of type errors [Rit93a]. He proposes a specification for such a system but this has not been implemented.

Oukseh Lee and Kwangkeun Yi proved that M will always fail sooner than W (it looks at less of the program to find an inconsistency) [LY98]. This means that it has a greater chance of announcing the location where a mistake was actually made.

3.3 Changes to the User Interface

Laurence Rideau and Laurent Thérey [RT97] have written an interactive programming environment for SML and CamlLight. This includes a syntax editor which type-checks programs as they are written. Built into the type checker is Wand-style error location, and Duggan-style type explanation. These are integrated into the editor so that the relevant locations in the program can be highlighted.

Jonathan Whittle has also created an editor for SML with an integrated type checker [WBL97, Whi98]. This editor is based on the principal of *proofs as programs*, in order to type check a program a proof is devised to prove that it correctly types. If a proof cannot be found then there is a type error. Programs are written by analogy — users start with a simple program (perhaps the map function) and transform it into the program they require (perhaps the foldleft function). Each stage in the transformation should result in a valid program. If type checking fails after any transformation then the new additions are highlighted as a possible source for the mistake. The system is only designed for teaching, it does not support the full SML language and it does not reflect the editing strategies used by experienced programmers. In order to investigate what sort of help would be useful, Whittle conducted a survey of the mistakes made by students learning to program in SML [Whi96]. The conclusion of this survey was that syntax errors had more effect than type errors. This is why programs are authored by transforming a syntactically valid program using steps guaranteed to preserve syntactic validity. Hence, Whittle's work is not of such direct relevance to this work.

Karen Bernstein and Eugene Stark presented a modified version of algorithm W which allows the type checking of open expressions with unbound variables [BS95]. Their idea is that programmers wanting to find out about their program can find out what types an expression required its environment to have. This has not yet been implemented in any widely available system.

Yang Jun's interest is the visualisation of types as graphical symbols, the psychology of how people manually check types, and finding the location where a mistake has been made [Yan97, YM97, Yan99, YMT00, YMT01, Yan01]. An experimental comparison of graphical means of displaying types with traditional textual representations showed that the graphical representation did not help groups of programmers or learners solve problems involving types.

There have been other more minor works on changing the user interface of type inference, for example David A. Turner completed an undergraduate project which modified the error messages produced by the ML-Kit compiler [Tur90]. The changes were to the text of error messages (rather than highlighting different syntax or announcing different types). For example he produced a specific message if the condition of an 'if' expression was not a boolean.

3.4 Background Reading

The background information used in this thesis includes information about type systems, programming languages and their compilers, user interfaces for compilers and unification.

3.4.1 Type Systems

The Hindley-Milner type system and inference algorithm W (described in Chapter 2) are described by Robin Milner and Luis Damas[Mil78, DM82]. More information on these can be found in Luca Cardelli's writings about the Hindley-Milner system [Car87] and a survey of type systems in general [Car97]. Oukseh Lee and Kwangkeun Yi proved that algorithm M is correct [LY98] and that hybrid top-down and bottom-up algorithms are possible [LY00].

Trevor Jim [Jim95] has written on the properties of principal typings as opposed to principal types (a typing is an assertion of the form $\Gamma \vdash e : \tau$). Jim's work influenced Bernstein and Stark. Jim claims that using a type system with principal typings can produce more accurate error messages because the type of the use of an identifier can be inferred independently of its definition.

Nikolaj Skallerud Bjørner [Bjø94] has looked at producing minimal typing derivations (finding the most specific types that identifiers can have, in contrast to the usual most general types). Again this type could be of use to programmers wishing to understand the types in their program by removing unnecessary polymorphic types.

Xavier Leroy has tackled the problem of typing references and continuations [Ler93]. Damas also dealt with this problem [Dam85]. To show the difficulty with with polymorphic references, consider for example

```
val r = ref []
fun f r = (r := [1, 2, 3])
val p = (f r, r)
```

From the first line, it looks like `r` should have the polymorphic type scheme ∀`'a.'a list ref` (according to the semantics in Figure 2.3. In the last line, however the pair cannot have type scheme ∀`'a.unit * 'a list ref` because `r` is now a `int list ref`. There are obvious issues in explaining types in programs like this, and repairing the mistakes that may come from misunderstanding. This topic, however, is not dealt with in this thesis because if the incompatible approaches taken in different languages (*cf.* the differences between SML in 1990 and 1997 [MTH90, MTHM97]).

Roberto Di Cosmo has written a book on type isomorphisms [Di 95]. These relations form the basis of Chapter 6.

3.4.2 Compilers and Languages

The examples of type errors in this thesis are based on Standard ML, and implementations are also in this language. Standard ML has a formal definition which can be found in [MTH90] or [MTHM97], and an accompanying commentary in [MT91]. Andrew Appel has written a critique of the language [App93] and Stefan Kahrs has highlighted some mistakes and ambiguities in the 1990 definition [Kah93]. Textbooks and introductions to SML include [Tof89, FF98, Pau96, Gil97].

The original version of the ML-Kit[1] was an interpreter based directly on the semantics. This

[1]The ML-Kit is available from `http://www.it-c.dk/research/mlkit/`.

has been documented [BRTT93].

MLj is an extension of the SML language (and a compiler for the extended language) which offers integration with Java[1]. Information about the implementation and semantics of MLj can be found in [BKR98, BK99, MKB00].

Another variant of ML is Caml, information on which can be found in [Ler97, LRVD99][2].

Apart from the variants of ML, Haskell is a Hindley-Milner based language[3]. This language differs from ML in that it is lazy and pure. This creates very slightly different requirements for type errors (for example there is no need to consider imperative features) but everything discussed in this thesis should be relevant to Haskell. Haskell's type system is documented in [PH98, HHPW94].

3.4.3 User Interfaces

Norman Ramsey has discussed a method to prevent printing too many error messages [Ram97] which could be applied to type errors. There has been a conference on User Interfaces for Theorem Provers which has also dealt with type checking [UIT96].

3.4.4 Unification

We have seen that unification is critical to type inference. A survey of unification algorithms and applications has been produced by Kevin Knight [Kni89]. As well as equality unification (producing substitutions to make terms equal), this thesis deals with unification modulo isomorphism (Chapter 6). I was alerted to this in part by Mikael Rittri's work on library searches [Rit93b]. Algorithms for isomorphic and associative commutative unification can be found in [NPS93, Sti75, Sti81, LC89].

[1] MLj is available from `http://www.dcs.ed.ac.uk/home/mlj/`.
[2] Objective Caml is available from `http://caml.inria.fr/ocaml/`.
[3] Implementations of Haskell are available from `http://www.haskell.org/`.

Chapter 4

Order of Type Inference

We have seen in Chapter 2 that the order of type inference is important to the error messages produced (witness the difference between bottom-up and top-down implementations). This chapter gives alternative inference algorithms which type check in different orders and examples of how they make type inference fail at different points (i.e. at different expressions and when different types fail to unify) and can produce more pleasing error messages. First a new algorithm to make type inference of applications symmetric is presented in Section 4.1, and then other syntactic forms are examined in Section 4.2.

4.1 Symmetric type inference[1]

Critical to the operation of type inference algorithms is their use of substitutions. We will see that the way in which substitutions are applied in type inference algorithm W means that errors are detected towards the right-hand side of expressions. This section introduces a new operation — unification of substitutions — which allows greater control over the use of substitutions so that this bias can be removed.

We will see in Section 4.1.1 that one complaint about type errors is that the part of the program highlighted by the compiler is generally not the part of the program in which the programmer made the mistake. We then examine the type inference algorithm and see that it has a *left-to-right bias* towards detecting problems late in the code and this bias is caused by the way unification and substitutions are used.

[1] This section is based on material in the published paper [McA98a] and the technical report [McA98b]

The solution to this problem with the conventional inference algorithm is a new type inference algorithm designed from a pragmatic perspective. The key idea here is that the algorithm should be *symmetric*, treating subexpressions identically so that there is no bias causing errors to tend to be reported in one part of a program rather than another. The new algorithm rests upon the novel concept of the *unification of substitutions* to allow the symmetric treatment of subexpressions. Section 4.1.2 introduces the operation of unifying substitutions and discusses how it will remove the left-to-right bias from type inference. Section 4.1.3 presents a variation of the classic type inference algorithm for Hindley-Milner type systems which uses this substitution unifying procedure.

Further uses of unifying substitutions are given in Section 4.1.5 and issues in implementing these ideas are discussed in Section 4.1.6.

4.1.1 Left-to-Right bias in W

Let us first consider a simple λ-calculus example of a Hindley-Milner untypeable expression. This function should take a real number and return the sum of the original number and its square root:

$$\lambda \mathsf{a}.\mathsf{add}\ \mathsf{a}\ (\mathsf{sqrt}\ \mathsf{a})$$

A typical error message from a type checker would read:

Cannot apply $\mathsf{sqrt} : \mathsf{real} \rightarrow \mathsf{real}$ to $\mathsf{a} : \mathsf{int}$.

A programmer, seeing this message, may be confused as a is intended to be real. So the mistake cannot be that sqrt is being applied to a, it must be that something else causes a to appear to be an int. The source of the problem will become apparent if we look at the type environment the expression is checked inside:

$$\begin{aligned} \mathsf{add} &: \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \\ \mathsf{sqrt} &: \mathsf{real} \rightarrow \mathsf{real} \end{aligned}$$

The mistake has been to use integer addition where real number addition is required.

Clearly in this case the error message is inappropriate as there is equal evidence that a should be a real as there is that it should be an int. The type checker has incorrectly given priority to the information derived from the leftmost subexpression — it has a left-to-right bias. It would have been more informative in the example if the type checker had pointed out that there was an inconsistency *between* the two subexpressions, instead of falsely claiming that either was internally inconsistent.

A classic example used to illustrate the monomorphism of function arguments is

$$\lambda I.\langle I\ 3, I\ \mathsf{true}\rangle$$

The programmer has written a function which expects the identity function as an argument. This is not possible in Hindley-Milner type systems as arguments cannot be used polymorphically. When a compiler is faced with this expression it type checks from left to right, first establishing that I must have a type of the form $\mathsf{int} \rightarrow \alpha$ and then finding that I cannot, therefore, be applied to true of type bool. The programmer will be given an error message of the form

Cannot apply $\mathsf{I} : \mathsf{int} \rightarrow \alpha$ to $\mathsf{true} : \mathsf{bool}$.

This message implies that there is an inconsistency inside $I\ \mathsf{true}$. In fact there is an inconsistency in the use of I between the two subexpressions. The algorithm presented here will find this inconsistency in the uses of I between the two subexpressions of $\langle I\ 3, I\ \mathsf{true}\rangle$ instead of finding an apparent mistake in $I\ \mathsf{true}$.

The type inference rule APP in Figure 2.3 states that if (given the type environment Γ) subexpression e_0 has type $\tau' \rightarrow \tau$ (it is a function), and that e_1 has type τ' under the same type environment (it is a suitable argument for the function), then the application of e_0 to e_1 has type τ. Non-determinism arises from the function argument type τ'. If we are attempting to show $\Gamma \vdash e_0 e_1 : \tau$, there is no way of knowing what τ' to use in the sub-derivation for each of e_0 and e_1. This is handled by introducing a type variable and using unification.

The inference rule tells us that in order for $e_0 e_1$ to be typeable, three conditions must be satisfied. e_0 must be typeable, e_1 must be typeable and the types of the two must be compatible for application. It is, therefore, desirable for error messages to describe a mistake as being either in e_0 or e_1, or as an incompatibility between them. We saw that this distinction is not made in current type checkers. Sometimes they announce an incompatibility as if it was a problem inside e_1. Similar statements can be made about the rule TUP which is used to type $\langle I\ 3, I\ \mathsf{true}\rangle$.

The result of type inference shown to the programmer (when it succeeds) is a polymorphic type scheme. If W returns (S, τ) then the type scheme is $\overline{(S\Gamma)}(\tau)$. Since Γ typically does not have any free type variables, all type variables in the result type will normally be universally bound.

Table 4.1 Ways in which W can fail to type check an application expression.

Point of failure	Possible meanings
Recursive call $W(\Gamma, e_0)$	There is an internal inconsistency in e_0. e_0 is incompatible with Γ.
Recursive call $W(S_0\Gamma, e_1)$	There is an internal inconsistency in e_1. e_1 is incompatible with Γ. There is an inconsistency between e_0 and e_1.
Unification $U(S_1\tau_0, \tau_1 \to \beta)$	e_0 cannot be applied to e_1.

The left-to-right bias arises in application expressions because the first substitution, S_0, is applied to the type environment, Γ, before type checking e_1. This means that if an identifier is used incorrectly in e_0 and used correctly in e_1 inference on e_1 could fail and wrongly imply that e_1 contains an error.

Table 4.1 shows the different ways in which the application case of W can fail, and how these can be interpreted. The concern here is that it is not possible to differentiate inconsistencies between e_0 and e_1 from inconsistencies inside e_1. Hence, it is the third cause of failure of $W(S_0, \Gamma, e_1)$ which we wish to eliminate. The solution proposed here is to delay applying the substitutions to Γ by having some means of combining substitutions. Such a means is described in the next section.

4.1.2 Unifying Substitutions

We have already seen some examples demonstrating the left-to-right bias of W. We have also seen how the algorithm works and know why the bias arises in the case of application. The objective of the new algorithm is to allow us to infer types and substitutions for each subexpression independently. The new algorithm U_S deals with combining substitutions, the next section shows how to modify W to make use of it and Section 4.1.5 shows how to further extend the algorithm and how to apply it to other type inference algorithms and other type systems.

To treat the subexpressions e_0 and e_1 independently in a modified version of W, the recursive calls must be $W(\Gamma, e_0)$ and $W(\Gamma, e_1)$. This will yield two result pairs: (S_0, τ_0) and (S_1, τ_1). It is necessary then to

- check that the two substitutions are consistent
- apply terms from S_0 to τ_1 and from S_1 to τ_0 so that the resulting types have no free type variable in the support of either substitution, and
- return a well-formed substitution containing entries from both S_0 and S_1.

The second of these requirements is not simply $S_0\tau_1$ and $S_1\tau_0$, because these could have unwanted free type variables. Likewise the third of these is not simply S_1S_0 or S_0S_1. The essence of these three operations can be summarised in these two requirements:

- check the substitutions are consistent, and if they are
- create a substitution which contains the effect of both.

We must *unify* the two substitutions.

4.1.2.1 Examples of Unifying Substitutions

Before we look at the algorithm for unifying substitutions, it will be worthwhile seeing some examples.

The simplest case is where the two substitutions are completely independent.

$$\begin{aligned} S_0 &= \{\alpha \mapsto \mathsf{int}\} \\ S_1 &= \{\beta \mapsto \gamma\} \\ U_S(S_0, S_1) &= \{\alpha \mapsto \mathsf{int}, \beta \mapsto \gamma\} \end{aligned}$$

If the supports of S_0 and S_1 contain a common type variable, we must unify the relevant types:

$$\begin{aligned} S_0 &= \{\alpha \mapsto \mathsf{int} \rightarrow \beta\} \\ S_1 &= \{\alpha \mapsto \gamma \rightarrow \mathsf{real}\} \\ U_S(S_0, S_1) &= \{\beta \mapsto \mathsf{real}, \gamma \mapsto \mathsf{int}\} \end{aligned}$$

Note that equivalent results cannot be obtained simply by composing the substitutions (S_0S_1 or S_1S_0). The previous example would have occurred inside the lambda term $\lambda f.\lambda x.(f\,1) + ((f\,x) + 0.1)$. S_0 is the substitution produced from $f\,1$ and S_1 comes from $(f\,x) + 0.1$ (α is the type variable for f).

Substitution unification can fail, for example with

$$\begin{aligned} S_0 &= \{\beta \mapsto \alpha \to \mathsf{real}\} \\ S_1 &= \{\beta \mapsto \mathsf{real} \to \mathsf{real}, \alpha \mapsto \mathsf{int}\} \end{aligned}$$

There is an inconsistency between the instantiations of α in this case.

Unification can also fail with an *occurs* error, as in this case.

$$\begin{aligned} S_0 &= \{\alpha \mapsto \mathsf{int} \to \beta\} \\ S_1 &= \{\beta \mapsto \mathsf{int} \to \alpha\} \end{aligned}$$

Clearly the two substitutions together here imply that α and β should map to infinite types, hence the two substitutions cannot be unified.

4.1.2.2 Formal Definition of Unifying Substitutions

A substitution, S', unifies substitutions, S_0 and S_1, if $S'S_0 = S'S_1$. In particular a most general unifier of a pair of substitutions is S' such that

$$(S'S_0 = S'S_1) \wedge (\forall S'' : (S''S_0 = S''S_1) \Rightarrow (\exists R : S'' = RS'))$$

i.e. S' unifies S_0 and S_1, and any other unifying substitution is an instance of S'.

The unified substitution, $S'S_0$, has the effect of both S_0 and S_1 since $S'S_0\alpha < S_0\alpha$ and $S'S_0\alpha < S_1\alpha$, for all α.

4.1.2.3 Algorithm U_S

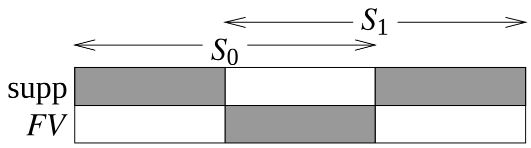
Algorithm U_S computes the most general unifier of a pair of substitutions.

To see how the algorithm works, note that the support of the result consists of three parts as shown in Figure 4.1. The algorithm to be introduced here deals with each of the three parts of the support separately. The free variables in the range of the unifier are free in either S_0 or S_1 and are in the support of neither.

The algorithm, commented in italics, is in Figure 4.2. Note that it can terminate with an occurs error as indicated, or if U fails.

Figure 4.1 The support of $U_S(S_0, S_1)$.

The support consists of three parts (shaded). The disjoint parts of the supports of S_0 and S_1, and the free variables in their ranges where their supports overlap



4.1.2.4 Verification of U_S

It must be shown that U_S does indeed compute the most general unifier of a pair of substitutions. Two theorems define this property.

Theorem 5 (Soundness of U_S) *For any pair of substitutions, S_0 and S_1, if $U_S(S_0, S_1)$ succeeds then it returns a unifying substitution.*

Proof of this appears in Appendix A.1.

Theorem 6 (Completeness of U_S) *If S'' unifies S_0 and S_1 then*

1. *$U_S(S_0, S_1)$ succeeds returning S', and*
2. *there is some R such that $S'' = RS'$.*

Proof of this appears in Appendix A.2.

4.1.3 The New Version of W

Now that we know what it means to unify two substitutions and have seen that this is possible, let us now look at the new type inference algorithm, W^{SYM}, in Figure 4.3. This differs from W only in the case for applications and tuples.

As stated earlier, the algorithm treats e_0 and e_1 symmetrically and features U_S in an analogous manner to (and in addition to) U.

Figure 4.2 Algorithm U_S commented in italics.

$U_S(S_0, S_1) \quad = \quad$ **let**

First split the supports into three disjoint parts:

$D_0 = \text{supp}(S_0) - \text{supp}(S_1) \qquad T_0 = S_0|_{D_0} = \{\alpha \mapsto S_0\alpha : \alpha \in D_0\}$

$D_1 = \text{supp}(S_1) - \text{supp}(S_0) \qquad T_1 = S_1|_{D_1} = \{\alpha \mapsto S_1\alpha : \alpha \in D_1\}$

$D_\cap = \text{supp}(S_0) \cap \text{supp}(S_1)$

Note: $FV(T_0) \cap \text{supp}(S_0) = \{\}$, *similarly for* T_1.

Start with T_0 *and add terms for variables in* D_1 *one at a time,*

always producing idempotent substitutions.

$S'_0 = T_0 \qquad \{\alpha_1 \mapsto \tau_1 \cdots \alpha_n \mapsto \tau_n\} = T_1$

$S'_{i+1} =$ **let**

Consider $\alpha_{i+1} \mapsto \tau_{i+1}$

Substitute away type variables of supp(S'_i) *from* τ_{i+1},

and remove α_{i+1} *from* S'_i:

$\tau'_{i+1} = S'_i\tau_{i+1}$

If $\alpha_{i+1} \in FV(\tau'_{i+1})$ **terminate (occurs error)**

in $\{\alpha_{i+1} \mapsto \tau'_{i+1}\}S'_i$

S'_n *is the unifier for* T_0 *and* T_1.

Now deal with type variables in $\{\beta_1 \cdots \beta_m\} = D_\cap$.

$V_0 = S'_n$

$V_{i+1} =$ **let**

$\tau_0 = V_iS_0\beta_{i+1} \qquad \tau_1 = V_iS_1\beta_{i+1}$

If $\beta_{i+1} \in FV(\tau_0) \cup FV(\tau_1)$ **terminate (occurs error)**

in $U(\tau_0, \tau_1)V_i$

in V_m

Figure 4.3 Algorithm W^{SYM}, cases for application and tuples.

$$
\begin{array}{rcl}
W^{SYM}(\Gamma, e_0 e_1) & = & \textbf{let} \\
& & \quad (S_0, \tau_0) = W^{SYM}(\Gamma, e_0) \\
& & \quad (S_1, \tau_1) = W^{SYM}(\Gamma, e_1) \\
& & \quad S' = U_S(S_0, S_1) \\
& & \quad \tau_0' = S'\tau_0 \qquad \tau_1' = S'\tau_1 \\
& & \quad V = U(\tau_0', \tau_1' \to \beta) \text{ (for new } \beta) \\
& & \textbf{in} \\
& & \quad (VS'S_0, V\beta)
\end{array}
$$

$$
\begin{array}{rcl}
W^{SYM}(\Gamma, \langle e_1, \ldots, e_n \rangle) & = & \textbf{let} \\
& & \quad (S_1, \tau_1) = W^{SYM}(\Gamma, e_1) \\
& & \quad \vdots \\
& & \quad (S_n, \tau_n) = W^{SYM}(\Gamma, e_n) \\
& & \quad S = S_1 +_{U_S} \ldots +_{U_S} S_n \\
& & \textbf{in} \\
& & \quad (S, \langle S\tau_1 \times \ldots \times S\tau_n \rangle)
\end{array}
$$

Where $S +_{U_S} T = (U_S(S,T))S$ (i.e. the *unified* substitution, rather than the *unifying* substitution). This operator is commutative and associative.

4.1.3.1 Correctness of W^{SYM}

Algorithm W^{SYM} should produce the same results as W. To verify this it is necessary to prove the soundness and completeness theorems for W^{SYM}. These theorems are analogous to those which Damas proved for W.

The algorithm is sound if every answer it gives is a type for the parameter expression under the type environment obtained from applying the substitution to the original type environment.

Theorem 7 (Soundness of W^{SYM}) *If $W^{SYM}(\Gamma, e)$ succeeds with (S, τ) then there is a derivation* *of* $S\Gamma \vdash e : \tau$.

The proof of this can be found in Appendix A.3.

Theorem 8 (Completeness of W^{SYM}) *Given* Γ *and e, let* Γ' *be an instance of* Γ *and* η *be a type scheme such that* $\Gamma' \vdash e : \eta$.
Then

1. $W^{SYM}(\Gamma, e)$ *succeeds*

2. *If* $W^{SYM}(\Gamma, e) = (P, \pi)$ *then for some R:* $\Gamma' = RP\Gamma$, *and* η *is a generic instance of* $R\overline{P\Gamma}(\pi)$.

Proof of this theorem can be found in Appendix A.4.

Because W^{SYM} satisfies the same soundness and completeness theorems as W, and we know that the solutions of these theorems are unique (from the principal type scheme theorem of [DM82]) we know that W^{SYM} always produces the same results as W.

Corollary 1 (W^{SYM} and W are equivalent) *For any pair,* (Γ, e), $W(\Gamma, e)$ *succeeds and returns* (S, τ) *if and only if* $W^{SYM}(\Gamma, e)$ *succeeds and returns* (S, τ).

4.1.4 Interpreting the Failure of W^{SYM}

The argument for using W^{SYM} is that it is possible to create better error messages when it fails than is possible with W. First, recall the ways in which the application case of W can fail and the possible causes of this as given in Table 4.1. In particular when $W(S_0\Gamma, e_1)$ fails this may be caused by e_1 being incompatible with a mistake in e_0. This is the sort of error message which programmers find so frustrating as it is not easy to find the original source of the problem from it.

Now consider the possible causes of failure of the application case of W^{SYM}, given in Table 4.2. Clearly from this, type checkers using W^{SYM} can produce more informative error messages than those using W.

4.1.5 Other uses of U_S

This section explores further uses of U_S. It can be used to type check larger expressions than simple applications symmetrically (for example curried expressions); and it can be used in other type inference algorithms.

Table 4.2 Ways in which W^{SYM} can fail to type check an application expression.

Point of failure	Possible meanings
Recursive call $W(\Gamma, e_0)$	There is an internal inconsistency in e_0. e_0 is incompatible with Γ.
Recursive call $W(\Gamma, e_1)$	There is an internal inconsistency in e_1 e_1 is incompatible with Γ.
Substitution unification $U_S(S_0, S_1)$	There is an inconsistency between e_0 and e_1.
Unification $U(S_1\tau_0, \tau_1 \to \beta)$	e_1 is not a suitable argument for e_0.

4.1.5.1 Larger Syntactic Structures

The type inference algorithm given earlier treated application expressions symmetrically. Similarly, the treatment of tuples and records is typically asymmetric and U_S can be used to eliminate this asymmetry.

Not only can U_S be used to treat simple application expressions symmetrically — it can also be used for curried applications of the form $e_0 e_1 \cdots e_n$. Each of the subexpressions must be type checked, then all the resulting substitutions are unified and the type of the curried expression is found. The advantage of this technique is that it allows type checking to follow the structure of the program in the way the programmer views it. This is discussed in Section 4.2.1.

4.1.5.2 Type Inference Algorithm M

It is clear that this algorithm suffers from the same left-to-right bias as W in the application and tuple cases, and again it is a simple matter to change M to remove these biases as can be seen from Figure 4.4.

If the inference $M^{SYM}(\Gamma, e_0, \beta \to \tau)$ fails, this implies that e_0 is not a function, or does not have the correct return type. The inference $M^{SYM}(\Gamma, e_1, \beta)$ will fail if and only if Γ and e_1 are inconsistent (there is no typing for $\Gamma \vdash e_1$). If the unification fails then either e_1 is not a suitable argument for e_0, or there is some other inconsistency between them.

Figure 4.4 The application case for the modified M.

$$
\begin{array}{rcl}
M^{SYM}(\Gamma, e_0 e_1, \tau) & = & \textbf{let} \\
 & & \quad S_0 = M^{SYM}(\Gamma, e_0, \beta \to \tau) \text{ for new } \beta \\
 & & \quad S_1 = M^{SYM}(\Gamma, e_1, \beta) \\
 & & \textbf{in } (U_S(S_1, S_0))S_0
\end{array}
$$

$$
\begin{array}{rcl}
M^{SYM}(\Gamma, \langle e_1, \ldots, e_n \rangle, \tau) & = & \textbf{let} \\
 & & \quad S = U(\tau, \langle \beta_1, \ldots, \beta_n \rangle \text{ for new } \beta\text{s} \\
 & & \quad S_1 = M^{SYM}(S\Gamma, e_1, S\beta_1) \\
 & & \quad \vdots \\
 & & \quad S_n = M^{SYM}(S\Gamma, e_n, S\beta_n) \\
 & & \quad S' = S_1 +_{U_S} \ldots +_{U_S} S_n \\
 & & \textbf{in} \\
 & & \quad S'
\end{array}
$$

Where $S +_{U_S} T = (U_S(S,T))S$ (i.e. the *unified* substitution, rather than the *unifying* substitution). This operator is commutative and associative.

4.1.6 Implementation

The modified version of W has been implemented for a simple λ-with-let calculus. The implementation has also been extended to deal with curried expressions.

One difficulty in implementing type inference using U_S for a full programming language is that it prevents substitutions from being implemented using references (as mentioned in Section 2.3.1). Because updating global references represents a greedy strategy it is in conflict with the cautious approach taken here of waiting until as late as possible to apply substitutions.

4.2 Other Syntactic Forms

U_S is new function for changing the order of type inference, but other changes can be made to type inference without using this.

4.2.1 Curried Expressions

A common syntactic structure in functional programs is the curried expression. There are particular problems with producing error messages for curried expressions as type inference algorithms strictly follow the series of applications, but the programmer does not really regard these as a series of applications. Instead the view is as a single application with several arguments (e.g. Paulson [Pau96] describes `map` as being both a function with two arguments and as a functional which returns a function).

The difficulty is that a series of unifications is used to type check the expression in W, and any of these could fail resulting in a error message referring to a function which is itself an application expression and which misses some arguments. Similarly, M constrains each subexpression as being a function rather than constraining the first to be a higher-order function.

It is easy to change the application case of W or M to use a single unification to deal with curried applications. These changes are shown in Figures 4.5 and 4.6.

Figure 4.5 Curried application case for W.

$$
\begin{array}{rcl}
W^{CURR}(\Gamma, f\ a_1 \ldots a_n) & = & \mathbf{let} \\
& & \quad (S_0, \tau_f) = W(\Gamma, f) \\
& & \quad (S_1, \tau_1) = W(S_0\Gamma, a_1) \\
& & \quad \vdots \\
& & \quad (S_n, \tau_n) = W(S_{n-1} \ldots S_0\Gamma, a_n) \\
& & \quad S = U(S_n \ldots S_1\tau_f, S_n \ldots S_1(\tau_1 \to \ldots \to \tau_n \to \beta)) \text{ (for new } \beta) \\
& & \mathbf{in}\ (SS_n \ldots S_0, S\beta)
\end{array}
$$

To implement these, it is typically necessary to write a short routine to go through the abstract syntax of an application expression and find the function and list of arguments.

Figure 4.6 Curried application case for M.

$$
\begin{aligned}
M^{CURR}(\Gamma, f\ a_1 \ldots a_n, \tau) \quad = \quad & \textbf{let} \\
& \quad S_0 = M(\Gamma, \beta_1 \to \ldots \to \beta_n \to \tau) \text{ (for new } \beta\text{s)} \\
& \quad S_1 = M(S_0\Gamma, S_0\beta_1) \\
& \quad \vdots \\
& \quad S_n = M(S_{n-1}\ldots S_0\Gamma, S_{n-1}\ldots S_0\beta_n) \\
& \textbf{in } S_n \ldots S_0
\end{aligned}
$$

This change has been made to W in the MLj compiler as part of the work described in Chapter 6. These changes to the algorithms can also be accompanied with uses of U_S as discussed previously.

4.2.2 Signatures

A complaint of Standard ML programmers, and a criticism sometimes levied against the language, is that type specifications in signatures are not used during the type checking of a structure. For example in the following program

```
structure Test : sig
  val testVal : int
  val testFun : int -> int
end =
struct
  val testVal = 1.0
  fun testFun x = (x * 2) + testVal
end
```

Type checking this will typically yield at least two errors stating, in this order

- The arithmetic expression cannot be correct as it mixes real and integer arithmetic (this could result in several messages)

- The structure does not match its signature because `testVal` is specified as an int and defined as a real.

The set of error messages is clearly inconsistent, on the one hand there is the complaint that testVal should be an integer and on the other there is the complaint that testVal cannot be added to integers.

It would be preferable if definitions were checked against their specification before moving on to the next definition. It might be suggested that the specification is more likely to be correct than the definition (as in an implicitly typed language the programmer has gone to some effort to supply it, and it is typically smaller than a function definition). In the case of the example, this would result in a message that `testVal` does not meet its specification, but no messages about the arithmetic expression as thereafter `testVal` would be assumed to be an integer.

One difficulty about this regards SML programs such as

```
structure Test : sig
  val test : int
end =
struct
  val test = 1.0
  val two = 1.0 + test
  val test = 1
end
```

This program is correctly typed even though it appears that the first definition of `test` does not meet its specification. The program is correctly typed because only the second `test` becomes part of the structure which the signature must match. Hence, to generate valid error messages it is necessary to first establish which definitions actually need to be checked against the signature. This is generally a simple syntactic analysis (though SML's `open` in a functor can make it more difficult). This syntactic structure is quite common as it is often useful to define one version of a function (e.g. a pretty printer taking a syntax tree *and* a record of tabbing information) and then export a simpler version (e.g. the pretty printer taking only a syntax tree) but to give both the same name.

Changing the syntactic structure of the language will also prevent this problem as can be see by Haskell. Haskell requires that type specifications are placed next to definitions and type checks each definition against its specification before moving on to the next definition.

4.3 Conclusions

This chapter introduced the concept of *unifying substitutions* in type inference. This new operation was used to create symmetric type inference algorithms. The algorithm for unification and modified type inference algorithms were given and proved sound and complete.

The chapter also discussed other changes to the order of type inference, in particular using a single unification for curried expressions and using signature constraints while type checking structures.

Chapter 5

Graphs for Type Inference[1]

This chapter presents a way of recording information about the types of programs as a graph, which works whether or not the program is correctly typed. The graph describes how the types of expressions were inferred. Information proposed by other authors as means to help programmers debug programs (seen in Chapter 3) can be extracted from these graphs.

This chapter starts by looking at the motivation behind this data structure, then describes the graphs (Section 5.2) and the algorithms for generating and analysing them (Section 5.3). Methods for extracting more useful information are given in Section 5.4. SML implementations of the algorithms are described in Section 5.5 and, finally, conclusions are summarised in Section 5.6.

5.1 Purpose of Graphs

In conventional type inference (algorithms W and M for example), the algorithm fails when the input program cannot be typed. Most compilers then employ *ad hoc* methods to produce error messages and to continue type checking as well as is possible. Most of the work reviewed in Chapter 3 is a more systematised way to deal with failure of a basic algorithm, or a replacement algorithm which can also fail under some circumstances.

One problem with the existing work is that typically the whole program is not inspected before generating an error message. Lee and Yi [LY98] give the argument that it is better to inspect as little of the program as possible before discovering that there is a type error as this will reduce the possible space in which the mistake might lie. This however does not consider the fact

[1] This chapter is based on material in [McA99a] and [McA99b].

that while the mistake must lie within the inspected portion of the program, information which could help track down and repair the mistake may lie elsewhere. For example it may be quickly established that a particular monomorphic function is applied to both integers and strings, and thus there is a type error. Inspecting the rest of the program may establish that the function is used repeatedly on integers, and hence the application to a string is likely to be the mistake.

A second problem is that the methods proposed by different authors to help programmers are largely incompatible. It would be desirable to present several different types of information (for example a proposed location for the mistake in the style of [Wan86] *and* an explanation of how the type was inferred in the style of [Dug98]).

In this chapter the intention is to be able to inspect the whole of any syntactically valid program building up information about the types in it. After the information has been collected (as a graph) it can be inspected to find out whether the program was typed, what type (if any) it has, and other information.

This chapter also shows that most of the work previously proposed for helping programmers facing this problem can be characterised in a uniform way.

5.2 Graphs

The structure of these graphs follows the structure of types (they are not annotated syntax trees). The graphs are analogous to the representation of types using mutable references for type variables: an edge is analogous to a reference being filled with a type. There are vertices representing the types of expressions and vertices representing type constructors. The varieties of vertex are illustrated and defined in Figure 5.1.

The types of vertices are

Program fragment vertices. These are identified by a program fragment, f. A fragment is a subexpression of the program and its location within the program, i.e. a node of the syntax tree, or a particular *occurrence* of a subexpression. These may also be tagged by a sequence of other fragments to give $[f_0]_{f_1,f_2\ldots f_n}$. The tagging is used to implement Hindley-Milner polymorphism as explained later. These vertices represent the types of fragments or instances of the types of these from instantiation of polymorphic type schemes.

The circle denotes the fact that there is a type variable associated with every fragment in a program, and shows that edges can come from the fragment.

Figure 5.1 Vertices.

Illustrated are a vertex for the program fragment $\lambda i.i$; the fragment $\lambda i.i$ tagged with an instance of I; a nullary type-constructor, int; a unary type-constructor, list; the binary function type-constructor; and a type variable, α.

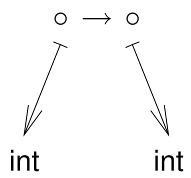
$$\circ\lambda i.i \qquad \circ[\lambda i.i]I \qquad \mathsf{int} \qquad \circ\,\mathsf{list} \qquad \circ \rightarrow \circ \qquad \alpha$$

v	$::=$	$\circ f$	A program fragment
	$\mid$	$\circ[f_0]_{f_1,f_2\ldots f_n}$	A program fragment tagged by other program fragments
	$\mid$	$(\circ_0 \ldots \circ_{n-1})c_i$	A type constructor c with arity n and graph-wide unique identifier i
	$\mid$	$c_i.j$	The jth connection point of the vertex $(\circ_1 \ldots \circ_n)c_i$
	$\mid$	α	A type variable

Type constructor vertices. These are identified by a type constructor and some unique identifier. For example, there may be several function type constructor vertices in a graph each with its own identifier: $\rightarrow_1, \rightarrow_2 \ldots$. This variety of vertex contains a number of sub-vertices called *connection points*. Function type constructor vertices are denoted $\circ \rightarrow_i \circ$ with two connection points, and an identifier i. The connection points may be referred to as $\rightarrow_i .0$ and $\rightarrow_i .1$. In general a type constructor vertex c with identifier i and arity n is $(\circ_0 \ldots \circ_{(n-1)})c_i$. There is also a special nullary type constructor unbound.

Type variable vertices. These are identified by some unique identifier and written α (much like type variables in traditional type inference).

Figure 5.2 The type int $\rightarrow$ int



Edges are added between vertices which must represent the same type. Written $\overset{f}{\mapsto}$, they are labelled by program fragments and cannot go from type constructor or type variable vertices. They can, however, go from the connection points of type constructor vertices. Types are built up by edges going *from* connection points, as in Figure 5.2. Following is a list of the combinations of vertices an edge can go between, and their meanings. The algorithm for building a graph for a particular program is given later, in Section 5.3.1.

- $\circ f \overset{l}{\mapsto} (\circ_0 \ldots \circ_{n-1})c_i$ (an edge from a program fragment to a type constructor) means that the type of the fragment, f, is constructed with the given type constructor. For example $\lambda i.i \overset{l}{\mapsto} \circ \to \circ$ means that $\lambda i.i$ is a function. Rather than going to a type constructor, the edge may go to a type variable (in which case we don't know anything about the type of the fragment).

- $\circ f \overset{l}{\mapsto} \circ f'$ (an edge from one fragment to another) means that the first fragment has the same type as the second.

- $\circ f \overset{l}{\mapsto} \circ[f']_f$ (an edge from a program fragment to a tagged program fragment) means that the type of f is an instance of the type of f'.

- $\circ c_i.j \overset{l}{\mapsto} v$ (an edge from a connection point to any vertex) says that one argument of the type constructor is given by the type at v.

- $\circ x \overset{x}{\mapsto} \mathsf{unbound}$ (edge from identifier x to the unbound type constructor, labelled by x) means that x is an unbound identifier.

Figure 5.3 shows the graph for the identity function. The graph shows that $\lambda i.i$ must be a function and that the type of the function argument is the same as its result.

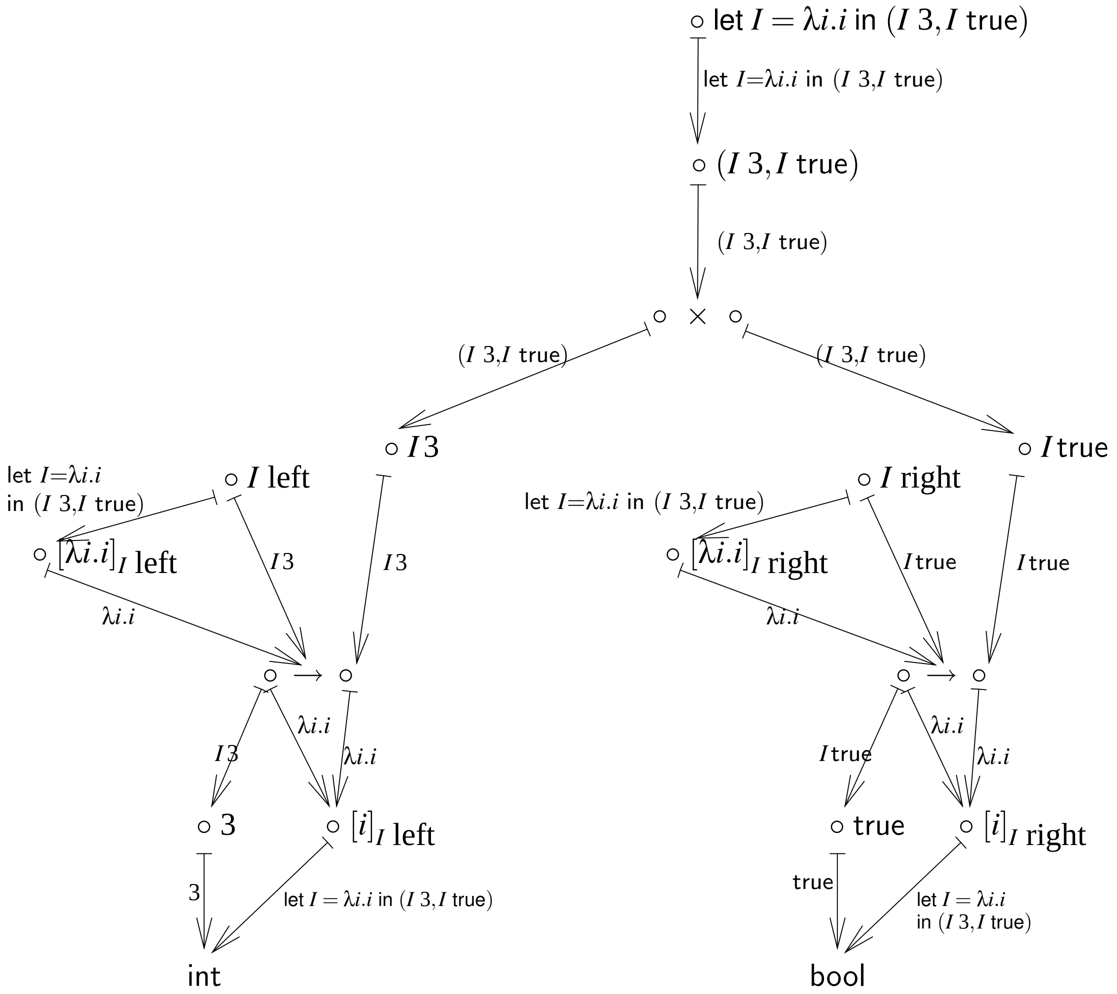
Figure 5.3 Graph for the identity function. Read as $\alpha_i \to \alpha_i$.

$$\begin{tikzcd}
 & \circ\lambda i.i \arrow[d, mapsto, "\lambda i.i"] & \\
\circ \arrow[dr, mapsto, "\lambda i.i"'] & \longrightarrow & \circ \arrow[dl, mapsto, "\lambda i.i"] \\
 & \circ i &
\end{tikzcd}$$

A more complex example is given in Figure 5.4. This is a let expression, $\mathsf{let}\ I = \lambda i.i\ \mathsf{in}\ (I\ 3, I\ \mathsf{true})$, which contains function applications. There are several features to observe.

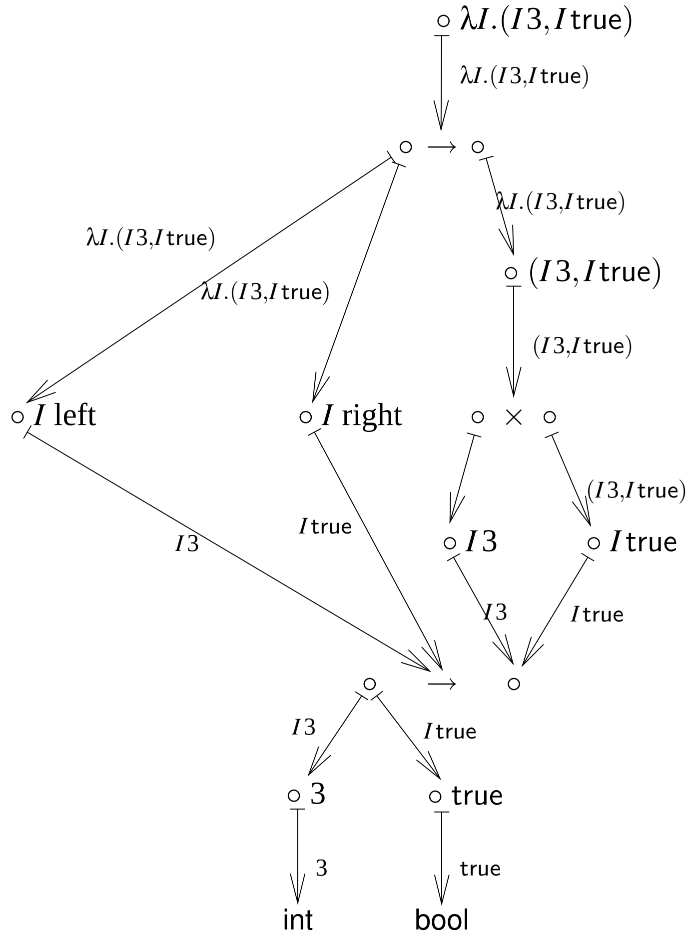
- The graph for $\lambda i.i$ (seen in Figure 5.3) appears twice, labelled with each occurrence of I. This is because each instance of identifier I is an instance of the type of $\lambda i.i$.
- The application expressions, I 3 and I true, tell us that each occurrence of I must be a function.
- The application expressions tell us how the type schemes of $\lambda i.i$ should be instantiated.
- Tuples are represented in the obvious way, with a connection point for each component type.

Figure 5.4 A let expression. The type is read as int $\times$ bool.



Graphs can also be generated for untypeable programs. The program in Figure 5.5 is similar to that of the previous figure, except that I is λ-bound instead of let-bound — this makes the program untypeable. From the graph, we can see roughly what the type should be (a function taking a function and returning a tuple).

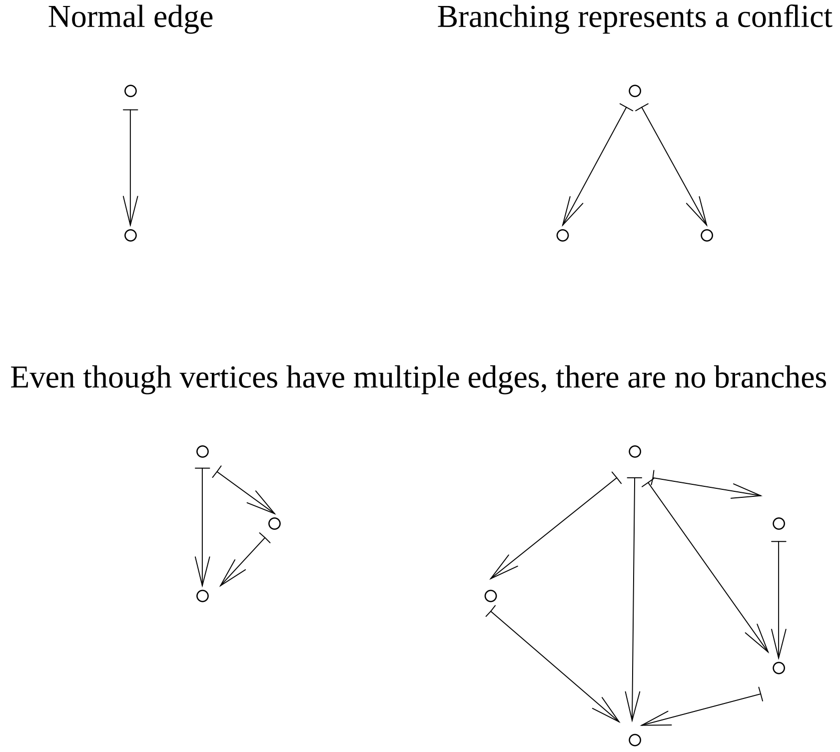
Figure 5.5 An untypeable program.



The difference between the patterns of edges in Figures 5.4 and 5.5 are shown in Figure 5.6. Sometimes when a vertex has several edges from it, they all ultimately meet up, other times they diverge. When edges diverge, we will call it a branch. Branches indicate that programs are untypeable. There is a conflict between two types, corresponding to a unification failure in conventional type inference.

A program may also be untypeable if it has unbound identifiers. Figure 5.7 shows the graph

Figure 5.6 Patterns of edges.



for a program with unbound identifiers. The unbound identifiers are marked with the unbound type constructor. In Section 5.4.1 we will see how to read the required types of unbound identifiers following the example of [BS95].

The final way in which a graph can indicate that its program is untypeable is if it contains a cycle, as in Figure 5.8. This corresponds to an occurs error in traditional type inference. Cycles are indirect, as they can span across the gaps between type constructor vertices and their connection points, e.g. in the figure, it is not possible to reach any vertex from itself, but it is possible to reach the lower $\rightarrow$ from its own connection point.

5.3 Algorithms

Having informally seen some examples of graphs, we can now see how they are actually used to find out basic information about programs. Analysis of programs using graphs is a two stage process: first generate a graph, then read useful information from the graph. We will look at the algorithm for generating graphs representing typings (Section 5.3.1). Then in Section 5.3.2 we

Figure 5.7 An open expression.

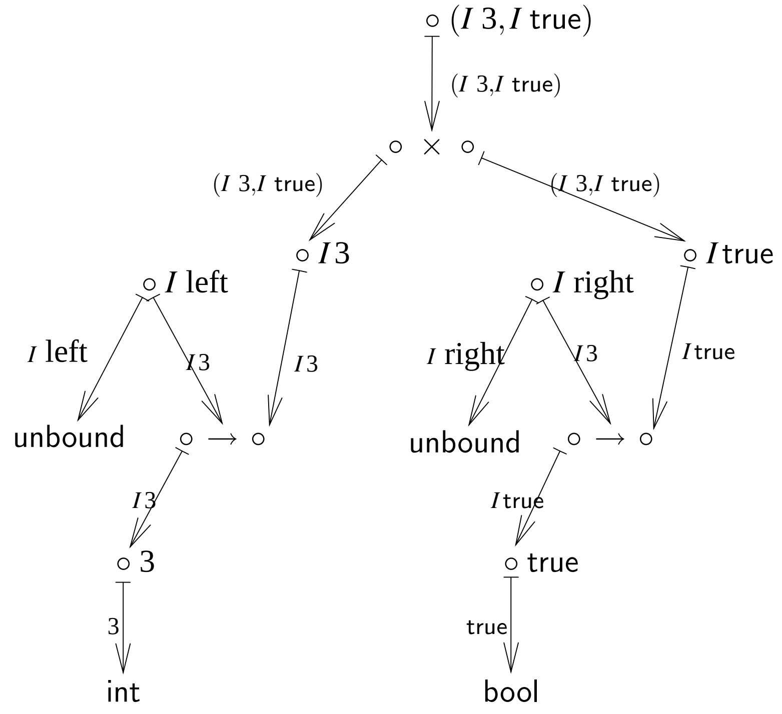
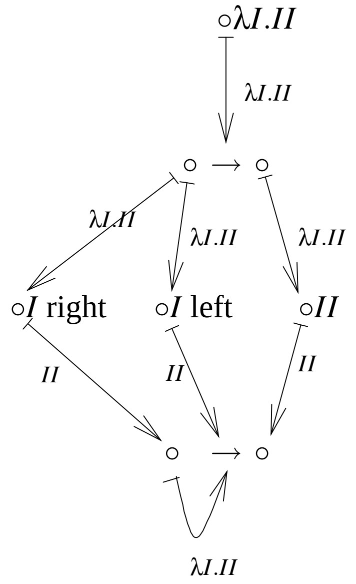


Figure 5.8 A graph with a cycle.



will see how to read typings from graphs.

5.3.1 Generating Graphs

There are several important algorithms for generating graphs: fine scale traversal, closing the graph, and the actual generation algorithm. Once a graph has been generated, the algorithms in Section 5.3.2 tell us how to interpret it.

5.3.1.1 Fine Scale Traversal

Generating and reading graphs requires an operation which searches for all the type constructor vertices, type variable vertices and other vertices without children, reachable from a particular root vertex. The algorithm for this can be found in Figure 5.9.

Figure 5.9 Algorithm search, for finding important vertices starting from a root.

Takes a graph and vertex, returns three sets of vertices.

$$
\begin{aligned}
\mathsf{search}(G,(\circ_0,\cdots,\circ_{n-1})c) &= (\{(\circ_0,\cdots,\circ_{n-1})c\},\{\},\{\}) \\
\mathsf{search}(G,\alpha) &= (\{\},\{\alpha\},\{\}) \\
\mathsf{search}(G,v) &= \textbf{if } \mathsf{children}(G,v)=\{\} \textbf{ then} \\
&\qquad (\{\},\{\},\{v\}) \\
&\quad \textbf{else} \\
&\qquad \textbf{let} \\
&\qquad\quad r=\bigcup\{\mathsf{search}(G,v') : v' \in \mathsf{children}(G,v)\} \\
&\qquad \textbf{in} \\
&\qquad\quad (\bigcup\{cs : (cs,tvs,vs) \in r\}, \\
&\qquad\quad \bigcup\{tvs : (cs,tvs,vs) \in r\}, \\
&\qquad\quad \bigcup\{vs : (cs,tvs,vs) \in r\})
\end{aligned}
$$

The three sets returned by search are:

Type constructor vertices. If this contains several instances of any one type constructor (e.g. several lists) then these instances should be merged (as described shortly) to close the graph.

Type variable vertices. If there are several type variables found then they can be merged. If any type constructors were found then the type variables can be removed.

Other vertices with no edges going from them. These should have edges attached from them to the type constructor or type variable vertices.

If search returns more than one vertex, then the graph currently contains a branch at the given root vertex.

The search function is used whenever the graph must be traversed. We use it to ignore vertices which have edges from them and therefore have been instantiated as another type. In most type inference algorithms based on references, chains of references are eliminated (aliasing), but in the form of inference in this paper we keep the vertices corresponding to these chains of references as they represent valuable information.

The search function stops at type constructors. The remaining functions will traverse the graph through type constructors to their connection points.

5.3.1.2 Closing a graph

Adding an edge to a graph (to say that I has the same type as $\lambda i.i$) is shown in Figure 5.10. This involves a *close* operation: an extra edge must be added to close up the branch that is created (this edge is highlighted in the last part of the figure). Closure is used to ensure that:

- There is at most one instance of any type constructor reachable from a vertex
- If a type constructor and some other vertex are reachable from any vertex then there is a path from the other vertex to the type constructor.

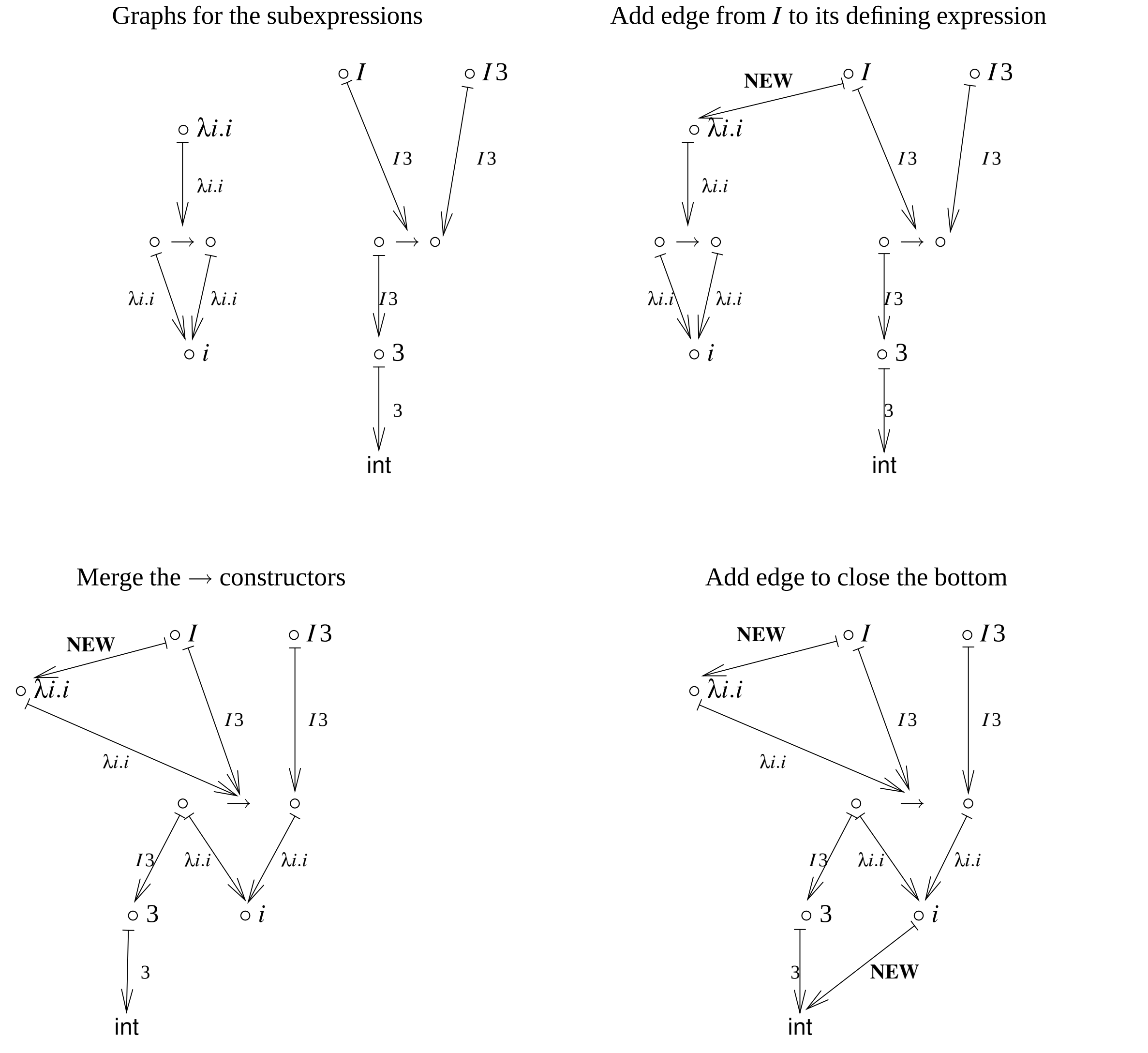
In order to do this, closure merges distinct instances of type constructors, and adds edges from other vertices to type constructors.

Figure 5.11 shows the closure algorithm, and Figure 5.12 the accompanying merging algorithm (the two are mutually recursive).

There are four possible results of search in the main case of close.

- If there are no other vertices reachable, then do not do anything to the graph (just return an updated list of vertices seen).

Figure 5.10 Adding an edge to a graph and closing the graph.



- If there are several vertices reachable, none of which are type constructors or variables, then create a new type variable vertex and link to it (e.g. if $x \mapsto y$ and $x \mapsto z$ then we must have $y \mapsto \alpha$ and $z \mapsto \alpha$ to eliminate the branch).

- If there are several type variables (and other vertices), then remove all but one arbitrary type variable and redirect all edges to all the type variables to the nominated type variable (and connect all other vertices to the remaining type variable).

- If there are type constructors reachable, then merge all similar type constructors (i.e. merge all $\circ \rightarrow \circ$ vertices and all int vertices), remove the type variables connecting their edges to all the type constructors (type variables are removed if the actual type is known), and connect any other vertices to all the type constructors.

merger takes a set of vertices. It picks a vertex and finds all the vertexes like it in the set, then removes all the similar vertices from the graph and set and connects their edges to the remaining one then closes below the remaining one. It repeats this until no vertices are left in its set.

5.3.1.3 Generating a graph

The graph generation algorithm in Figure 5.13 is quite simple. By closing the graph whenever an edge is added, there is no need for an explicit call to a unification function and because substitutions and types are combined in one data structure there is no need for explicit operations on substitutions.

The algorithm makes use of a function, $\mathsf{free}(e,x)$, which returns every syntactic instance of x in e. These fragments will be vertices in the graph for e.

The function generate must also make use of a type environment which keeps track of which identifiers are in scope and which refer to the basis environment. Γ is a pair, (I,B), of a set of bound identifiers, I, and a basis, B, mapping identifiers to a graph and vertex pair, (G,v). If generate encounters an identifier in I then it will produce a one-node graph, if the identifier is in the domain of B then a copy of the corresponding graph in B is used. If an identifier is in neither I nor B a graph representing the unbound type is returned. In generate, identifiers are added to I but B is never modified.

The last case of generate (for let expressions) is the most complex as it must deal with polymorphism. The graphs for the definition and use subexpressions are generated. A vertex for the let expression is added to the graph, and an edge connects it to the use expression vertex. The graph for the definition expression is copied and tagged with every instance of the bound identifier — unless there are no instances of the bound variable, in which case the graph is not altered. Edges connect instances of x to the tagged expression vertices. The graph is closed below every instance of the bound identifier.

The pattern of recursion in the generation function is symmetric (in the sense of Chapter 4. The graphs for the two sides of an application expression are generated independently as in Chapter 4.

Figure 5.11 The closure algorithm.

Takes a graph, vertex, label for new edges and set of vertices seen so far (initially this should be empty). Returns updated graph, and updated list of vertices seen.

$$
\begin{array}{rcl}
\mathsf{close}((V,E),(\circ_0\cdots\circ_{n-1})c,l,s) & = & \textbf{if } (\circ_0\cdots\circ_{n-1})c \in s \textbf{ then } ((V,E),s) \\
 & & \textbf{else } \mathsf{closeSet}((V,E),\{\circ_0\cdots\circ_{n-1}\},s,l) \\
\mathsf{close}((V,E),v,l,s) & = & \textbf{if } v \in s \textbf{ then } ((V,E),s) \textbf{ else} \\
 & & \textbf{case } \mathsf{search}(v,(V,E)) \textbf{ of} \\
 & & \quad (\{\},\{\},\{v'\}) \Rightarrow (G, s \cup \{v'\}) \\
 & & \mid (\{\},\{\},vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad \alpha = \mathsf{freshTyvarVertex}() \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V \cup \{\alpha\}, E \cup \{(v,l,\alpha) : v \in vs\}), s \cup \{v\}) \\
 & & \mid (\{\},\{\alpha\} \cup tvs, vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad E_0 = \{(v_0,l,v_1) : (v_0,l,v_1) \in E \wedge v_1 \notin tvs\} \\
 & & \qquad\qquad E_1 = \{(v_0,l,\alpha) : (v_0,l,\beta) \in E \wedge \beta \in tvs\} \\
 & & \qquad\qquad E_2 = \{(v_0,l,\alpha) : v_0 \in vs\} \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V - tvs, E_0 \cup E_1 \cup E_2), s \cup \{v\}) \\
 & & (cs, tvs, vs) \Rightarrow \textbf{let} \\
 & & \qquad\qquad (cs',(V',E'),s') = \mathsf{merger}((V,E),s,cs,l) \\
 & & \qquad\qquad E'' = \{(v_0,l,c) : (v_0,l',\alpha) \in E' \wedge \alpha \in tvs \wedge c \in cs'\} \\
 & & \qquad\qquad E''' = \{(v_0,l,c) : v_0 \in vs \wedge c \in cs'\} \\
 & & \qquad \textbf{in} \\
 & & \qquad\qquad ((V' - tvs, E'' \cup E'''), s')
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{closeSet}(G,\{\},l,s) & = & (G,s) \\
\mathsf{closeSet}(G,\{v\} \cup V,l,s) & = & \mathsf{closeSet}(\mathsf{close}((G,v,l,s),V,l)
\end{array}
$$

Figure 5.12 Merger algorithm.

Takes a graph, a set of seen vertices, a set of type constructor vertices and a label, merges all similar type constructors in the graph. Returns the remaining constructors from the set, the new graph, and the seen vertices.

$$
\begin{array}{rcl}
\mathsf{merger}(G,s,\{\},l) & = & (\{\},G,s) \\
\mathsf{merger}((V,E),s,\{(\circ_0\cdots\circ_{n-1})c\}\cup cs,l) & = & \textbf{let} \\
 & & \quad like = \{c' : c' \in cs \wedge \mathsf{tycon}(c') = \mathsf{tycon}(c)\} \\
 & & \quad unlike = \{c' : c' \in cs \wedge \mathsf{tycon}(c') \neq \mathsf{tycon}(c)\} \\
 & & \quad E_0 = \{(v_0,l,c) : (v_0,l,c') \in E \wedge c' \in like\} \\
 & & \quad E_1 = \{(v_0,l,c.\imath) : (v_0,l,c'.\imath) \in E \wedge c' \in like\} \\
 & & \quad E_2 = \{(v_0,l,v_1) : \\
 & & \qquad (v_0,l,v_1) \in E \wedge v_1 \notin like \wedge \\
 & & \qquad (\nexists c \in like : c.\imath = v_1)\} \\
 & & \quad E' = E_0 \cup E_1 \cup E_2 \\
 & & \quad E'_0 = \{(c.\imath,l,v_1) : (c'.\imath,l,v_1) \in E' \wedge c' \in like\} \\
 & & \quad E'_1 = \{(v_0,l,v_1) : \\
 & & \qquad (v_0,l,v_1) \in E' \wedge \nexists c \in like : c.\imath = v_0\} \\
 & & \quad E'' = E'_0 \cup E'_1 \\
 & & \quad (G',s') = \mathsf{close}((V - like, E''),c,l,s) \\
 & & \quad (cs'',(G'',s'')) = \mathsf{merger}(G',s',unlike,l) \\
 & & \textbf{in} \\
 & & \quad (cs'' \cup \{c\},(G'',s''))
\end{array}
$$

$\imath$ denotes an arbitrary connection point index.

E_0 is the set of edges which went to connection points in *like*, moved so that they go to the corresponding connection point of c.

Figure 5.13 Generating a graph.

Takes an environment consisting of a set of bound variables and a basis, and a program. Returns a graph for the programs.

$$
\begin{array}{rcl}
\mathsf{generate}((I,B),x) & = & \textbf{if } x \in I \textbf{ then } ((\{x\},\{\}),x) \\
& & \textbf{else if } (x,((V,E),v)) \in B \textbf{ then } ((V \cup \{x\}, E \cup \{x,x,v\}),x) \\
& & \textbf{else (} x \textbf{ is unbound) let } v = \mathsf{vertex}(\mathsf{unbound}) \textbf{ in } (\{x,v\},\{(x,x,v)\}) \\
\mathsf{generate}((I,B),\lambda x.e_0) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}((I \cup \{x\},B),e_0) \\
& & \quad v = \mathsf{vertex}(\rightarrow) \qquad V = V_0 \cup \{\lambda x.e_0, v\} \\
& & \quad E = E_0 \cup \{(\lambda x.e_0, \lambda x.e_0, v), (v.1, \lambda x.e_0, e_0)\} \cup \\
& & \qquad \{(v.0, \lambda x.e_0, e) : e \in \mathsf{free}(e_0,x)\} \\
& & \textbf{in} \\
& & \quad \mathsf{close}((V,E),\lambda x.e_0,\lambda x.e_0) \\
\mathsf{generate}(\Gamma,e_0e_1) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}(\Gamma,e_0) \quad (V_1,E_1) = \mathsf{generate}(\Gamma,e_1) \\
& & \quad v = \mathsf{vertex}(\rightarrow) \qquad V' = V_0 \cup V_1 \cup \{v, e_0e_1\} \\
& & \quad E' = E_0 \cup E_1 \cup \{(e_0e_1, e_0e_1, v.1), (v.0, e_0e_1, e_1), (e_0, e_0e_1, v)\} \\
& & \textbf{in} \\
& & \quad \mathsf{close}((V',E'),e_0,e_0e_1) \\
\mathsf{generate}((I,B),\mathsf{let}\ x = e_0 \ \mathsf{in}\ e_1) & = & \textbf{let} \\
& & \quad (V_0,E_0) = \mathsf{generate}((I,B),e_0) \\
& & \quad (V_1,E_1) = \mathsf{generate}((I \cup \{x\},B),e_1) \\
& & \quad V = V_1 \cup \{\mathsf{let}\ x = e_0 \ \mathsf{in}\ e_1)\} \\
& & \quad E = E_1 \cup \{(\mathsf{let}\ x = e_0 \ \mathsf{in}\ e_1, \mathsf{let}\ x = e_0 \ \mathsf{in}\ e_1, e_1)\} \\
& & \quad G' = \textbf{if } \mathsf{free}(e_1,x) = \{\} \textbf{ then } \{G_0\} \textbf{ else} \\
& & \qquad \{[G_0]e : e \in \mathsf{free}(e_1,x)\} \\
& & \quad V' = V \cup \bigcup\{V : (V,E) \in G'\} \\
& & \quad E' = E \cup \bigcup\{E : (V,E) \in G'\} \\
& & \qquad \cup\{(e, \mathsf{let}\ x = e_0 \ \mathsf{in}\ e_1, [e_0]e) : e \in \mathsf{free}(e_1,x)\} \\
& & \textbf{in} \\
& & \quad \mathsf{closeSet}((V',E'),\mathsf{free}(e_1,x),\mathsf{let} x = e_0 \mathsf{in} e_1)
\end{array}
$$

5.3.2 Reading Graphs

There are two distinct questions to be answered by reading a graph

1. Does the entire graph show that the expression is typeable?

2. What is the type represented by some vertex within the graph?

To find the typing, $\Gamma \vdash e : \tau$, for an expression, e, we must generate a graph, G, for the expression. Then find out whether the entire graph shows the expression is typeable (i.e. answer question 1), then if it is find the type τ represented by the vertex e (answer question 2).

We can consider the first question to be analogous to type checking (it has a boolean response), and the other to be analogous to type inference (it results in an inferred type).

5.3.2.1 Type Checking

To check whether the entire graph represents a valid typing, we must visit every vertex and check the following

- There are no branches, i.e. the vertex has at most one type constructor or one type variable vertex reachable from it (by search).
- The vertex is not part of a cycle (it is not reachable from itself). The cycle could involve type constructors and their connection points.
- The vertex is not the unbound type constructor.

These conditions are given by Definition 1.

Definition 1 (Valid Typing) (V, E) *is the graph for a correctly typed program if it has*

- *No branches:*

$$\begin{aligned} &\nexists v \in V : \exists v_1, v_2 \in V : \\ &\qquad v \mapsto^* v_1 \ \wedge\ (\nexists v_1' : v_1 \mapsto v_1') \ \wedge \\ &\qquad v \mapsto^* v_2 \ \wedge\ (\nexists v_2' : v_2 \mapsto v_2') \ \wedge \\ &\qquad v_1 \neq v_2 \end{aligned}$$

  Where $\mapsto^*$ is the reflexive transitive closure of $\mapsto$. i.e. there is at most one leaf reachable from any vertex (all type constructor and type variable vertices are leaves).

- *No cycles:* $\nexists v \in V : v \Rightarrow^{+} v$.
  Where $v \Rightarrow v'$ *iff* $v \mapsto v'$ *or* v' *is a connection point of* v*,*
  and $\Rightarrow^{+}$ *is the non-reflexive transitive closure of* $\Rightarrow$*.*

- *No unbound identifiers:* $\nexists i : \mathsf{unbound}_i \in V$.

An algorithm for checking this is a depth first search of the graph (branching at type constructors, to check the connection points). It stores the path used to reach the current vertex to detect cycles and also rejects the graph if any vertex has more than one type variable or constructor vertex as a descendant. For efficiency it is also convenient to build up a list of vertices already visited and known to be acceptable, this prevents areas of the graph being traversed more than once.

5.3.2.2 Type Inference

The search function seen earlier is also used to read graphs. Recall that the result of searching is a set of vertices without edges from them, and a set of type constructor vertices. There are three possibilities for the type of a vertex:

- If there is exactly one type constructor vertex and no other vertices, then the type is formed from that type constructor (the rest of the type can be built recursively from the connection points).

- If there is a single other vertex (type variable, expression or connection point) then the type is a type variable labelled by that vertex.

- Otherwise there is no type (the graph represents some failure of type inference).

An algorithm for reading a type is in Figure 5.14, this takes a graph and vertex and produces a type. It will always terminate but will not give a type if there is any sort of conflict.

First search is used to find the significant vertices from the current vertex. If search finds only vertices with no children (and does not find any type constructors) then the type is a new type variable, and a relation is created relating this type variable to the set of vertices. If search finds only a single type constructor, then the type is formed by this constructor.

Function type checks for cycles and branching in the graph. This allows it to be used on graphs which do not represent valid typings (i.e. graphs which would be rejected by checkGraph).

Figure 5.14 Algorithm for reading a type.

Takes a graph, a vertex, and a set of vertices already seen (initially empty); returns the type associated with the vertex (if one exists).

$$
\begin{array}{lll}
\mathsf{type}((V,E),v,U) & = & \textbf{let} \\
 & & \quad \textbf{if } v \in U \textbf{ then terminate (cyclic type)} \\
 & & \textbf{case } \mathsf{search}((V,E),v) \\
 & & \quad (\{\},\{\},\{v\}) \Rightarrow \mathsf{mkTyvar}(v) \\
 & & \quad (\{\},\{\alpha\},\{\}) \Rightarrow \alpha \\
 & & \quad (\{(\circ_0 \cdots \circ_{n-1})c\},\{\},\{\}) \Rightarrow \\
 & & \qquad (\mathsf{type}((V,E),\circ_0,U) \cdots \mathsf{type}((V,E),\circ_{n-1},U))c \\
 & & \quad (cs,tvs,vs) \Rightarrow \textbf{terminate (conflict between constructors)}
\end{array}
$$

type alone is not sufficient, however, to see whether a graph represents a typing as it will not visit every vertex in the graph.

type does not treat the unbound type constructor specially. This allows it to produce types such as $\alpha \rightarrow \mathsf{unbound}$. As it stands, type is not suitable for reading the required types of unbound identifiers (such as I in Figure 5.7) as it will detect a conflict between unbound and the required type. It is clear that only a minor modification is required to ignore unbound, this modified algorithm is explored in Section 5.4.1

5.4 Generalising Techniques for Type Debugging

Several authors have presented algorithms to help programmers understand the types in their programs and to help debug type errors in Hindley-Milner based languages. These methods were mentioned in Chapter 3. Each of these has supplied a different form of information to the programmer: the types of unbound identifiers and suggestions of where in programs mistakes may lie are examples of such information.

This chapter has introduced a new means of representing type information as graphs. From this we can extract a range of different facts about the types in programs. So far we have seen how to find out whether or not the program is typed, and what its type is. The graph representation can also be used to simulate some of the schemes proposed by other authors.

Bernstein and Stark [BS95] chose to describe the *types* of unbound identifiers, while Mitchell Wand [Wan86] describes *locations* in the program which may contain a mistake and Duggan [Dug98] produces type *explanations*. We see that graphs encode the information produced by other authors. That is, that the information can be retrieved by a traversal of the graph. Hence, it is claimed that this work generalises a number of pieces of previous work. This is of practical use as it allows several different forms of information to be presented without repeated calculation.

5.4.1 Bernstein and Stark's Assumption Environments

Bernstein and Stark's system [BS95] is concerned with the types which unbound identifiers must have in order for a program to be well typed. The results of their inference algorithm can also be obtained by reading graphs.

5.4.1.1 Bernstein and Stark's Technique

Bernstein and Stark [BS95] wrote alternative operational type semantics for expressions with unbound identifiers. They also gave a deterministic inference algorithm for these semantics. For the open expression $(I\ 3, I\ \mathsf{true})$ Bernstein and Stark's algorithm derives the *assumption environment* $\{I : \{\mathsf{int} \rightarrow \alpha, \mathsf{bool} \rightarrow \beta\}\}$ and the type $\alpha \times \beta$. This means that in order for $(I\ 3, I\ \mathsf{true})$ to type check, it should be put in a context which gives I a type scheme which can be specialised to types $\mathsf{int} \rightarrow \alpha$ and $\mathsf{bool} \rightarrow \beta$, or the instances of I should be replaced by expressions with these types. The system can only produce assumption environments for internally consistent fragments (e.g. not $\lambda I.(I\ 3, I\ \mathsf{true})$).

To use Bernstein and Stark's work, first estimate the probable location of the type error (using a conventional error message), then examine parts of the program. This investigation reveals types of fragments and the types of their free identifiers. Bernstein and Stark's contribution is to overcome Milner and Damas's [DM82] limitation of not letting you look inside expressions to see the types of their subexpressions.

5.4.1.2 Extracting Assumption Environments from Graphs

The graph for the open expression $(I\ 3, I\ \mathsf{true})$ was shown in Figure 5.7. Two types can be read for each instance of I: either unbound or a function type. The presence of unbound means that Bernstein and Stark's analysis is relevant.

To generate an assumption environment from a graph, first locate all the vertices representing unbound identifiers. These are identified by edges going to unbound, labelled by the identifier. The types of these identifiers are then read as normal, only ignoring unbound. Figure 5.15 shows the algorithm taking a graph and vertex, and returning the type represented by the vertex in the graph.

Figure 5.15 Algorithm B-S-type.

Take a graph and vertex, return the type represented by that vertex, ignoring the unbound type constructor.

$$\mathsf{B\text{-}S\text{-}type}(G,v) = \mathsf{typeDFS}(G,v,\{\})$$

The DFS routine requires a list of vertices already seen. It will terminate without a result if it finds a cycle or branch (cf. Definition 1).

$\mathsf{typeDFS}(G,v,s) =$
  if $v \in s$ **then**
    Terminate (**cyclic type**)
  else
    let $(cs,vs,tvs) = \mathsf{search}(G,v)$
    and $cs' = \{c_i : c_i \in cs \wedge c_i \neq \mathsf{unbound}\}$
    case (cs',tvs,vs) **of**
      $(\{\},\{\},\{\}) \Rightarrow \alpha_v$
    $\mid (\{\},\{v'\},\{\}) \Rightarrow \alpha_{v'}$
    $\mid (\{\},\{\},\{\alpha\}) \Rightarrow \alpha$
    $\mid (\{(\circ_0 \dots \circ_{n-1})c_i\},\{\},\{\}) \Rightarrow$
      $(\mathsf{typeDFS}(G,c_i.0,\{v\}\cup s),\ \dots\ ,\mathsf{typeDFS}(G,c_i.(n-1),\{v\}\cup s))c_i$
    $\mid$ _ $\Rightarrow$ **Terminate** (**conflicting type**)

Applying B-S-type to the vertex representing the entire program in the graph for a correctly typed program will yield the same answer as type in Figure 5.14.

Bernstein and Stark's inference algorithm fails if no assumption environment exists. This is the case if the graph contains any branches representing type conflicts, or cycles. Before using B-S-type, the graph should be checked for cycles and branches, i.e. that it meets the first two conditions in Definition 1. Providing there are no cycles or branches, type will always succeed.

There may be a number of vertices for each identifier: each instance of the vertex will appear, possibly with different tags. The set of all types for all vertices corresponding to a particular identifier is the set of types required for the assumption environment. Reading the graph in Figure 5.7 by using type on I left and I right gives the assumption environment $\{I : \{\mathsf{int} \rightarrow \alpha, \mathsf{bool} \rightarrow \beta\}\}$.

Implementing Bernstein and Stark's technique using graphs allows us to give the programmer slightly more useful information as it is possible to relate types to particular instances of identifiers (Bernstein and Stark's presentation discards this information). Indeed, we can give the type of any instance of an identifier (even bound) or any other part of the program. It is also sometimes possible to use type even if the graph fails the first two conditions in Definition 1. This makes it more flexible than Bernstein and Stark's algorithm.

5.4.2 Wand's Source of Type Errors

Wand's concern [Wan86] is with the *location* of a mistake. He observed that the location announced in a type error message is rarely the location at which the programmer has made a mistake. This is attributed to the fact that "the type-checker can only report an error when it finds a program fragment that cannot be assigned a type; because of the flexibility introduced by polymorphism, the actual error may be deeply embedded in the erroneous fragment". Wand in fact understates the problem: sometimes the fragment can be assigned a type, and the actual error is in a separate fragment earlier in the program.

5.4.2.1 Finding the Source of Type Errors

Wand's modified unification algorithm allows type error messages to propose probable locations for the programmer's mistake. Types must be represented using *structure sharing*: a type is a tree with type variables for leaves, an *environment* binds type variables to types. The environment is essentially a substitution but it is always passed explicitly and type variables in types are never expanded. The environment also contains *reasons* which are sets of subexpressions associated with a binding of a type variable to a type. For example the binding $\alpha_I \mapsto \mathsf{int} \rightarrow \beta$ might have the reason $\{(I\ 3)\}$. The unification algorithm must do two things with reasons. When making a new binding a reason must be added, and when unification fails reasons for the failure should be given. A reason for each type is built up as the algorithm traverses the types: each time a type variable inside type τ is expanded using the environment, the reason from the environment is added to the reason for τ. When unification fails, a reason is returned for each type. The representation shares similarities with the graphs of this chapter.

Figure 5.16 shows the error message produced for the program $\lambda I.(I\ 3, I\ \mathsf{true})$. From it, the programmer can establish that the mistake is likely to be I true or I 3 (the possibility that I should be let-bound rather than λ-bound is not considered).

Figure 5.16 Wand's error message for $\lambda I.(I\ 3, I\ \mathsf{true})$.

Mismatch between int and bool
In expression I true
Reason for 1st type: $\{(I\ 3)\}$
Reason for 2nd type: $\{\}$.

Wand has implemented this in the SPS system, but it was abandoned as it was found to produce too much output to be useful. We include this method here because it shows one use of graphs and because it may be possible to produce more useful output from the reasons.

5.4.2.2 Using Graphs to Find Wand's Source

To find the probable source of a type error first identify a branch in the graph which gives two distinct types to a vertex. The graph for the program $\lambda I.(I\ 3, I\ \mathsf{true})$ was shown earlier in Figure 5.5, In this figure there is such a branch at the left-hand connection point of the bottom $\rightarrow$. A branch like this corresponds to a unification failure in type inference.

Having found a branch, we must find the reasons associated with it. These are the labels of its edges. In Figure 5.5, from the vertices above the branch we see that the expression being examined when the branch appeared was either I 3 or I true (depending on the order of type inference) and we also see the reasons for the mismatched type I true or I 3.

Wand's system suffered from a left-to-right bias which led it to treat I 3 and I true differently. One is given as the expression being examined, the other as a reason. Despite the slight differences between Wand's system and the graph: from the point of view of suggesting the 'source of type errors' they provide the same list of candidates.

We can also extract more information from the graph. We can see that not only is the mismatch associated with the application expression, but also that it is in the type of identifier I. Programming experience suggests that this information could be more useful than the site at which unification was performed.

5.4.3 Duggan's Correct Type Explanations

Dominic Duggan has formally defined the notion of *correct type explanation* [Dug98]. It should be possible to generate type explanations from graphs by traversing edges from a program fragment vertex to type constructor vertices and recording the labels on the edges. In generating correct explanations, some edges must be traversed backwards and some should not be traversed. The difficulty is in deciding which edges to follow and in which direction.

5.4.3.1 Correct Type Explanations

A type explanation is a set of expressions, like Wand's reasons. Duggan associates the expressions which build up explanations with *constraints* of the form $\tau_1 = \tau_2$. Each constraint is labelled by a set of expressions which explain how it was obtained. Duggan's constraints are unlike Wand's environments which constrain type variables to be equal to a type, rather than any two types to be equal. A set of constraints can be used to infer what some type is, for example what type some type variable α_I (associated with a λ-bound identifier I) is. The sets of expressions labelling the constraints used to infer a type form the explanation of that type.

An example of a constraint set and explanation is in Figure 5.17. The constraint equates the type variable, α_e, for each expression, e, with some other type. The explanation is the union of the labels on the constraints used to work out the type for a particular type variable.

Duggan defines two conditions which must be met for an explanation to be correct:

Completeness states that the explanation is large enough, i.e. no constraints with labels outside the set are required to infer the type.

Soundness states that the explanation is not too large, i.e. all of the labels belong to some constraint necessary to infer the type. This is the notion of minimal generating sets.

Figure 5.17 A constraint set and correct type explanation.

- Constraint set for $\lambda(i,x).(i\,x, i\,3)$

$$\{\alpha_i \overset{\{i\,x\}}{=} \alpha_x \rightarrow \alpha_{ix},\ \ \alpha_i \overset{\{i\,3\}}{=} \alpha_3 \rightarrow \alpha_{i3},\ \ \alpha_3 \overset{\{3\}}{=} \mathsf{int}\}$$

- Correct type explanation of $x : \mathsf{int}$

$$\{i\,x, i\,3, 3\}$$

If the expression $i\,x$ was not in an explanation of $x : \mathsf{int}$ then the explanation would be incomplete. If the same expression was in an explanation of $i : \mathsf{int} \rightarrow \alpha_{i3}$ then the explanation would be unsound (i.e. not minimal).

There can be a number of correct type explanations for a given subexpression. For example if i was applied to 2 and 3 then either application (but not both) would be acceptable for a correct explanation of its type. The explanation produced in inference depends on the inference strategy (top down or bottom up) and constraint solver used.

5.4.3.2 SML/E: an Implementation of Explanations

SML/E[1] is Dominic Duggan's implementation of type explanations as part of the Standard ML of New Jersey compiler. The current version (version 1.0) does not appear to meet its specification (it does not produce correct definitions according to [Dug98]).

For example, for the program

```
fun f i x = (i x, i 3) ;
```

the following explanation is given for the type of '`f.i`' (i.e. the argument `i` of function `f`)

```
The explanation for the type of f.i is as follows:

[0] f.i : 'A0  by Assumption

[1] f.i : 'A1 -> 'a  because...
```

[1]SML/E is available from `http://guinness.cs.stevens-tech.edu/~dduggan/Public/Smle/index.html`

```
'A0 (the type variable for f.i) was instantiated to 'A1 -> 'a as a result of
type-checking this expression:

i x

[2] f.i : int -> 'a  because...

'A1 (the type variable for f.x) was aliased to int (the type for 3)
as a result of type-checking this expression:

i 3
```

This explanation has a set of expressions $\{$`i x`,`i 3`$\}$. This explanation does not represent the minimal generating set as `i x` is unnecessary (knowing that `i` is applied to `3` is sufficient to know what type it has). Duggan has suggested that the definition of minimal generating sets may be too restrictive for practice[1].

It should also be noted that while correct type explanations are described as an aid to debugging type errors, SML/E cannot operate on untypeable programs (a standard SML/NJ type error message is given in such cases).

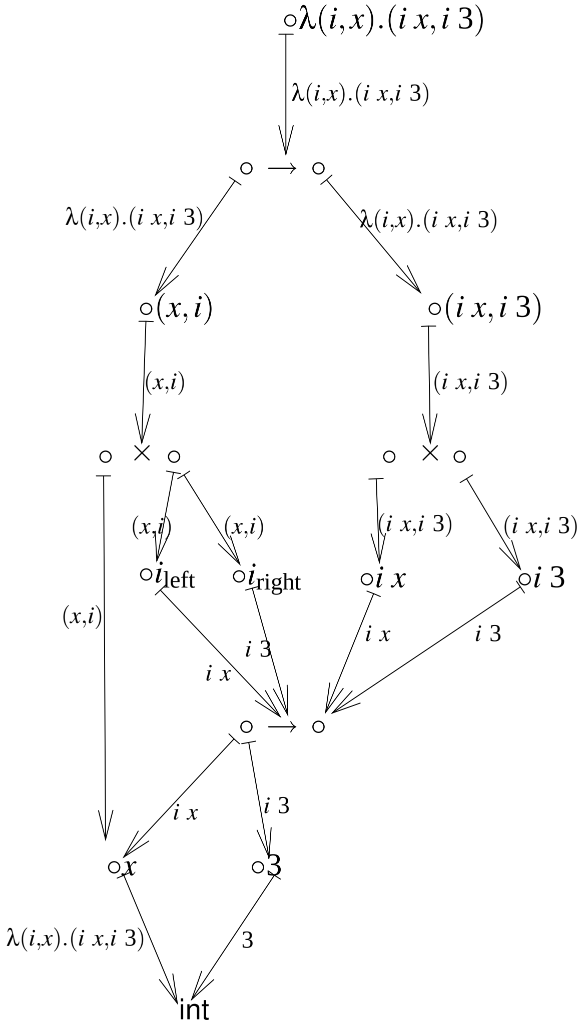
5.4.3.3 Reading Explanations from Graphs

Explanations from graphs are formed from the labels on edges. For example, in Figure 5.18 the explanation of the type of either occurrence of *i* must involve the corresponding application (*i x* or *i* 3) as this is the only edge which can be followed to discover a type. Explanations are built from the labels on the edges forming a path from a program fragment vertex to a type constructor vertex.

The main difficulty in creating an algorithm to extract explanations from graphs is that the paths to be followed are non-directed (you must traverse *up* edges as well as down). For example the correct type explanation of the type of *x* is $\{$`i x`,`i 3`,`3`$\}$ which must be formed from the path $x \overset{i\,x}{\hookleftarrow} \circ \overset{i\,3}{\mapsto} 3 \overset{3}{\mapsto} \mathsf{int}$ rather than the path $x \overset{\lambda(x,i)\ldots.}{\mapsto} \mathsf{int}$ (*cf.* Figure 5.17). Note that in this case, unlike in the case of the explanation of the type of `i`, both the subexpressions `i 3` and `i x` are necessary.

[1] Personal communication, 23rd March 1999.

Figure 5.18 Graph for $\lambda(x,i).(i\,x,i\,3)$.



Backtracking in this way is required because, in order to make them easier to read, graphs are directional. Ensuring a directed path exists from an expression to its type and that the graph is acyclic entails adding extra edges labelled by super-expressions of the expressions who's type they define. In this case the extra edge is labelled by $\lambda(i,x).(i\,x,i\,3)$ and comes from a vertex for x). These edges are added as side effects of these explicitly added edges, for example if we add an edge from v to int, and there is already an edge from v to v' then we must add an extra edge from v' to int to ensure v and v' are read as the same type. This process of adding edges performs the task of unification or constraint solving but does not record reasons. Reasons are not recorded because there may be a number of different valid reasons for adding the edge. These extra edges should not be followed when building an explanation.

Figure 5.19 shows the edges which should be followed to generate explanations. These are the edges which are explicitly added during graph generation.

Figure 5.19 Edges for explanations.

- $\{v_0 \overset{i}{\mapsto} v_1 : v_0, v_1 \in V\}$
  i.e. All edges labelled by an identifier.
-
  $$\begin{aligned}&\{[\lambda x.e]_t \overset{\lambda x.e}{\mapsto} \circ \rightarrow_i \circ,\\ &\rightarrow_i .0 \overset{\lambda x.e}{\mapsto} [x]_t,\\ &\rightarrow_i .1 \overset{\lambda x.e}{\mapsto} [e]_t :\\ &[\lambda x.e]_t \overset{\lambda x.e}{\mapsto} \circ \rightarrow_i \circ\}\end{aligned}$$

  i.e. Edges labelled with $\lambda x.e$ which are directly associated with the $\circ \rightarrow \circ$ for this expression (optionally tagged by some tag, t).
-
  $$\begin{aligned}&\{[e_0e_1]_t \overset{e_0e_1}{\mapsto} \rightarrow_i .1,\\ &\rightarrow_i .0 \overset{e_0e_1}{\mapsto} [e_0]_t,\\ &[e_1]_t \overset{e_0e_1}{\mapsto} \circ \rightarrow_i \circ :\\ &[e_0e_1]_t \overset{e_0e_1}{\mapsto} \rightarrow_i .1\}\end{aligned}$$

  i.e. Edges labelled with e_0e_1 which are directly associated with the $\circ \rightarrow \circ$ for this expression.
-
  $$\begin{aligned}&\{[\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]_t \overset{\mathsf{let}\ x=e_0\ \mathsf{in}\ e_1}{\mapsto} [e_1]_t,\\ &[x]_t \overset{\mathsf{let}\ x=e_0\ \mathsf{in}\ e_1}{\mapsto} [e_0]_t :\\ &[\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]_t \overset{\mathsf{let}\ x=e_0\ \mathsf{in}\ e_1}{\mapsto} [e_1]_t\}\end{aligned}$$

  i.e. Edges labelled with a let expression going from an instance of the let bound identifier to its defining expression, and from the let expression to its value expression.

Figure 5.20 gives the algorithm for generating type explanations. Essentially this is a cycle avoiding depth-first search which traverses only the edges given in Figure 5.19. explain takes the graph, vertex and path traversed so far (an initially empty set of edges); and returns the path from the vertex to the vertex defining its type, and this vertex. For the vertex representing the type of x in the graph in Figure 5.18, explain returns the int vertex and the edge set $\{x \overset{i\,x}{\hookleftarrow} \to .0,\ \to .0 \overset{i\,3}{\mapsto} 3,\ 3 \overset{3}{\mapsto} \mathsf{int}\}$. explain only gives the explanation as far as the main type constructor in a type, for example applying it to either instance of i in Figure 5.18 will return the path to the $\to$ vertex. To get a full explanation apply explain to each of the connection points of the type constructor vertex. This is analogous to the user interface of SML/E, which pauses after each type constructor in the type has been explained.

Figure 5.20 Algorithm explain.

$\mathsf{explain}(G, v, p) =$
  let $c = \mathsf{children}(G, v)$
  if $c = \{\}$ **then**
    $(\{\}, v)$
  else
    let $E =$ **Valid edges to or from** v **not in** p
    For each $v \overset{l}{\mapsto} v' \in E$
      if $\mathsf{explain}(G, v', \{e\} \cup p)$ **succeeds**
        let $(x, v'') = \mathsf{explain}(G, v', \{e\} \cup p)$
        return $(e \cup x, v'')$
      else try next edge
    if none succeed, then fail

By detouring around unwanted edges in different ways, it is possible to generate a range of different type explanations. These type explanations, however, are not necessarily *correct* in the sense described by Duggan as they may have extra expressions in them. For example for the explanation of the type of `i` in the example program, both `i x` and `i 3` appear. This means that the algorithm explain suffers from the same unsoundness problem was SML/E.

On a pragmatic note, while both SML/E and type are incomplete with respect to the paper [Dug98] there is no reason to suppose that it is not the definition rather than the implementations which are incorrect. The litmus test for this is whether or not the explanations help programmers (no such test has been conducted to my knowledge).

5.5 Implementation

All the algorithms given in this chapter have been implemented and tested for a small λ-with-let based language. The implementation was intended as a test-bed for ideas rather than a full-scale practical application of use on real programs. Source code is omitted here but can be found in [McA99a].

5.6 Conclusions

We have seen a way of representing the results of type inference as a graph. Both typeable and untypeable programs can be represented in this way — this is the first work to treat typeable and untypeable programs equally. Algorithms for generating and reading graphs have been given. From these graphs, we can extract a number of different forms of information about programs. The types of unbound identifiers as proposed by [BS95] and probable locations of mistakes in the style of [Wan86]. We also saw an algorithm to extract type explanations of the form of SML/E and noted that neither the algorithm for graphs, nor SML/E meet the requirements for *correct* type explanations in [Dug98]. The graphs therefore generalise these other forms of information.

Chapter 6

Repairing Mistakes with Type Isomorphisms

In the previous chapters of this thesis and in the work by others on type error messages [JW86, Wan86, BS93, Rit93a, BS95, DB96, Dug98, Yan97, Yan99, YMT00], the error messages and information produced relates to explaining where in a program the mistake may lie, and how the types which indicate the error were derived. This information is of help to the programmer in debugging the program, but is not of direct application. Recall the example of a spell-checker: the best way to help someone repair a mistake is to suggest ways for them to repair it.

In contrast to the other work, this chapter suggests a way to generate messages which do suggest how to correct mistakes. The technique is an application of the theory of *type isomorphisms* and has been implemented as part of the MLj compiler.

This chapter will first describe the intended content of the error messages (Section 6.1). It will then look at the theory behind the work (Section 6.2), the novel algorithms required to use the theory (Section 6.3), and the implementation (Section 6.4).

6.1 The Most Effective Message

The characteristics of effective error messages listed earlier in Section 2.1, originally by Jakob Nielson [Nie01], was to be explicit, human-readable, polite, precise and to give constructive advice. The emphasis in this chapter is on making a message which is human readable and which gives constructive advice.

To give constructive advice, the error message should suggest how the program can be modified in order to correct it. An example of such a message appears in Figure 6.1.

Figure 6.1 An error message offering constructive advice.

```
Try changing
  map ([1, 2, 3], Int.toString)
To
  map Int.toString [1, 2, 3]
```

The message in the figure is also human readable. It will be understood by the programmer as its content is all code which the programmer has written or which closely resembles the original code. The message would be less readable if, for example, it introduced new functions to uncurry `map` and exchange the arguments.

It might be suggested that if the compiler can work out a way of correcting the mistake then it could simply correct the mistake and issue a warning, without requiring the programmer to change the source code. This however is not desirable as there may be several ways to correct the mistake, not all of which may be found by the system. In particular if a function were given several arguments of the same type, then there would be a number of permutations of arguments all of which would type correctly. A simple example of such a mistake would be `compare a b`, in which the function is incorrectly used as if it were curried, but there are two possible ways to repair it (`compare (a, b)` or `compare (b, a)`). The error message could form part of an integrated development environment which would make it easy for the programmer to make a change to the program (in a similar way to a word processor making it easier to correct type error messages than using a command-line based spell-checker), but it would still be desirable for the programmer to confirm that changes should be made — witness the frustration experienced when typing unusual words into some word processors and having them corrected automatically. Edward Tenner warns that programs which make changes to data without the user's knowledge can reduce productivity [Ten96]. Donald Norman [Nor98] recommends against increasing the complexity of the external interface of software. The intent here is not to get the compiler to do a new task (repairing software) but to get it to do an existing job better (giving error messages).

6.2 Theory

Now that the content of the message has been established, let us look at the theory which will yield the results.

6.2.1 Function and Context Types

Reconsider the example in Figure 6.1. In it a programmer has applied a function to arguments which are in uncurried form rather than curried form as required and which are also in the wrong order. The type of the function, $\tau_{function}$, is

```
('a -> 'b) -> 'a list -> 'b list
```

but it appears in a context which suggests that it should have a type of the form $\tau_{context}$,

```
((int list) * (int -> string)) -> 'c
```

(for some `'c`). Type inference fails (in most algorithms) when the function and context types (or some other types) fail to unify, and most error messages describe the error in terms of unification failing.

The key to this technique is to note that while no substitution exists which makes the types *equal*, there is a substitution, $\{'\mathtt{a} \mapsto \mathtt{int}, '\mathtt{b} \mapsto \mathtt{string}\}$, which makes them similar:

```
(int -> string) -> int list -> string list
```

is similar to

```
(int list * (int -> string)) -> string list
```

The ‘similarity’ is that any value of one type can be transformed into a value of the other type by a simple function, and transformed back to the original value by a second simple function. The functions `m` and `m'`, defined by

```
fun m  map (theList, theFunction) = map theFunction theList

fun m' map theFunction theList = map (theList, theFunction)
```

will do the conversion for the example. The existence of these functions means that the types are *isomorphic*. Two types τ and τ' are isomorphic ($\tau \leftrightarrows \tau'$) iff there exist functions (morphisms) $m : \tau \to \tau'$ and $m' : \tau' \to \tau$ such that $\forall x : \tau . m'(mx) = x$ and $\forall x : \tau' . m(m'x) = x$. This means that the original types (the function and context types which failed to unify) are *unifiable modulo isomorphism*. That is, there exist a substitution S and morphisms m and m' such that $\forall x : S\tau . m'(m\,x) = x$ and $\forall x : S\tau' . m(m'\,x) = x$. For information on isomorphisms, see [Di 95].

One of the morphisms has an important property. If we apply the morphism which has type of the form $\tau_{function} \to \tau_{context}$ to the function in the incorrect expression, then the function will fit its context and there will be no type error. E.g.

```
m map ([1, 2, 3], Int.toString))
```

is a correctly typed program. We will call morphisms with type of the form $\tau_{function} \to \tau_{context}$ *repair morphisms*.

This new expression could form the basis of a *constructive* error message (it does suggest a way to remove errors), but lacks readability as the programmer must understand the definition of `m` in order to decide whether or not this is the correct way to repair the program. In order to produce the advice in Figure 6.1, the application of `m` must be partially evaluated.

A method for producing constructive readable messages for some mistakes is, therefore

- Detect an application expression in which there is an error.
- Identify the function and context types.
- Unify the function and context types, modulo isomorphism. Generate the repair morphism while doing this.
- Apply the repair morphisms to the original expression, and partially evaluate.

The remainder of this chapter is about the algorithms required to do this (in particular generating the repair morphism is novel work), and the implementation of the method in the MLj compiler.

6.2.2 The Type Language

The type language in this chapter requires type variables, function types, constructors of arbitrary arity, and tuples of arbitrary size. These features are also present in Standard ML. The syntax for such types is given in Figure 6.2. The unit type is a tuple with no fields, and a tuple with one field is included for completeness' sake (similarly, unary tuples exist in Standard ML, but are not used in practice).

Figure 6.2 Syntax for types.

τ	$::=$	α	Type variables
	$\mid$	$(\tau_1, \ldots, \tau_n)c$	Types formed from n-ary type constructor c
	$\mid$	$\tau \to \tau'$	Function types
	$\mid$	$\langle\rangle$	The unit type
	$\mid$	$\langle\tau\rangle$	Unary tuple
	$\mid$	$\langle\tau_1 \times \ldots \times \tau_n\rangle$	Tuple of size n

Note that parametric polymorphism and type schemes play no part in this work. The types to be unified will be instantiations of polymorphic type schemes.

6.2.3 Location

For this chapter is is assumed that the correct location at which the repair must be made has been correctly identified. In existing implementations of type inference this is where W or M fails. This is the correct location for some mistakes, but for others it is known that the error is detected too late and the wrong location announced.

Other authors have proposed means for finding the correct location of mistakes, so this issue is not discussed further here [LY98, Wan86, DB96, Chi01].

6.3 Algorithms

Two novel algorithms are required to generate the new advice for the error message. Firstly, an algorithm to generate the repair morphism is presented in Section 6.3.1. Once the repair morphism has been generated, we also need to partially evaluate its application to obtain the modified expression. Partial evaluation is described in Section 6.3.2.

6.3.1 Unification Modulo Linear Isomorphism

There are a number of different theories of isomorphisms, each characterised by a different set of axioms and a different set of possible morphisms. Di Cosmo makes a comprehensive review of these in [Di 95]. It is known that unification is not decidable for most theories, but that it is decidable for *linear isomorphisms* [NPS93].

Axioms defining the linear isomorphism relation are shown in Figure 6.3. The axioms are annotated with the morphisms above and below the $\leftrightarrows$. Other axioms for isomorphism exist, shown in Figure 6.4. Before commencing with the unification algorithm, it will be worthwhile checking that the linear theory is appropriate to program repair. The currying axiom corresponds to a programmer mistakenly using an uncurried function as curried, or *vice versa* — the example already used in this chapter shows that this axiom is applicable to debugging. Likewise, the example shows that commutativity is useful. Associativity is of direct use when functions take nested tuples as arguments, and is also essential for deriving more complex isomorphisms (for example it is necessary to produce a three-argument version of the uncurry morphism). Unary associativity is also of use for deriving other isomorphisms. Unit currying plays an important role for programs written in a lazy style, as it shows that a lazy value is isomorphic to a strict one. Of the non-linear axioms, elimination would be applicable to some errors involving imperative functions, and distributivity represents using two functions where one would do. Repairing errors with either non-linear axiom involves either deleting or copying source-code, which seem to be things a programmer is less likely to forget to do. This intuitively suggests that the non-linear axioms are not as relevant to debugging.

Narendran, Pfenning and Statman [NPS93] given an algorithm for unification modulo linear isomorphism which has two phases: rewriting to uncurry all functions and flatten all tuples (i.e. to deal with the first three axioms), and then associative-commutative (AC) unification (for the final axiom). This is shown as a commuting diagram in Figure 6.5. Any AC-unification algorithm may be used and no method of generating morphisms is given (the paper is concerned with deciding unify-ability, not with generating the witnesses).

Importantly, AC-unification can have many solutions and hence there may be many different ways to repair found.

Linear isomorphisms do not work for some mistakes. E.g. a value may be used where a list of values is required, this is easy to do by missing out square brackets. In the algorithms following, it is suggested how some "pseudo-isomorphisms" like this can be dealt with.

Figure 6.3 Axioms for linear isomorphisms.

Currying

$$\tau \to \tau' \to \tau'' \quad \underset{\lambda f.\lambda x.\lambda y.f\langle x,y\rangle}{\overset{\lambda f.\lambda\langle x,y\rangle.fxy}{\rightleftarrows}} \quad (\tau \times \tau') \to \tau''$$

Unit currying

$$\langle\rangle \to \tau \quad \underset{\lambda x\lambda\langle\rangle.x}{\overset{\lambda f.f\langle\rangle}{\rightleftarrows}} \quad \tau$$

Associativity

$$\langle\langle\tau_{1,1} \times \ldots \times \tau_{1,m_1}\rangle \times \ldots \times \langle\tau_{n,1} \times \ldots \times \tau_{n,m_n}\rangle\rangle$$
$$\lambda\langle\langle x_{1,1},\ldots,x_{1,m_1}\rangle,\ldots,\langle x_{n,1},\ldots,x_{n,m_n}\rangle\rangle.\langle x_{1,1},\ldots \times x_{1,m_1},\ldots,x_{n,1},\ldots,x_{n,m_n}\rangle$$
$$\leftrightarrows$$
$$\lambda\langle x_{1,1},\ldots \times x_{1,m_1},\ldots,x_{n,1},\ldots,x_{n,m_n}\rangle.\langle\langle x_{1,1},\ldots,x_{1,m_1}\rangle,\ldots,\langle x_{n,1},\ldots,x_{n,m_n}\rangle\rangle$$
$$\langle\tau_{1,1} \times \ldots \times \tau_{1,m_1} \times \ldots \times \tau_{n,1} \times \ldots \times \tau_{n,m_n}\rangle$$

Unary Associativity

$$\langle\tau\rangle \quad \underset{\lambda x.\langle x\rangle}{\overset{\lambda\langle x\rangle.x}{\rightleftarrows}} \quad \tau$$

Commutativity

$$\langle\tau_1 \times \ldots \times \tau_i \times \ldots \tau_n\rangle$$
$$\lambda\langle x_1,\ldots,x_i,\ldots x_n\rangle.\langle x_i,x_1,\ldots x_{i-1},x_{i+1},\ldots,x_n\rangle$$
$$\rightleftarrows$$
$$\lambda\langle x_i,x_1,\ldots x_{i-1},x_{i+1},\ldots,x_n\rangle.\langle x_1,\ldots,x_i,\ldots x_n\rangle$$
$$\langle\tau_i \times \tau_1 \times \ldots \tau_{i-1} \times \tau_{i+1} \times \ldots \times \tau_n\rangle$$

Figure 6.4 Non-linear isomorphisms axioms.

Elimination

$$\tau \to \langle\rangle \quad \underset{\lambda\langle\rangle.\lambda y.\langle\rangle}{\overset{\lambda f.\langle\rangle}{\leftrightarrows}} \quad \langle\rangle$$

Distribution

$$\tau \to \langle \tau' \times \tau'' \rangle \quad \underset{\lambda\langle f,g\rangle.\lambda x.\langle f\ x, g\ x\rangle}{\overset{\lambda f.\langle \lambda x.(\lambda\langle a,b\rangle.a)(f\ x).\lambda x.(\lambda\langle a,b\rangle.b)(f\ x)\rangle}{\leftrightarrows}} \quad \langle \tau \to \tau' \times \tau \to \tau'' \rangle$$

Figure 6.5 The algorithm as a commuting diagram.

$$\begin{tikzcd}
\tau \arrow[r, "f"] \arrow[d, "g' \circ h \circ f"'] & \langle \tau_1 \times \ldots \times \tau_n \rangle \arrow[d, "h"] \\
\tau' & \langle \tau'_1 \times \ldots \times \tau'_n \rangle \arrow[l, "g'"]
\end{tikzcd}$$

Normal forms of τ *and* τ'

f and g' are produced from rewriting, h is from AC-unification.

6.3.1.1 The Rewriting Phase

The rewriting phase is directed by the axioms. Building up the complete morphism is not difficult since each axiom is annotated by a morphism. The morphisms are constructed from the annotations on the axioms, and from 'map' functions. Type constructors, c, with arity n, may have a map function

$$\mathtt{map}_c : \langle(\alpha_1 \to \beta_1) \times \ldots \times (\alpha_n \to \beta_n)\rangle \to (\alpha_1, \ldots, \alpha_n)c \to (\beta_1, \ldots, \beta_n)c$$

an example of such a function is Standard ML's `List.map`. There is also a map for every arity, i, of tuple

$$\begin{aligned} \mathtt{map}_{\times i} &: \langle \alpha_1 \to \beta_1 \times \ldots \times \alpha_i \to \beta_i \rangle \to \langle \alpha_1 \times \ldots \times \alpha_i \rangle \to \langle \beta_1 \times \ldots \times \beta_i \rangle \\ &= \lambda\langle m_1, \ldots, m_i\rangle.\lambda\langle a_1, \ldots a_i\rangle.\langle m_1\ a_1, \ldots, m_i\ a_i\rangle \end{aligned}$$

When the arity of $\mathtt{map}_{\times i}$ is clear, it will be denoted simply by $\mathtt{map}_{\times}$. Finally we must have a map for functions, defined as

$$\mathtt{map}_{\to} : \langle(\beta \to \alpha) \times (\alpha' \to \beta')\rangle \to (\alpha \to \alpha') \to (\beta \to \beta') = \lambda(m, m').\lambda f.m' \circ f \circ m$$

Note that the type of the map for functions differs from the standard form of map functions as its first argument function has type $\beta \to \alpha$ instead of $\alpha \to \beta$.[1]

Some user defined abstract types may not have a map function, for example a binary search tree may not be equipped with one. If there is no map for a constructor then there is no isomorphism (apart from equality) over types formed by that constructor.

Morphisms for permuting one tuple into another are also required. This operation is seen in the associativity and commutativity axioms. These morphisms are $\lambda p.p'$, where p and p' are patterns according to the syntax in Figure 6.6.

The syntax of morphisms is summarised in Figure 6.7.

There are two functions required for rewriting: one to flatten tuples (following the associativity axiom) and one to apply the currying axioms. These are in Figures 6.8 and 6.9.

[1] We could require every map to take pairs of complementary morphisms, in which case all the maps could take the same standard form. This would make it easier to find a morphisms complement, but would also make the algorithms harder to read and would not be necessary for the implementation.

Figure 6.6 Patterns.

$$
\begin{array}{rcl}
p & ::= & x \\
& | & \langle p_1, \ldots, p_n \rangle
\end{array}
$$

There is a @ operator for appending patterns.

$$
\begin{array}{rcl}
x@\langle p_1, \ldots, p_n \rangle & = & \langle x, p_1, \ldots, p_n \rangle \\
\langle p_1, \ldots, p_n \rangle @x & = & \langle p_1, \ldots, p_n, x \rangle \\
\langle p_1, \ldots, p_m \rangle @\langle p'_1, \ldots, p'_n \rangle & = & \langle p_1, \ldots, p_m, p'_1, \ldots, p'_n \rangle
\end{array}
$$

and a corresponding operator for types

$$
\begin{array}{rcll}
\tau@\langle \tau_1 \times \ldots \times \tau_n \rangle & = & \langle \tau \times \tau_1 \times \ldots \times \tau_n \rangle & \tau \text{ is not a tuple type.} \\
\langle \tau_1 \times \ldots \times \tau_n \rangle @\tau & = & \langle \tau_1 \times \ldots \times \tau_n \times \tau \rangle & \tau \text{ is not a tuple type.} \\
\langle \tau_1 \times \ldots \times \tau_m \rangle @\langle \tau'_1 \times \ldots \times \tau'_n \rangle & = & \langle \tau_1 \times \ldots \times \tau_m \times \tau'_1 \times \ldots \times \tau'_n \rangle &
\end{array}
$$

Figure 6.7 Linear Morphisms.

$$
\begin{array}{rcl}
m & ::= & \lambda p.p' \\
& | & \texttt{appUnit} \mid \texttt{mkLazy} \\
& | & \texttt{curry} \mid \texttt{uncurry} \\
& | & \texttt{map}_{\times}\langle m_1, \ldots, m_n \rangle \\
& | & \texttt{map}_{c}\langle m_1, \ldots, m_n \rangle \\
& | & \texttt{map}_{\rightarrow}\langle m, m' \rangle \\
& | & m_1 \circ \ldots \circ m_n \\
& | & \texttt{I}
\end{array}
$$

In $\lambda p.p'$, no identifier occurs more than once in either pattern, and the set of identifiers is the same for the two patterns. This ensures linearity (every input the the morphism occurs exactly once in its output).

Figure 6.8 Flattening.

$\text{FLATTEN}^M : type \rightarrow (type \times morphism \times morphism)$

$$\begin{array}{rcl} \text{FLATTEN}^M(\tau) & = & \textbf{let} \\ & & (\tau', p, p') = \text{MKPATTERN}(\tau) \\ & & \textbf{in } (\tau', \lambda p.p', \lambda p'.p) \end{array}$$

$\text{MKPATTERN} : type \rightarrow (type \times pattern \times pattern)$

The first pattern returned matches the argument type, the second is for the flattened type.

$$\begin{array}{rcl} \text{MKPATTERN}(\langle \tau_1 \times \tau_1 \times \ldots \times \tau_n \rangle) & = & \textbf{let} \\ & & (\tau'_1, p_1, p'_1) = \text{MKPATTERN}(\tau_1) \\ & & \vdots \\ & & (\tau'_n, p_n, p'_n) = \text{MKPATTERN}(\tau_n) \\ & & \textbf{in } (\tau'_1 @ \ldots @ \tau'_n, \langle p_1, \ldots, p_n \rangle, p'_1 @ \ldots @ p'_n) \\ \text{MKPATTERN}(\alpha) & = & (\langle \alpha \rangle, v, \langle v \rangle) \text{ for new } v \\ \text{MKPATTERN}((\tau_1, \ldots, \tau_n)c) & = & (\langle (\tau_1, \ldots, \tau_n)c \rangle, v, \langle v \rangle) \text{ for new } v \\ \text{MKPATTERN}(\tau \rightarrow \tau') & = & (\langle \tau \rightarrow \tau' \rangle, v, \langle v \rangle) \text{ for new } v \end{array}$$

Figure 6.9 Rewriting.

$\textsc{Rewrite}^M : type \to (type \times morphism \times morphism)$

$$
\begin{array}{l}
\textsc{Rewrite}^M(\tau_0 \to \tau_1) = \\
\quad \textbf{let} \\
\qquad (\tau_0', m_0, m_0') = \textsc{Rewrite}^M(\tau_0) \\
\qquad (\tau_1', m_1, m_1') = \textsc{Rewrite}^M(\tau_1) \\
\qquad (\tau, m, m') = \textsc{AppRule}(\tau_0' \to \tau_1') \\
\quad \textbf{in } (\tau,\ m \circ \texttt{map}_{\to}\langle m_0', m_1\rangle,\ \texttt{map}_{\to}\langle m_1', m_0\rangle \circ m')) \\
\textsc{Rewrite}^M(\langle \tau_1 \times \ldots \times \tau_m\rangle) = \\
\quad \textbf{let} \\
\qquad (\langle \tau_1' \times \ldots \times \tau_n'\rangle, m, m') = \textsc{Flatten}^M(\langle \tau_1 \times \ldots \times \tau_m\rangle) \\
\qquad (\tau_1'', m_1, m_1') = \textsc{Rewrite}^M(\tau_1') \\
\qquad \vdots \\
\qquad (\tau_n'', m_n, m_n') = \textsc{Rewrite}^M(\tau_n') \\
\quad \textbf{in } (\langle \tau_1'' \times \ldots \times \tau_n''\rangle,\ \mathsf{map}_{\times}\langle m_1,\ \ldots, m_n\rangle \circ m,\ m' \circ \mathsf{map}_{\times}\langle m_1', \ldots, m_n'\rangle) \\
\textsc{Rewrite}^M((\tau_1, \ldots \tau_n)c) \text{ (when } c \text{ has a map)} = \\
\quad \textbf{let} \\
\qquad (\tau_1', m_1, m_1') = \textsc{Rewrite}^M(\tau_1) \\
\qquad (\tau_n', m_n, m_n') = \textsc{Rewrite}^M(\tau_n) \\
\quad \textbf{in } ((\tau_1', \ldots \tau_n')c,\ \texttt{map}_c\ \langle m_1, \ldots, m_n\rangle,\ \texttt{map}_c\ \langle m_1', \ldots, m_n'\rangle) \\
\textsc{Rewrite}^M(((\tau_1, \ldots \tau_n)c) \text{ (when } c \text{ has no map)} = \\
\quad ((\tau_1, \ldots \tau_n)c,\ \texttt{I},\ \texttt{I}) \\
\textsc{Rewrite}^M(\alpha) = (\alpha, \texttt{I}, \texttt{I})
\end{array}
$$

$\textsc{AppRule}^M : type \to (type \times morphism \times morphism)$

$$
\begin{array}{l}
\textsc{AppRule}^M(\tau_0 \to (\tau_1 \to \tau_2)) = \\
\quad \textbf{let} \\
\qquad (\tau_a, m, m') = \textsc{AppRule}^M(\tau_0 \times \tau_1) \\
\quad \textbf{in} \\
\qquad (\tau_a \to \tau_2, \\
\qquad \mathsf{uncurry} \circ \texttt{map}_{\to}(m', \lambda i.i), \\
\qquad \texttt{map}_{\to}(m, \lambda i.i) \circ \texttt{curry}) \\
\textsc{AppRule}^M(\langle\rangle \to \tau) = \\
\quad (\tau,\ \lambda f.f\langle\rangle,\ \lambda x.\lambda\langle\rangle.x)
\end{array}
$$

The first case of $\textsc{AppRule}$ handles currying, and the second handles the curry and unit axiom.

6.3.1.2 The Unification Phase

The most important step in unification modulo linear isomorphism is associative commutative unification. There are two existing algorithms for this: the original complete algorithm by Mark Stickel [Sti75] and Lincoln and Christian's simpler, but incomplete algorithm [LC89]. Neither of these algorithms generates morphisms.

AC unification differs from equality unification in that it can have an arbitrary number of solutions rather than a unique most general solution. For example $\langle\alpha\times\beta\rangle =_{AC} \langle a\times b\rangle$ has two solutions, $\alpha = a \wedge \beta = b$ and $\alpha = b \wedge \beta = a$. Because of this, the AC-unification algorithms must return sets of results (each result being a substitution). Each solution has at least one morphism associated with it. There may be more than one morphism for a given substitution, e.g. for $\langle a\times a\rangle =_{AC} \langle a\times a\rangle$ has only one substitution $\alpha = a$, but two morphisms $\lambda\langle x,y\rangle.\langle x,y\rangle$ and $\lambda\langle x,y\rangle.\langle y,x\rangle$.

Stickel's algorithm reduces the problem to solving linear diophantine equations. For example, the solutions to the problem $a\times b\times\alpha\times b =_{AC} \beta\times a\times c$ are characterised by the set of solutions to the linear diophantine equation $a+2b+\alpha = a+c+\beta$. The solution $\alpha = c\wedge\beta = 2b$ corresponds to the substitution $\{\alpha\mapsto c, \beta\mapsto 2b\}$. Stickel does not specify a particular scheme for solving the equations so there is no direct way to generate morphisms in the algorithm.

Lincoln and Christian claim to "slay the diophantine dragon" in their paper. Their method resembles Stickel's but they generate simpler equations which have only 1 and 0 as resulting coefficients. The substitutions are produced by generating boolean matrices representing particular solutions.

The algorithm here is based on Lincoln and Christian's method, but differs in significant ways, primarily in the form of its recursive calls. Interested readers are recommended to consult Lincoln and Christian's paper for more information on the original.

The algorithm has four functions. These take equations, or conjunctions of equations; and return sets of results in which each single result contains one or more morphisms and a substitution. The function names are labelled with -$^{M}_{AC}$ to say that they are for AC unification and generate morphisms.

- $\textsc{Unify}^{M}_{AC}$ takes an equation and returns a complete set of unifiers (each unifier is a morphism and substitution).

- $\textsc{UnifyConjunction}^{M}_{AC}$ takes a conjunction of equations and returns a set each element of which is a substitution and morphism list. Each morphism in a list corresponds to one of the equations. This is used by UNIFY^M_{AC} for recursive calls (e.g. the two recursive cases in $(\tau_0, \tau_1)c =_{AC} (\tau'_0, \tau'_1)c$ are so $\tau_0 =_{AC} \tau'_0 \wedge \tau_1 =_{AC} \tau'_1$ or $\tau_0 =_{AC} \tau'_1 \wedge \tau_1 =_{AC} \tau'_0$).

- $\text{UNIFYWITHSET}^M_{AC}$ takes a set of substitution and morphism lists (representing the many solutions to previously solved parts of the original equation) and a conjunction of equations. It returns a set of substitutions and morphism lists. This is used to handle multiple results.

- $\text{MATRIXSOLVE}^M_{AC}$ takes a conjunction of equations and internally generates boolean matrices representing possible solutions. It then returns a set whose elements are four-tuples of a substitution, two morphisms (pre- and post- processing) and a list of equations.

The type of my $\text{MATRIXSOLVE}^M_{AC}$ differs from Lincoln and Christian's, in that they only return a substitution and not a conjunction of equations. This means that there are more complex recursive calls in my algorithm. This was necessary in order to find the components of the morphism for subterms of the type, in order to build the complete morphism. For example, to unify $\langle \tau_0 \times \tau_1 \rangle =_{AC} \langle \tau'_0 \times \tau'_1 \rangle$ it is necessary to record the morphisms corresponding to equations such as $\langle \tau_0 =_{AC} \tau'_0 \rangle$. The algorithms for the functions follow.

Function UNIFY^M_{AC} This is the entry function to be called by the type debugging system. It has type $equation \rightarrow \langle morphism \times substitution \rangle set$. For example given $a \times b =_{AC} \beta \times a$ as an argument, it will produce a set containing $\langle \lambda(x,y).(y,x), \{\beta \mapsto b\} \rangle$.

The algorithm follows a similar structure to conventional equality unification except in the case of tuple types. The algorithm in ML pattern matching style can be seen in Figure 6.10.

The tuple case (perhaps unsurprisingly) looks rather complicated. It follows the following steps

- Remove terms common to the two tuple types with REMOVEDUPLICATES. The common types are in tuple τ_D and the uncommon ones (unique to one of the original tuples) are in τ_U and τ'_U. We also have two morphisms $m_D : \tau \rightarrow \langle \tau_D \times \tau_U \rangle$ and $m'_D : \langle \tau_D \times \tau'_U \rangle \rightarrow \tau'$. E.g. for $\tau = \langle b \times a \rangle$ and $\tau' = \langle a \times c \rangle$, $\tau_D = \langle a \rangle$, $\tau_U = \langle b \rangle$, $\tau'_U = \langle c \rangle$.
  There may be more than one way to remove duplicates each with its own morphisms, for example in $\langle a \times \alpha \rangle =_{AC} \langle a \times a \times \beta \rangle$. The algorithm as presented (and as implemented)

does not take this into account. See Section 6.3.1.3 for information on how this affects the algorithm's results.

- Sort each of the types into constant, term, variable order (required for MATRIXSOLVE). The sorted types are τ_S and τ'_S. We have morphisms $m_{sort} : \tau_U \rightarrow \tau_S$ and $m'_{sort} : \tau'_S \rightarrow \tau'_U$. A constant is a nullary type constructor (it can only be unified with itself or a type variable), and a term is any constructed type. Any arbitrarily selected sorting will work, and hence implementation of SORT is simple.

- Apply MATRIXSOLVE to τ_S and τ'_S. This results in a set, R, containing tuples $(S, m_{pre}, m_{post}, C)$ (a substitution, pre- and post-processing morphisms and conjunction of equations).

- For each entry in the set R, solve the conjunction under the substitution, using UNIFY-WITHSET$^M_{AC}$. This results in a set, each element of which is list of morphisms, m, and a new substitution, S'.

- Build the set of results. Each morphism has to: remove the common types; then for the uncommon types: sort, pre-process, do the morphisms from solving the conjunction, post-process and unsort; then reinsert the common types.

Note also that in the function case of the algorithm, the call to UNIFYCONJUNCTION$^M_{AC}$ is with $\tau_0 =_{AC} \tau'_0 \wedge \tau'_1 =_{AC} \tau_1$. The second conjunct has the types in the opposite order so that the generated morphisms are suitable for $\mathtt{map}_{\rightarrow}$.

The algorithm could be modified to allow non-reversible "pseudo-isomorphisms", such as transforming values into lists. At the type constructor cases, if the types cannot be unified in any way (and empty set of unifiers is obtained) then the types can be checked to see whether they match any other mistakes, if this is the case then a unifier and a morphisms could be generated. For example if τ and τ' list cannot be unified but τ and τ' can be unified giving substitution S and morphism m then the S and $\mathtt{mkList} \circ m$ may be returned.

Functions UNIFYWITHSET$^M_{AC}$ and UNIFYCONJUNCTION$^M_{AC}$ These functions handle multiple recursive calls (i.e. conjunctions of equations).

UNIFYCONJUNCTION$^M_{AC}$ is similar to UNIFY$^M_{AC}$, but takes a conjunction of equations, and each result in the set contains a list of morphisms (one for each equation).

For example the result of $\textsc{UnifyConjunction}^M_{AC}(a \times b =_{AC} b \times a \wedge c =_{AC} c)$ will include the list of morphisms $[\lambda\langle x,y\rangle.\langle y,x\rangle,\ \lambda x.x]$.

$\textsc{UnifyWithSet}^M_{AC}$ is similar, but it takes a set of previous solutions as well as a conjunction. Its job is to try every possible way of solving the conjunction with every possible previous solution it is given. It has type $(substitution \times morphism\ list)set \times conjunction \rightarrow (substitution \times morphism\ list)set$.

The definitions for these two functions are in Figure 6.11.

Function $\textsc{MatrixSolve}^M_{AC}$ Matrix solve determines how the two types being unified could be rearranged according to the associativity and commutativity axioms. Its type is
$equation \rightarrow (substitution \times morphism \times morphism \times conjunction)set$.

$\textsc{MatrixSolve}^M_{AC}$ differs from Lincoln and Christian's version as they generated only a substitution rather than a substitution and conjunction of equations. Where the equation $\tau =_{AC} \tau'$ is produced in $\textsc{MatrixSolve}^M_{AC}$, Lincoln and Christian would produce the substitution $\{\alpha \mapsto \beta, \alpha' \mapsto \beta\}$ and solve the equations $\alpha =_{AC} \tau, \alpha' =_{AC} \tau'$. The problem is that these equations do not directly correspond to the generation of morphisms.

As a simple example of the output, consider $b \times b \times \alpha =_{AC} c \times \beta$. This has a preprocessing morphism to rearrange the tuple: $\lambda\langle a,b,c\rangle.\langle c,a,b\rangle$; a post processing morphism to build the term of type β: $\lambda\langle c,a,b\rangle.\langle c,\langle a,b\rangle\rangle$; a substitution $\{\beta \mapsto \langle \beta_1 \times \beta_2\rangle\}$; and a conjunction of equations $\beta_1 = b \wedge \beta_2 = b \wedge \alpha = c$. The equations are to be solved by $\textsc{UnifyConjunction}^M_{AC}$ and the morphisms relating to that refer to each equation and should be executed between the pre- and post-processing morphisms.

Rather than giving a full algorithm as we saw for the other function, we sketch how the matrices are generated, and how they are converted to pre- and post-morphisms (with substitutions and conjunctions) by looking at an example.

Matrices are generated exactly as in Lincoln and Christian's algorithm. Below is an example based on solving $\langle b \times (\langle d \times e\rangle \rightarrow f) \times \beta\rangle =_{AC} \langle a \times c \times (\langle e \times d\rangle \rightarrow \gamma)\rangle \times \alpha$. Every way of filling a boolean matrix that does not cause inherent inconsistencies (for example marking a constant to be unified with a term) is produced. Some may have inconsistencies which cause them to be eliminated later when the resulting equations are solved. This is why the algorithm produces sets of results.

	a	c	$\langle e \times d \rangle \to \gamma$	α
b				✓
$\langle d \times e \rangle \to f$			✓	
β	✓	✓		

Lincoln and Christian directly translate a matrix into a substitution by introducing new type variables (one for every type making up the tuples being unified). This is not possible when generating morphisms. Instead we must first introduce a new step. If a type variable, β, has more than one 'tick' in its row or column, we must create a new type variable for each tick, say $\beta_1, \ldots, \beta_n$. The ticks are distributed between the new type variables, and a substitution, $\{\beta \mapsto \beta_1 \times \ldots \times \beta_n\}$, is generated. Two morphisms are also generated: one to 'split' the row labels, and one to 'join' the column labels. Here is the result of splitting the variables for the matrix above.

	a	c	$\langle e \times d \rangle \to \gamma$	α
b				✓
$\langle d \times e \rangle \to f$			✓	
β_1	✓			
β_2		✓		

This gives substitution $S = \{\beta \mapsto \langle \beta_1 \times \beta_2 \rangle\}$, split morphism $m_{split} = \lambda\langle v, w, \langle y, z\rangle\rangle.\langle w, x, y, z\rangle$ and join morphism $m_{join} = \lambda\langle w, x, y, z\rangle.\langle w, x, y, z\rangle$ (as none of the column headings were split).

Now the matrix has one tick in every row and column. The morphism for rearranging, $m_{rearrange}$, and conjunction of equations remaining to be solved, C, are easy to generate now.

$$
\begin{aligned}
C \quad = \quad & \beta_1 =_{AC} a \\
\wedge \quad & \beta_2 =_{AC} c \\
\wedge \quad & \langle d \times e \rangle \to f =_{AC} \langle e \times d \rangle \to \gamma \\
\wedge \quad & b =_{AC} \alpha
\end{aligned}
$$

$$
m_{rearrange} \quad = \quad \lambda\langle w, x, y, z\rangle.\langle y, z, x, w\rangle
$$

In the final result of MATRIXSOLVE, the pre-processing morphism is $m_{rearrange} \circ m_{split}$ and post-processing morphism is m_{join}. Morphisms will be generated when the conjunction is solved (in the example, this involved one non-trivial unification), and these are composed in between the pre-and post-processing morphisms.

Figure 6.10 Algorithm for function $\textsc{Unify}^M_{AC}$.

$\textsc{Unify}^M_{AC}$: *equation* $\rightarrow$ (*morphism* $\times$ *substitution*)*set*

$$\textsc{Unify}^M_{AC}(\alpha =_{AC} \tau) = \{(\lambda x.x, \{\alpha \mapsto \tau\})\}$$
$$\textsc{Unify}^M_{AC}(\tau =_{AC} \alpha) = \quad \{(\lambda x.x, \{\alpha \mapsto \tau\})\}$$

$\textsc{Unify}^M_{AC}((\tau_1, \ldots \tau_n)c =_{AC} (\tau'_1, \ldots, \tau'_n)c))$ (when c has no map function) $=$
$\quad \{(\lambda x.x, \textsc{Unify}_=((\tau_1, \ldots \tau_n)c = (\tau'_1, \ldots, \tau'_n)c))\}$

$\textsc{Unify}^M_{AC}((\tau_1, \ldots \tau_n)c =_{AC} (\tau'_1, \ldots, \tau'_n)c))$ (when c has a map function) $=$
$\quad \mathrm{map}_{set}$
$\quad\quad (\lambda(m, S).(\texttt{map}_c\, m, S))$
$\quad\quad (\textsc{UnifyConjunction}^M_{AC}(\tau_1 =_{AC} \tau'_1 \wedge \ldots \wedge \tau_n =_{AC} \tau'_n))$

$\textsc{Unify}^M_{AC}(\tau_0 \rightarrow \tau_1 =_{AC} \tau'_0 \rightarrow \tau'_1) =$
$\quad \mathrm{map}_{set}$
$\quad\quad (\lambda(m, S).(\texttt{map}_\rightarrow\, m, S))$
$\quad\quad (\textsc{UnifyConjunction}^M_{AC}(\tau_0 =_{AC} \tau'_0 \wedge \tau'_1 =_{AC} \tau_1))$

$\textsc{Unify}^M_{AC}(\langle \tau_1 \times \ldots \times \tau_m \rangle =_{AC} \langle \tau'_1 \times \ldots \times \tau'_n \rangle) =$
let
$\quad \tau = \langle \tau_1 \times \ldots \times \tau_m \rangle \quad \tau' = \langle \tau'_1 \times \ldots \times \tau'_n \rangle$
$\quad (\tau_D, \tau_U, \tau'_U, m_D, m'_D) = \textsc{RemoveDuplicates}(\tau, \tau')$
$\quad (\tau_S, m_{sort}, \_) = \textsc{Sort}(\tau_U) \quad (\tau'_S, \_, m'_{sort}) = \textsc{Sort}(\tau'_U)$
$\quad R = \textsc{MatrixSolve}^M_{AC}(\tau_S, \tau'_S)$
in
$\quad \bigcup \mathrm{map}_{set}$
$\quad\quad (\lambda(S, m_{pre}, m_{post}, C).$
$\quad\quad\quad \mathrm{map}_{set}$
$\quad\quad\quad\quad (\lambda(m, S').$
$\quad\quad\quad\quad (m'_D \circ \texttt{map}_\times \langle \lambda x.x, m'_{sort} \circ m_{post} \circ \texttt{map}_\times\, m \circ m_{pre} \circ m_{sort} \rangle \circ m_D,$
$\quad\quad\quad\quad S'))$
$\quad\quad\quad (\textsc{UnifyWithSet}^M_{AC}((S, []), C)))$
$\quad\quad R$
end

$\textsc{Unify}^M_{AC}(\tau, \tau') =$
$\quad \{\}$ (no solutions)

Figure 6.11 Algorithms for functions $\textsc{UnifyConjunction}^M_{AC}$ and $\textsc{UnifyWithSet}^M_{AC}$.

$\textsc{UnifyConjunction}^M_{AC} : conjunction \rightarrow (substitution \times morphism\ list)set$

$$\textsc{UnifyConjunction}^M_{AC}[] =$$
$\{[], \{\}\}$ (single result of empty list of morphisms and identity substutition)

$$\textsc{UnifyConjunction}^M_{AC}[\tau =_{AC} \tau'] =$$
$$\text{map}_{set}(\lambda(m,S).([m],S))(\textsc{Unify}^M_{AC}(\tau =_{AC} \tau'))$$

$$\textsc{UnifyConjunction}^M_{AC}[\tau_1 =_{AC} \tau_1' \wedge \ldots \tau_n =_{AC} \tau_n'] =$$
let
$$V = \textsc{Unify}^M_{AC}(\tau_1 =_{AC} \tau_1') \qquad V' = \text{map}_{set}(\lambda(m,S).([m],S))V$$
in
$$\textsc{UnifyWithSet}^M_{AC}(V', \tau_2 =_{AC} \tau_2' \wedge \ldots \wedge \tau_n =_{AC} \tau_n')$$
end

$\textsc{UnifyWithSet}^M_{AC}$:
$(substitution * morphism\ list)set \times conjunction \rightarrow$
$(substitution \times morphism\ list)set$

$$\textsc{UnifyWithSet}^M_{AC}(T,C) =$$
$$\bigcup(\text{map}_{set}$$
$$(\lambda(m,S).$$
$$\text{map}_{set}\ (\lambda(m',S').(S \cup S', m'@m))\ \textsc{UnifyConjunction}^M_{AC}(SC))$$
$$T))$$

6.3.1.3 Soundness and Completeness Issues

There are two issues for proving soundness of UNIFY^{M}_{AC}: showing it produces valid substitutions, and showing that the morphisms are sound for the substitutions. It should be possible to modify Lincoln and Christian's proof to show that the substitutions are correct, and showing that morphisms are correct should not be difficult as they are all simple functions. Neither of these aspects has been proven but I believe the algorithm to be sound on the grounds that it has the same basic structure as a sound algorithm, and where it generates subproblems, it is the case that the validity of the subproblems implies validity of the problem.

Lincoln and Christian's algorithm is incomplete as it sometimes generates subproblems of the same form as the original problem, and hence does not terminate. An example of a problem which cannot be solved by it is $\langle \alpha \times \alpha \rangle =_{AC} \langle \beta \times \beta \rangle$. This problem, however may be solved by my version of the algorithm — it appears that the modified recursive calls of the new algorithm always solve simpler subproblems. This has not been proven, so it is not clear whether or not the new algorithm is complete.

There is a further issue of completeness of the set of morphisms. In fact the set of morphisms is known to be incomplete in two ways. Firstly, because common subterms of the types are removed before $\text{MATRIXSOLVE}^{M}_{AC}$ there are some rearrangements missing. This has no consequence to the completeness of the set of substitutions, but leaves some morphisms missing. For example, solving $\langle a \times \alpha \times \beta \rangle =_{AC} \langle a \times a \times a \rangle$ with the algorithm will never produce a morphism which moves the first entry of the tuple (as it is removed as being a 'duplicate' type and so cannot be changed by the rearranging morphism). Secondly there are situations where it may be better to produce a non-optimal substitution in order to produce a simpler morphism. For example for $\langle a \times \alpha \rangle =_{AC} \langle \alpha \times a \rangle$, the algorithm will produce $(\{\}, \lambda\langle x, y\rangle.\langle y, x\rangle)$. It might be preferable to have $(\{\alpha \mapsto a\}, \lambda x.x\rangle$ as well (this indicates a simpler way of debugging the program). This would change both the set of morphisms and substitutions produced. Both of these aspects are related to REMOVEDUPLICATES and it may be useful to evaluate these modifications in the future. In the current implementation this deficiency does not appear to cause problems as repeated types are rare (apart from in simple arithmetic functions) and programmers must always be cautious about getting arguments in the correct order when using these functions (e.g. when using division).

6.3.2 Partial Evaluation

Unification takes a function type and context type and produces morphisms. To generate error messages, it is then necessary to apply each morphism to the original function and partially evaluate the resulting expression. That is, we have started with an application expression $f\ a_1 \ldots a_n$, generated m_{repair} and must partially evaluate $m_{repair}\ f\ a_1 \ldots a_n$ to get the repaired expression.

One approach to partial evaluation would be to treat the morphism as a lambda expression and evaluate using β-reduction. The morphisms, however, consist of easily understood functions (see the syntax in Figure 6.7) rather than being arbitrary λ-expressions and it would be preferable that these parts appear unevaluated in any error message than arbitrary λ-expressions. Unevaluated parts of the morphism could appear if, for example, a function is to be uncurried but its arguments are not present. Hence the partial evaluation system used is based on the syntax for morphisms in Figure 6.7 rather than on the equivalent λ-terms.

Because of the nature of the morphisms, and the expression they are applied to, it will be useful to have a specific syntactic form for curried applications. For example, this makes it easy to evaluate `curry` by acting on the second and third arguments rather than on the first. The syntax for morphism expressions is in Figure 6.12 (these are the morphisms in Figure 6.7 extended with curried applications, tuples and other ML expressions).

The semantics for partial evaluation of curried applications are given in Figure 6.13 as a transition system.

6.4 Implementation[1]

The previous sections showed algorithms for generating type error messages which are intended to be more useful for debugging type errors than those which are usually produced by compilers.

The MLj compiler [BKR98][BK99][MKB00][2] has been modified to incorporate these algorithms. Emphasis in this section is on how to implement the system for this real language and compiler (Section 6.4.1) and on the performance of the implementation (Section 6.4.2).

[1]This section is based on material in [McA01].
[2]MLj is available from `http://www.dcs.ed.ac.uk/home/mlj/`.

Figure 6.12 Morphism expressions.

$$
\begin{array}{rcl}
e & ::= & \lambda p.p' \\
& | & \texttt{appUnit} \mid \texttt{mkLazy} \\
& | & \texttt{curry} \mid \texttt{uncurry} \\
& | & \texttt{map}_{\times}\langle e_1, \ldots, e_n \rangle \\
& | & \texttt{map}_{c}\langle e_1, \ldots, e_n \rangle \\
& | & \texttt{map}_{\rightarrow}\langle e, e' \rangle \\
& | & e_1 \circ \ldots \circ e_n \\
& | & \texttt{I} \\
& | & (e_0\ e_1\ \ldots\ e_n) \\
& | & \langle e_1, \ldots, e_n \rangle \\
& | & \text{“}e_{ml}\text{”}
\end{array}
$$

6.4.1 Implementing the Theory

There are several steps to producing an error message using the theory and algorithms described in this chapter.

- Type inference fails and the relevant types and the syntax which they refer to must be recorded.
- The two types are unified modulo linear isomorphism and a set of morphisms is produced.
- Each morphism is inserted into the existing syntax and the resulting expression is partially evaluated.
- Each new expression is incorporated into the error message.

To modify a compiler to follow this procedure the existing type inference routine must be changed to incorporate the first step. This involves using the algorithm in Section 4.2.1 to collect the function and context types from curried expressions. As we shall see, there are other changes to the type inference procedure too. The middle two steps involve the novel algorithms from the

Figure 6.13 Partial evaluation of morphism application.

$$\frac{}{\texttt{uncurry} f\ \langle a_1, a_2\rangle\ a_3 \ldots a_n \Rightarrow f\ a_1\ a_2\ a_3 \ldots a_n}$$

$$\frac{}{\texttt{curry} f\ a_1\ a_2\ a_3\ \ldots a_n \Rightarrow f\langle a_1, a_2\rangle\ a_3 \ldots a_n}$$

$$\frac{}{\texttt{appUnit} f\ a_1 \ldots a_n \Rightarrow f\ \langle\rangle\ a_1 \ldots a_n}$$

$$\frac{}{\texttt{mkLazy} f\langle\rangle\ a_1 \ldots a_n \Rightarrow f\ a_1 \ldots a_n}$$

$$\frac{(m' \circ f \circ m)a_1 \ldots a_n \Rightarrow e}{(\texttt{map}_{\rightarrow}\langle m, m'\rangle)\ f\ a_1 \ldots a_n \Rightarrow e}$$

$$\frac{m_1\ a_1 \Rightarrow e_1 \quad \ldots \quad m_n\ a_n \Rightarrow e_n}{(\texttt{map}_{\times}\langle m_1, \ldots, m_n\rangle)\ \langle a_1, \ldots, a_n\rangle \Rightarrow \langle e_1, \ldots, e_n\rangle}$$

These rules rewrite compositions

$$\frac{(m_2 \circ \ldots \circ m_n)f \Rightarrow e \quad m_1\ e\ a_1 \ldots a_n \Rightarrow e'}{(m_1 \circ \ldots \circ m_n)f\ a_1 \ldots\ a_n \Rightarrow e'}$$

$$\frac{m\ a_1 \ldots a_m\ b_1 \ldots b_n \Rightarrow e}{(m\ a_1 \ldots a_m)\ b_1 \ldots b_n \Rightarrow e}$$

If none of the left hands of the schema fit, then the following scheme is applied.

$$\frac{}{m\ f\ a_1 \ldots a_n \Rightarrow (m\ f)a_1 \ldots a_n}$$

previous section. The final stage is specific to a particular compiler, as each has its own way to produce error messages and pretty-print syntax.

The linear isomorphism unification routine has been implemented independently of MLj for a generic type language. This is a complex piece of code, particularly the part which solves associative-commutative unification. It took a considerable length of time to implement this, and was difficult to avoid making mistakes when writing it. Opportunities for mistakes when writing include confusing the two types being unified, and composing morphisms in the wrong order. Unfortunately a dependant type system would be required, rather than Hindley-Milner, to detect such problems while programming (dependant type checking could be used to check that the morphisms produced when the algorithm runs always have the types they are required to have).

The decision to implement unification in a generic manner meant that conversion from MLj types to the generic types was required. The advantage of this is that it should be possible to reuse the unification routines in other compilers. Based on my experience with MLj, I believe that this will be the easiest route to changing other compilers. It should also be possible to reuse the partial evaluation system.

6.4.1.1 Detecting the Error

MLj's type inference routine is based on Milner and Damas's bottom-up algorithm W [DM82] with an imperative implementation of substitutions (type variables are references which unification fills with types). This is fairly standard in compilers, though some use a top-down algorithm [LY98] and others use a hybrid of top-down and bottom-up [LY00]. Chapters 2 and 4 have more information about these algorithms.

The application case of MLj's implementation of W first finds a type for the function, τ_f, then for the argument, τ_a, and then unifies to solve the equation $\tau_f = \tau_a \rightarrow \beta$. The result type of the function is β. This approach does not take into account the structure of curried expressions in which the actual function is nested inside the left hand side of the application, and therefore does not create the context and function types used in Section 6.2.1. It is partly because W does not take the structure of curried expressions into account that error messages can be confusing.

Because of this, the type inference algorithm is modified to treat curried expressions as a single-level expression using the algorithm in Section 4.2.1 which infers a type for each argument and uses these to form a 'context' type with which the function type is unified.

A second change must be made to take account of the imperative nature of the implementation. Before unifying the types they must be recorded without their type variables instantiated (frozen). This is because when unification fails it may already have instantiated some variables. When the type is frozen, it is also converted to the particular form required for isomorphic unification (described in Section 6.4.1.2).

Lastly, it was necessary to detect when unification failed, and to act on this by calling a new routine. For MLj, this meant modifying unification so that it reports whether it succeeded or failed. This was because the unification routine is responsible for generating error messages and applies a “best guess” substitution regardless of whether it failed. Imperatively generating an error message is the only indication that an error has occurred, and it was not previously possible for the type inference routine to act on this. Hence a new version of the unification routine was added which does not report errors and does return a boolean indicating whether errors were found. This was implemented by setting a new boolean reference which is inspected before `unify` attempts to report an error. If the reference is set then the occurrence of an error is recorded in a second reference instead of being reported as a type error.

When a type error is detected the context and function types are passed to the new type debugging module, along with the syntax for the function and curried arguments. Pseudocode for this is shown in Figure 6.14.

The pseudocode makes use of some functions

- `splitFunc` splits the expression into the function and a list of curried arguments.
- `infExp C e` returns an elaborated version of expression `e` and its type under context `C`.
- `buildContextType` takes a new type variable and results of elaborating the arguments, it returns a list of elaborated expressions and the context type.
- `ConvertTypes.convertType` freezes the types.
- `unifyNoReport` unifies two types imperatively, returning the unified type and a boolean indicating whether or not unification succeeded. Its extra arguments are used to produce error messages (they are not actually used).
- `buildElabExp` builds an elaborated curried application from elaborated function and arguments.

Two additional complications in the implementation are recording the locations which annotate the abstract syntax and must be reported in error messages, and type checking applications of Java static methods (this is particular to MLj).

Figure 6.14 Pseudocode for modified MLj type inference (application expression case).

```
App(func, arg) =>
  let
     val (e0, es) = splitFunc(func, [arg])
     val (e0', tFunction) = infExp C e0
     val esAndTs = map (infExp C) es
     val beta = SMLTy.freshType()
     val (es', tContext) =buildContextType (beta,  esAndTs)
     val tFunctionLI = ConvertTypes.convertType tFunction
     val tFunctionLI = ConvertTypes.convertType tContext

     val (_, unifyErr) =
         unifyNoReport ((SOME loc, "actual function type", tFunction),
                        (SOME loc, "context requires function type", tContext))

     val _ = if unifyErr then
               (case TyDebug.tydebug(tFunctionLI, tContextLI, e0::es) of
                  SOME s =>
                    (* Generate type debugging error message. *)
                | NONE =>
                    (* Generate traditional error messages. *) )
             else
                ()

     val e' = buildElabExp (e0', es')

  in
    (e', beta)
  end
```

6.4.1.2 Representation of Types

The implementation of unification is designed to be used in any compiler written in SML, and has its own representation of types. This representation is based on an SML datatype to allow pattern matching (unlike the representation in MLj, which is an abstract datatype with explicit inspection functions).

For unification modulo linear isomorphism, it is necessary to explicitly represent the tuples in the type. This differs from the presentation of many type systems, in which the product operator is a binary operator, and differs from the semantics of SML, in which a tuple is a derived form based on records. Other types are type variables, constructors of any arity and functions. The representation is shown in Figure 6.15.

Figure 6.15 Representation of Types.

```
datatype types =
    FUN of types * types
  | TYVAR of string
  | CON of string * types list (* e.g CON("list", [CON("int", [])]) *)
  | TUPLE of types list
```

The freezing of MLj types into this representation is extremely simple. MLj's type to string conversion functions are used to generate the strings for constructors and type variables. The implementation currently does not handle record types, these are simply converted to strings and recorded as nullary constructors. Ignoring records is unlikely to have much effect on usefulness as the presence of tuples in the language means that records are used less in Standard ML than they are in languages without tuples (they do not, for example, appear anywhere in the implementation of this chapter or in MLj's type inference).

The freezing routine for types in MLj is around 40 lines of code, whereas to rewrite the unification routine to use MLj types would have involved changing at least 5 files and over 500 lines of code. Using this technique in other compilers should make implementation similarly easy.

6.4.1.3 Representation of Morphisms

The representation of morphisms matches the definition of morphism expressions in Figure 6.12. This is shown in Figure 6.16.

Morphisms are treated as an abstract type during unification and are created using functions in the signature in Figure 6.17. The constructor functions perform simple optimisations on the

Figure 6.16 Morphism expression representation.

```
datatype morphism =
   CURRY | UNCURRY
 | APP_UNIT | MK_LAZY
 | MAP_FUN of
     morphism * morphism
 | MAP_CON of string * morphism list
 | MAP_TUPLE of morphism list
 | LAMBDA_PAT of pattern * pattern
 | COMPOSE of morphism list
     (* "a o b" is [a, b], [] represents identity *)
 | CURRY_APP of morphism * morphism list
     (* [] represents identity *)
 | I
 | ML_EXP of Syntax.Exp
 | TUPLE_EXP of morphism list
```

morphisms produced. These optimisations were implemented in order to make the morphisms produced shorter and more readable during program development. The optimisations are

- Mapping identity is the same as identity.
- Identity can be removed from compositions.
- $\lambda p.p$ is identity.
- $\mathtt{map}_{\times}\langle\lambda p_1.p'_1,\ldots,\lambda p_n.p'_n\rangle = \lambda\langle p_1,\ldots,p_n\rangle.\langle p'_1,\ldots p'_n\rangle$
- $\lambda p_1.p'_1 \circ \lambda p_2.p'_2 = \lambda S p_1.S p'_2$ where S unifies p'_1 and p_2.

The code for the representation of morphisms also includes a record of which type constructors have map functions, and what these functions are called. It currently offers these for the SML library lists, options, arrays and vectors.

Figure 6.17 Signature abstracting the morphism representation.

```
sig

     type morphism (* morphism expressions *)

     val I : morphism
     val curry : morphism
     val uncurry : morphism
     val appUnit : morphism
     val mkLazy : morphism
     val mapFun : morphism * morphism -> morphism
     val mapCon : string * morphism list -> morphism
     val mapTuple : morphism list -> morphism
     val lambdaPat : Patterns.pattern * Patterns.pattern -> morphism
     val compose : morphism * morphism -> morphism

     val hasMap : string * int -> bool

     (* This is the function for applying a morphism to a
        list of ML expressions (as curried arguments). *)
     val applyToML : morphism * Syntax.Exp list -> morphism

     val toString : morphism -> string

  end
```

6.4.1.4 Unifying and Generating Morphisms

The implementation of rewriting to normal forms and generating the relevant morphisms is relatively straight-forward, directly following the algorithms in Figures 6.8 and 6.9.

The implementation of AC-unification is much more complicated. In the implementation, the data is represented as follows

- A type equation, $\tau =_{AC} \tau'$ is represented as a pair of types.
- A conjunction of equations $\tau_1 =_{AC} \tau'_1 \wedge \ldots \wedge \tau_n =_{AC} \tau'_n$ is represented as a list of equations (in the order shown).
- Substitutions are lists of type variable and type pairs. Substitutions can be combined with list append (for $\cup$) — which will only work if their domains are disjoint and no type variable in the range of one appears in the domain of the other.
- Sets of results are represented as streams (this is on the advice of [LC89] and leads to an aesthetically pleasing implementation).

The main complication in the implementation is generating a lazy stream of matrices, the code follows Lincoln and Christian's design for this.

6.4.1.5 Partial Evaluation

After generating the morphism, we must evaluate its application to the original function expression. First, the MLj syntax is converted to the morphism expression type (in Figure 6.16) by looking through it for tuples and curried applications. A direct implementation of the transition system in Figure 6.13 is then used to evaluate the expression.

Here are some examples of the results which the implementation of partial evaluation produces

```
(curry o mapFun((fn (A7, A6) => (A6, A7)), I) o uncurry)
map oneTwoThreeList intToString
```

Rewrites to

```
map intToString oneTwoThreeList
```

And

```
(mapFun(I, curry) o curry o mapFun((fn (A11, (A12, A13)) =>
(A11, A12, A13)), I) o mapFun(((fn ((), (A17, A18, A19)) =>
(A17, A18, A19)) o mapTuple2 (I, ((fn (A24, A25, A23) =>
(A23, A24, A25)) o mapTuple3 (I, I, mapFun((fn (A38, A37) =>
(A37, A38) ), I)) o (fn (A20, A21, A22) => (A21, A22, A20) ))) o
```

```
 (fn (A14, A15, A16) => ((), (A14, A15, A16)))), I) o
mapFun((fn (A4, A5, A6) => (A4, (A5, A6))), I) o uncurry o
mapFun(I, uncurry)) foldleft addReciprocals zero intList
```

Rewrites to

```
foldleft (addReciprocals o (fn (A38, A37) => (A37, A38))) zero intList
```

All the morphisms shown above were actually generated during unification — so it is clear that rewriting is essential to reduce the size of output and make it human readable. The programmer's mistake in the second example was to pass `foldleft` a function which takes its arguments in the wrong order (a mistake I frequently make and which leads to extremely obscure error messages).

The most notable artefact from the isomorphism is in the last example above where the anonymous function is left unevaluated. Future work is to partially evaluate the composition of a lambda term morphism with an MLj lambda term, e.g. to partially evaluate

```
( fn (total, i) => total + (1.0 / (Real.fromInt i))) o
  (fn  (A38, A37) => (A37, A38) )
```

to get `fn (i, total) => total + (1.0 / (Real.fromInt i))`. This evaluation of $(\lambda p_1.e) \circ (\lambda p_2.p_3)$ (where $\lambda p_2.p_3$ is a linear morphism) is obtained by unifying p_3 with p_1 to get substitution (on identifiers) S and rewriting as $\lambda Sp_1.Se$. This optimisation is currently implemented for composition of two linear morphisms when generating morphisms.

6.4.1.6 Modifying the User Interface

Finally, having generated a morphism and rewritten the expression, the compiler is ready to generate an error message. This section is mostly specific to MLj, but it should shed light on design features which make it easier to improve error messages.

The intention of this chapter was to create error messages which describe how to repair errors by printing syntax. The first problem faced with MLj was that it does not have a pretty printer for abstract syntax (see the sample error messages in Chapter 2). The first step in development, therefore, was to write a pretty printer for the expressions produced from partial evaluation (which are represented by the type in Figure 6.16 and contain MLj expressions and morphisms). The current implementation is currently overly simple : the only sort of MLj expression it can print is a simple identifier, and it does not have any sort of intelligent indentation. While the implementation is not yet of a quality for general release, it is sufficient to demonstrate that the concepts in this chapter can be applied.

The abstract syntax in MLj does contain “locations” which are indices into the source code file showing where an expression begins and ends. Unfortunately the source file is closed before type checking, and the information about where the file is located in the file system is out of scope during type checking. Syntax trees containing the string representation of the syntax would be ideal for the error messages.

In Standard ML of New Jersey, or Moscow ML there are syntax pretty-printers and this problem would not apply.

The string formed simply reads “Try changing . . . to . . . (or . . .)”. This is returned from the type debugging module to the type checker. If the context and function types could not be unified modulo isomorphism then no string is returned.

The type checker must then produce an error message from the string, or if no string is produced it must generate a traditional error message about types failing to unify. Following Nielson’s error message guidelines, the messages is precise about where the types came from: one is the function type and the other is the context type. This should make the message more useful than MLj’s old error messages in which one type is the “argument type” and the other is “expected”. the ML expression is also printed in the traditional error message.

6.4.2 Performance and Testing

There are two important aspects of performance: is the output useful and does the program run in a reasonable time? Also of interest, and discussed at the end of this section, is whether or not the program will be easy to extend to create further improved error messages and whether it will be easy to re-implement for other compilers (this issue has been touched upon already).

Memory usage is omitted, partly because of the difficulty in measuring this, but also because it is not particularly relevant as most platforms allow virtual memory. Using virtual memory means that the size of data stored has little relevance but that there is an impact on the speed of the program as pages are fetched which may be noticed by the user and is measured below.

6.4.2.1 Sample Output

Here are some actual examples of output from the modified version of MLj. The first example (seen in the beginning of the chapter in Figure 6.1) has curried function used as uncurried, and its arguments being passed in the wrong order. Note that all MLj input files must contain a structure.

```
structure Test1 = struct

  val intList = [1, 2, 3]
  val intToString = Int.toString

  val _ = map (intList, intToString)

end
```

The error message is

```
Error at 6.11-14: Try changing
  map (intList, intToString)
to
  (map intToString intList)
```

The test program starts by creating a named list (`intList`) and renaming a function (`intToString`). This is because the error message pretty-printer cannot currently print the syntax for lists or long identifiers. Substituting the definitions for the identifiers in the program results in the same morphism and form of expression being generated, as shown in the error message below

```
Error at 3.13-16: Try changing
  map (?, ?)
to
  (map ? ?)
```

A more complex example has an argument function with the wrong type

```
structure Foldleft =
    struct
```

```
        val foldleft = List.foldl
        val intList = [1 2, 3]
        val zero = 0.0
        fun addReciprocals (total, i) = total + (1.0 / (Real.fromInt i))

        val totalOfReciprocals = foldleft addReciprocals zero intList

    end
```

This error message is produced

```
  Error at 9.34-42: Try changing
    foldleft addReciprocals zero intList
  to
    (foldleft (addReciprocals o (fn (A38, A37) => (A37, A38) ))
zero intList)
```

The programmer could read this literally and use it as a way to repair the mistake, or he could reformulate the `addReciprocals` function to take its arguments in the other order.

6.4.2.2 Run Time

A short run time is not actually particularly important to this program. Programmers expect type inference to take some time (several seconds minimum) and when they have made a mistake they will have to spend time correcting it (usually several minutes).

The real time taken to unify, evaluate and generate error message text was measured on the desktop computer I work on (230MHz Intel Pentium II with 64Mb RAM running Redhat Linux version 6.2). The compiler was initialised and used to compile the Foldleft example 5 times. This was then repeated a total of 4 times (20 individual timings altogether). The first compilation after initialising the compiler consistently took longer than the others: $3410 \pm 60\mu s$. The other runs took $2760 \pm 30\mu s$. The shorter times are probably because the routine is stored in the processor cache after the first run. These times of a few thousandths of a second are short enough not to have any perceived impact on the run-time of the compiler.

6.4.2.3 Program Extensibility

As has been stated earlier, I believe the approach used to implement this system could be applied to other ML compilers. Changes to be made are

- Change the application case of type inference to deal with curried expressions.
- Convert the type to the special representation for isomorphic unification before conventional unification.
- Convert abstract syntax into morphism expressions and partially evaluate.

There are a number of known deficiencies in the current implementation:

- Some types may not be unified correctly due to the simplistic conversion (especially with MLj's constraint solving [MKB00]).
- Further partial evaluation is possible.
- The pretty-printer does not deal with all MLj expressions (only identifiers at present).
- A naïve location is proposed for the location of a mistake.
- Only linear isomorphisms are used to unify (and no pseudo-isomorphisms such as coercing a value to a list).

6.5 Conclusions

We have seen that a theoretical construction — type isomorphisms — implies that it is possible for compilers to suggest ways to repair programs with type errors. It has also been demonstrated that it is possible to implement this theory for a real language and compiler, but that there were obstacles to overcome. In particular the representation of types and syntax can make implementation tricky, and the extra features of programming language type systems do not fit in with the theory.

My source code is available online at `http://www.dcs.ed.ac.uk/home/bjm/`.

Chapter 7

Beyond Hindley-Milner: the MLj Type System[1]

The previous chapters have examined Hindley-Milner type systems. It has also been noted that most programming languages have other features which can cause difficulties in debugging type errors. For example the 1990 definition of Standard ML had 'sharing constraints' in its module system, and the 1997 definition has value polymorphism.

A popular feature of new type systems is subtyping. This can capture a range of different properties of types, in particular it is useful for object-oriented languages in which it can reflect the hierarchy of inheritance of classes. A number of languages combine Hindley-Milner style type inference with subtyping.

This chapter examines one such example of subtyping used to capture an inheritance relation. MLj[2] [BKR98, BK99, MKB00] is an extension of Standard ML which integrates with Java[3] [GJS96] and in which Java classes become ML types. A subtyping relation captures Java's inheritance and interface mechanisms. The extended type system is particularly relevant to this thesis as it has many pragmatically motivated features which do not directly correspond to any particular textbook theories and which can create unusual typing problems for the programmer. A prime example of this is the use of option types to represent possible null references, and implicit insertions of coercions to hide this from the programmer.

[1]This chapter contains material from [MKB00] which was joint work with Andrew Kennedy and Nick Benton completed while I was an employee of Microsoft Research UK Ltd.

[2]The latest version of MLj is available from `http://www.dcs.ed.ac.uk/home/mlj/`.

[3]Java is a trademark of Sun Microsystems, Inc. Compilers, documentation and run-time systems are available from `http://www.java.sun.com/`.

In this chapter, Sections 7.1–7.3 introduce MLj. Sections 7.4 and 7.5 describe the constraint solving part of type inference and Section 7.6 discusses possible error messages for MLj.

7.1 Introduction to MLj

MLj is an extension of the Standard ML language which allows inter-operation with Java. The compiler creates Java byte code from SML [BKR98], and the extensions to the language allow programmers to access Java classes and create new Java classes [BK99]. It has a type system which combines ML's parametric polymorphism and type inference with Java's subtyping (class hierarchy) and arbitrary overloading of methods. Type inference involves solving constraints [MKB00]. In this chapter, we will see some of the difficulties in understanding the results of type inference for MLj such as the fact that MLj programs do not necessarily have principal type schemes.

The Java type system differs considerably from SML's. In particular, objects of one Java class can be treated as objects of another Java class through *widening* (conversion to a superclass) or *narrowing* (run-time checked conversion to a subclass). Java methods can also be overloaded with arbitrary argument types.

MLj is unusual, in that it is not a clean new language design, nor even a clean new extension to an existing design, but rather attempts to merge features of one language (type inference in ML) with those of another (subtyping and overloading in Java). MLj was designed with the following aims.

- MLj should be a *conservative* extension of Standard ML. An SML program making no use of the MLj extensions type checks under MLj if and only if it would type check under SML with the same type. Furthermore, it has the same dynamic behaviour (although it diverges from the standard in minor ways, for example when arithmetic overflow occurs).

- Type inference should not make any "closed world" assumptions about external Java class libraries. For example, inference should not make use of the knowledge that some class has only a single subclass. If a program type checks against a Java class library then it should type check against extensions to the library, with the same type.

- As far as is sensible the Java extensions are "in the spirit of" ML. For example, it does not follow Java and support implicit widening of numeric types, as these are also SML types which have explicit widening functions in the standard library.

- The type system should be intuitive — type inference should not surprise the (SML) programmer.

It is also important to note two non-aims.

- The MLj language is not attempting to provide a full object-oriented extension to ML (like Objective Caml [LRVD99]). Java's subtyping is not extended to all ML types, and there is no general subsumption rule. Subtyping is applied only to inter-operability features, and only on the subset of ML types that match with Java.

- Type inference for MLj is not type inference for Java, which could have a different flavour entirely. Attempting to define type inference for Java is probably a bad idea anyway, as Java programs make so much use of overloading and implicit coercion that type inference would have very little information to guide it.

Two questions are examined here with reference to the production of helpful information for programmers using MLj.

- What type should be assigned to a program?

- How do we assign this type?

The first question seems odd, given that the semantics of the type system have already been defined and these might be expected to define which type should be assigned to a program. The semantics may actually allow several type derivations for a program. This is also the case in Standard ML and other Hindley-Milner languages but these always allow a *principal type scheme* [DM82] which captures the set of all types which may be assigned. MLj programs, however do not necessarily have a principal type scheme because of Java overloading. The particular derivation which is selected should be the one that the programmer intuitively expected. In addition, in the context of this thesis, we must ask

- What should we do if a program cannot be assigned a type?

7.2 Overview of MLj types

Here is a sketch of the type system of MLj as described in detail in [BK99].

SML and Java primitive types correspond (so MLj and Java `int` are the same). SML arrays and non-null Java arrays also correspond. The types of SML are extended to incorporate Java class and interface types. Java arrays and objects that originate in Java code can take the value `null`; we model this using ML's `NONE` value, giving references types of the form $\tau\,\texttt{option}$ for class or array type τ. Hence, a subset of MLj types are defined to be the set $\mathsf{JavaType}$ as described in Figure 7.1.

Figure 7.1 Java types.

$$\mathsf{PrimType} = \{\texttt{bool}, \texttt{int}, \texttt{char}, \texttt{real}, \texttt{Real32.real}, \texttt{Int8.int}, \texttt{Int16.int}, \texttt{Int64.int}\}$$

$$\frac{\tau \in \mathsf{PrimType} \cup \mathsf{ClassType}}{\tau \in \mathsf{JavaType}} \qquad \frac{c \in \mathsf{ClassType}}{c\ \texttt{option} \in \mathsf{JavaType}}$$

$$\frac{\tau \in \mathsf{JavaType}}{\tau\ \texttt{array} \in \mathsf{JavaType}} \qquad \frac{\tau \in \mathsf{JavaType}}{\tau\ \texttt{array option} \in \mathsf{JavaType}}$$

An important part of Java's type system is the relation which defines which types may be implicitly raised to other types when supplied as method arguments. For example `Object` is wider than `Integer`, so a method requiring an `Object` may be supplied with an `Integer`. We capture this in MLj with four relations defined in Figure 7.2.

- $\tau \leq_w \tau'$, if both types are in $\mathsf{JavaType}$ and τ can be widened to τ'.
- $\tau \leq_a \tau'$, if τ can be used as a method argument when the method takes something of type τ'. In this case either both types are in $\mathsf{JavaType}$ and related with $\leq_w$, or else they are exactly equal (hence the "or $\tau = \tau'$" in the definition).
- $\tau \leq_m \tau'$, if a method of type τ can be used as if it has type τ'.
- $\tau \geq_n \tau'$, if τ can be narrowed to τ' using a cast. See [BK99] for the definition of narrowing.

Figure 7.2 Subtyping relations.

$$\frac{}{\tau \leq_w \tau} \qquad \frac{\tau \leq_w \tau'' \quad \tau'' \leq_w \tau'}{\tau \leq_w \tau'} \qquad \frac{}{c \leq_w \mathsf{super}(c)} \qquad \frac{}{\tau\ \mathtt{array} \leq_w \mathtt{Object}}$$

$$\frac{}{\tau \leq_w \tau\ \mathtt{option}} \qquad \frac{\tau \leq_w \tau'}{\tau\ \mathtt{array} \leq_w \tau'\ \mathtt{array}} \qquad \frac{\tau \leq_w \tau'}{\tau\ \mathtt{option} \leq_w \tau'\ \mathtt{option}}$$

$$\frac{\tau \leq_w \tau' \text{ or } \tau = \tau'}{\tau \leq_a \tau'} \qquad \frac{\tau'_1 \leq_a \tau_1 \ldots \tau'_n \leq_a \tau_n}{\tau_1 \times \ldots \times \tau_n \to \tau' \leq_m \tau'_1 \times \ldots \tau'_n \to \tau'}$$

Note that $\leq_w$ is defined over only $\mathsf{JavaType}$. So it is not the case that $\tau \leq_w \tau$ `option` for all types τ (e.g. ML datatypes or functions), only when τ `option` $\in \mathsf{JavaType}$. This disallows cases like `Object option` $\leq_w$ `Object option option` as well. Widening is used to define method type conversion, $\leq_m$. This tells us, for example, that a method of type `Object -> bool` may be used as a method of type `Integer -> bool`. Non-$\mathsf{JavaType}$ method arguments must have exactly the correct type.

The restriction of $\leq_w$ to types in $\mathsf{JavaType}$ is expected to be one aspect of the type system with the potential to confuse programmers (based on observation of and discussion with some of the current users of MLj).

We are now in a position to understand the typing rules for the new MLj expressions, patterns, declarations and specifications, as shown in Figure 7.3. These rules were originally given in [BK99] and are in the style of [MTHM97].

Note the different treatments of fields and methods: though there is no syntactic difference fields have no subtyping relation, but methods do. While the (mth) rule allows subsumption, there is no *general* subsumption rule for application.

We previously noted that the principal type scheme property does not hold for MLj, because of overloading. An example of a program with two distinct type schemes is

```
(* Class C has overloaded method
     m : int -> bool     m : real -> string *)
fun f (c : C) x = C.#m x
```

Function `f` can take either the type `int -> bool` or `real -> string`. There is no type scheme

Figure 7.3 The typing rules for MLj syntax (apart from `_classtype` for creating new classes).

Expressions $C \vdash exp \Rightarrow \tau$

$$(statfld)\,\frac{\begin{array}{c}C(longvid) = \mathsf{member}(c, id)\\ (id : \tau) \in \mathsf{staticfields}(c)\end{array}}{C \vdash longvid \Rightarrow \tau} \qquad (fld)\,\frac{\begin{array}{c}C \vdash exp \Rightarrow c\\ (id : \tau) \in \mathsf{fields}(c)\end{array}}{C \vdash exp\,\texttt{.\#}id \Rightarrow \tau}$$

$$(statmth)\,\frac{\begin{array}{c}C(longvid) = \mathsf{member}(c, id)\\ (id : \tau) \in \mathsf{staticmethods}(c)\end{array} \quad \tau \leq_m \tau'}{C \vdash longvid \Rightarrow \tau'} \qquad (mth)\,\frac{\begin{array}{c}C \vdash exp \Rightarrow c\\ (id : \tau) \in \mathsf{methods}(c)\end{array} \quad \tau \leq_m \tau'}{C \vdash exp\,\texttt{.\#}id \Rightarrow \tau'}$$

$$(supmth)\,\frac{\begin{array}{c}C \vdash exp \Rightarrow c\\ (id : \tau) \in \mathsf{methods}(\mathsf{super}(c))\end{array} \quad \tau \leq_m \tau'}{C \vdash exp\,\texttt{.\#\#}id \Rightarrow \tau'} \qquad (cast)\,\frac{\begin{array}{c}C \vdash exp \Rightarrow \tau\\ C \vdash ty \Rightarrow \tau'\end{array} \quad \tau \leq_w \tau' \text{ or } \tau \geq_n \tau'}{C \vdash exp{:>}ty \Rightarrow \tau'}$$

Patterns $C \vdash pat \Rightarrow (VE, \tau)$

$$(patcast)\,\frac{vid \notin \mathrm{Dom}(C) \text{ or is of } C(vid) = \mathtt{v} \quad C \vdash ty \Rightarrow \tau \quad \tau \geq_n \tau'}{C \vdash vid{:>}ty \Rightarrow (\{vid \mapsto (\tau, \mathtt{v})\}, \tau')}$$

which describes this set of types. The program must be rejected but may be accepted as part of a larger program if its context allows its type to be decided. This restriction is analogous to the treatment of SML overloading and value polymorphism and is therfore compatible with the expectations of programmers, though still something to beware of when announcing errors.

7.3 Examples

We will now look at some examples of MLj programs to see what different types can be derived for them and say which (if any) is the one to assign. For those for which a type cannot or should not be assigned, possible error messages are discussed.

7.3.1 Simple Examples — Only One Type Derivable

First let us look at some simple examples which only have one derivable type. The first example creates a new Java class `Counter` with three methods, `reset`, `inc` and `read`. All the methods created have only one possible type as shown in the comments.

```
_classtype Counter()      (* Create a new class called Counter. *)
  with local
    val c = ref 0         (* Local to instance of class. *)
  in
    reset () = c := 0     (* Method reset : unit -> unit *)
    inc () = c := !c + 1  (* Method inc : unit -> unit *)
    read () = !c          (* Method read : unit -> int *)
  end
```

We now can write code to create an instance of this class and call its methods. Since the methods are not overloaded there is only one type derivation for this program.

```
val counter = Counter () (* Create an instance. *)
val _ = (counter.#inc () ; counter.#inc () ) (* Inc twice *)
val count = counter.#read()
val _ = counter.#reset ()
```

7.3.2 Casts

Two sorts of cast are allowed in MLj. We can cast an object to another class, or cast the argument of a function to a particular class. Both uses of casting involve runtime checks, in contrast to SML type annotations.

In the next program, the object is cast to have type `Object` then recast to `Integer`. This is possible because rule $(cast)$ from Figure 7.3 allows widening and narrowing.

```
val i = java.lang.Integer 3 (* val i : Integer *)
val i' = i :> Object        (* val i' : Object. *)
val i'' = i' :> Integer      (* val i'' : Integer. *)
```

Next we see the use of pattern casts (the rule *patcast* in Figure 7.3). Pattern casts are analogous to pattern matching, if the argument can be cast to the given type then the pattern is matched, otherwise the remaining patterns are attempted.

```
fun f (x :> java.lang.Integer) = "Integer"
  | f (x :> java.lang.String) = "String"
  | f _ = "Object"
(* val f : Object option -> string *)
val i = java.lang.Integer 3  (* val i : Integer *)
val r = f i                  (* returns "Integer". *)
val i′ = i :> Object         (* val i′ : Object *)
val r′ = f i′                (* Also returns "Integer". *)
```

7.3.3 Overloading — Use the Most Specific Method

Because of method overloading, several types may be derivable for a program. That is, the original program is ambiguous. Default resolution of ambiguity of method overloading is in the spirit of Java's "most specific method" choice [GJS96]. Consider the following program in which a method is overloaded at a superclass and subclass. Clearly when applied to the subclass the most specific version should be used.

```
(* Class A has overloaded method
     m : Object -> int     m : B -> bool *)
val a = A() val b = B()
val r = a.#m(b) end      (* val r : bool *)
```

It must be possible, however, for the programmer to override this behaviour. In this case, the expression `a.#m(b :> Object)` would force the type checker to choose the other method by casting `b` up to `Object` before passing it to `m`.

Next we show that an argument may be cast up without an explicit cast if this is necessary to create the correct result type. The less specific method is used since using the most specific will not allow the program to type-check.

```
(* Class A has overloaded method
```

```
      m : Object -> string     m : Integer -> int *)
  let val a = A()  val i = Integer 3
  in print (a.#m i) end
```

There is an issue of *explanation* for such resolution of overloading. If the program is given an unexpected type then the programmer may wish to know why this is.

7.3.4 Defining Methods — Give Most General Type

A defined method must be given the most general type possible.

```
  _classtype Foo ()
    with
      m x = 3
    end
  val obj = Foo ()             (* Create instance of Foo *)
  val r = obj.#m (Integer 5)   (* Apply method *)
```

The method `m` is given type `Object option -> int` regardless of it's use with `Integer 5`. This is in order that extending the program does not require the method to be given a more general type, for example if the lines `val r' = obj.#m (SOME (Object ()))` were added.

7.3.5 Rejected Programs

Programs may be rejected for a number of reasons. Sometimes there is no unique most specific solution as we saw in the previous section. For example the overloading in the next example cannot be resolved. Neither way of resolving is more general than the other.

```
 (* Class A has overloaded method
      m : C -> int
      m : Object -> bool
    Class B has overloaded method
      m : C -> bool
      m : Object -> int *)
  let val a = A() val b = B() val c = C()
```

```
in [a.#m(c), b.#m(c)] end
(* Program rejected. *)
```

Programs may also be rejected because of the *open world assumption*. This states that if another type may be derived for a program in some extension of the class hierarchy, then the program should be rejected. We reject the definition of `f`, below, because even though it is intended to be of type `C -> unit` in some extensions of the class hierarchy (those in which other classes have a method called `qwexiphlibit`) we would not be able to establish a unique type for `f`.

```
_classtype C ()
  with
    qwexiphlibit () = ()
  end
fun f obj = obj.#qwexiphlibit ()
(* Program rejected. *)
```

Type error messages for this sort of problem should mention that a class could not be found for the identifier in question.

Also, consider the identity method on objects. We reject the following version because the choice of seemingly reasonable types for method `I` are `Object option -> Object option` and `Integer -> Integer`, neither of which is more specific.

```
_classtype I ()
  with
    I x = x
  end
val i = I ()
val r = i.#I (Integer 3)
(* Program rejected. *)
```

This unusual case would require a message explaining why a type could not be assigned. This could possibly mention the two seemingly reasonable types.

We *can* type an identity method by adding annotations.

```
_classtype I_General ()
  with
    I (x : Object option) = x
  end
val r1 = (I_General ()).#I (Integer 3)
(* val r1 : Object option *)

_classtype I_Specific ()
  with
    I (x : Integer) = x
  end
val r2 = (I_Specific ()).#I (Integer 3)
(* val r2 : Integer *)
```

These fit our expectations of Java methods rather than SML functions. They operate at fixed types (with subsumption in the arguments at calls) rather than being polymorphic. Overcoming this would require extensive changes to the SML type system, and the introduction of bounded type variables for classes (to give a type like $\forall\alpha \leq_w \texttt{Object option}.\alpha \rightarrow \alpha$). Such types would make no sense in the Java world.

7.4 Constraints

In contrast to the unification-based type inference for ML, types for MLj cannot easily be inferred whilst traversing the parse tree. This is due to the presence of unrestricted overloading of method types and the subtyping relation. Instead, a constraint-based approach is used with several constraint forms.

- τ has $m : \tau'$, τ is a class with member $m : \tau'$.
- τ hasStatic $m : \tau'$, similarly for static members.
- $\tau \leq_{\mathsf{memb}} \tau'$, either equality for fields or $\leq_m$ for methods.
- $\tau \leq_{\mathsf{sup}} \tau'$, τ' is the super-type of τ.

- $\tau \leq_{\mathsf{meth}} \tau'$, implements $\leq_m$.
- $\tau \leq_{\mathsf{Cast}} \tau'$, τ can be cast to τ'.

Figure 7.4 shows how constraints are generated for each relevant syntactic structure during the conventional Hindley-Milner type inference algorithm, given in [DM82].

Constraint solvers are given constraints featuring variables. They return an assignment of the variables such that the constraints are satisfied. In type inference terminology, such an assignment is a *substitution* from type variables to types and this is the usual result of type inference.

Previous work on constraints has included John Mitchell's inference algorithms for idealised subtyped languages [Mit96, Mit91]. His language does not include all the complexities of MLj. This work is extended by [JM93] and [AW93]. Rather than looking at general subtyped languages, Eiefig, Smith and Trifonov look at type inference for objects in [EST95]. Constraints are also used in program analysis. Nielson, Nielson and Hankin describe this in [NNH98]. François Pottier's work [Pot96] is closest to MLj's, in particular in the manner he treats upper and lower bounds. Duggan's work on type explanations is also based on constraint solving (though the constraints are typically equality constraints) and may be of use in explaining how types were inferred. There is no other widely known work on explaining the results of constraint solving, or of explaining why a set of constraints could not be solved.

In particular the MLj algorithm differs from other work as it requires nondeterminism and backtracking. This is partly because the subtype relation is rather irregular (because of `option` and the set JavaType) and partly because of overloading.

Since type inference can fail while solving constraints it is necessary to create error messages during this process. It is also desirable to produce other information as the results of solving constraints may not be expected by the programmer (as seen in some of the examples previously).

7.5 The MLj Constraint Solver

The engine of the MLj constraint solver starts with a set of constraints and returns a set of *extended substitutions*. These are mappings from type variables to either an exact type or upper and lower bounds as described below[1].

[1] This section refers to the constraint solver described in [MKB00], which is in the current development version awaiting release due to copyright restrictions (Andrew Kennedy, personal communication).

Figure 7.4 Generating types and constraint sets for new expressions.

Constructors, *classname* (Where *classname* names the class c.)

$$(\tau_{\text{use}}, \{c \ \mathsf{hasStaticConstructor}\ \tau_{\text{def}},\ \tau_{\text{def}} \leq_{\mathsf{memb}} \tau_{\text{use}}\})$$
$$\text{where}\ \begin{array}{rcl} \tau_{\text{use}} & = & \beta_0 \to c \\ \tau_{\text{def}} & = & \beta_1 \to c \end{array}\ \text{for new}\ \beta_0, \beta_1$$

Static methods, *classname.vid* (Where *classname* names class c.)

$$(\beta_{\text{use}}, \{c\ \mathsf{hasStatic}\ vid : \beta_{\text{def}},\ \beta_{\text{def}} \leq_{\mathsf{memb}} \beta_{\text{use}}\})\ \text{for new}\ \beta_{\text{def}}, \beta_{\text{use}}$$

Member access, exp **.** **#**id

$$(\beta_{\text{use}}, \{\tau\ \mathsf{has}\ id : \beta_{\text{def}},\ \beta_{\text{def}} \leq_{\mathsf{memb}} \beta_{\text{use}}\})$$
$$\text{where}\ exp : \tau,\ \text{for new}\ \beta_{\text{use}}, \beta_{\text{def}}$$

Superclass member access, exp **.** **##**id

$$(\beta_{\text{use}}, \{\beta_{\text{super}}\ \mathsf{has}\ id : \beta_{\text{def}},\ \beta_{\text{def}} \leq_{\mathsf{memb}} \beta_{\text{use}},\ \tau \leq_{\mathsf{sup}} \beta_{\text{super}}\})$$
$$\text{where}\ exp : \tau,\ \text{for new}\ \beta_{\text{use}}, \beta_{\text{def}}, \beta_{\text{super}}$$

Expression casts, $exp{:>}ty$

$$(\tau', \{\tau \leq_{\mathsf{Cast}} \tau'\})\ \text{where}\ exp : \tau, ty : \tau'$$

Pattern casts, $id{:>}ty$ We must return a type for the pattern, an assignment of newly bound identifiers to types and a set of constraints. (The set assigning types to identifiers is not required for the previous expression cases.)

$$(\tau, \{id : \beta\}, \{\beta \geq_n \tau\})\ \text{where}\ ty : \tau,\ \text{for new}\ \beta$$

`_classtype` definitions are omitted for space constraints.

Constraint sets are solved by repeatedly selecting constraints, applying the current substitution to them and simplifying them to create new constraints and changes to the extended substitution. Some constraints have multiple solutions (due to arbitrary overloading) so the solver is backtracking and may produce multiple solutions (corresponding to the many possible derivations for a program).

After solving, the best extended substitution (if it exists) is selected.

Two difficulties arise in this. When we know upper and lower bounds for a type variable (e.g. it is an `Object` and `Integer` is an subclass of it), we may need to try different candidates to establish the correct type — this leads to non-determinism, and hence backtracking search. Similarly overloaded methods require backtracking. This is why a set of extended substitutions must be produced. This corresponds to the notion given earlier that there is no unique type scheme for a program in MLj.

The set of solutions must be analysed after solving to pick the single 'best' (according to the rules introduced in Section 7.3) solution (if any).

7.5.1 Extended Substitutions and Upper and Lower Bounds

An extended substitution maps type variables to one of three things.

- A type (as in a normal substitution).
- GLB and LUB classes, interfaces, and type variables as upper and lower bounds on the variable (which must, therefore be a class), as shown in Figure 7.5.
- Other types as upper and lower bounds on the variable (which must, therefore, be a $\mathsf{JavaType}$).

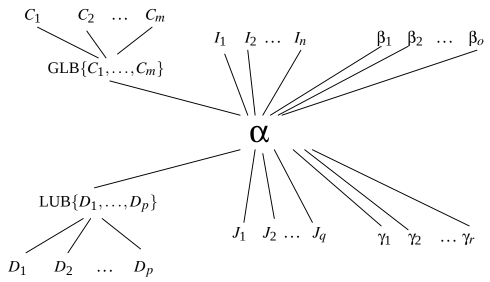
Figure 7.5 shows the upper and lower bounds of α, when this is a class. The upper bound on α is the greatest lower bound of all c_i, where $\alpha \leq_w c_i$ (this is `Object` if no such classes exist); all interfaces which α must implement; and all type variables above α. The lower bound (if any) is the least upper bound of all c_i such that $c_i \leq_w \alpha$, and all relevant type variables and interfaces.

If α is not known to be a class but appears in some subtyping constraint, LUBs and GLBs cannot be computed and all bounds must be recorded.

The extended substitution comes with operations for augmenting it with the following forms: $\alpha = \tau$, $\tau \leq_w \alpha$, $\alpha \leq_w \tau$, $\tau = \tau'$, $\tau \leq_w \tau'$ and $\{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$.

The first of these corresponds to adding a conventional substitution term. It may involve extra work however as we must check that τ is compatible with any existing bounds on α (or unify if

Figure 7.5 Upper and lower bounds of α, when it is a class.



we already know an exact type for α) and then add τ as a bound on any other type variables in place of α (this then involves recursive calls to add $\alpha \leq_w \tau$ or $\tau \leq_w \alpha$).

Similarly, adding $\tau \leq_w \alpha$ or $\alpha \leq_w \tau$ is like adding a single term to the extended substitution, though it may have wide ranging side effects. Ultimately, adding such a term can solve the type variable, for example if we know that $\tau \leq_w \alpha$ and add $\alpha \leq_w \tau$ then we infer $\alpha = \tau$.

Adding $\tau = \tau'$ or $\{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$ incorporates the result of unification into the substitution. Adding $\tau \leq_w \tau'$ has an analogous effect for subtyping (both types should be JavaType).

In the extended substitution we can refine general widenings (of JavaTypes) into class widenings (on ClassTypes). For example if we know that c `option` $\leq_w \alpha$ then we can infer that $\alpha = \beta$ `option` for some new β and $c \leq \beta$.

7.5.2 Rewriting Subtyping Constraints to Simplify

The constraint solver works by taking constraints from a set and reducing then to simpler constraints, or incorporating them into the extended substitution.

Subtyping constraints are simplified by rewriting rules in which each constraint is replaced by a new set of constraints. Of the resulting constraints, those which involve equality, $=$, or widening, $\leq_w$, are added to the extended substitution. The rewriting rules necessary to solve $\leq_{\mathsf{memb}}$ constraints are given in an SML pattern matching style notation (if more than one pattern fits any particular constraint, the first one listed should be used) in Figure 7.6.

Given a constraint of the form $\tau \leq_{\mathsf{memb}} \tau'$, if either type is known to be a function type then the member in question is a method and method subtyping applies. If both types are type variables, then we do not know whether the member is a method or a field and cannot simplify the constraint. Otherwise, one type must be known to be a non-function type (e.g. `int`) so the

Figure 7.6 Some rewriting rules.

$$
\begin{array}{rcl}
\tau_0 \to \tau_1 \leq_{\mathsf{memb}} \tau & \Rightarrow & \{\tau_0 \to \tau_1 \leq_{\mathsf{meth}} \tau\} \\
\tau \leq_{\mathsf{memb}} \tau_0 \to \tau_1 & \Rightarrow & \{\tau \leq_{\mathsf{meth}} \tau_0 \to \tau_1\} \\
\alpha \leq_{\mathsf{memb}} \alpha' & \Rightarrow & \{\alpha \leq_{\mathsf{memb}} \alpha'\} \quad \text{No change} \\
\tau \leq_{\mathsf{memb}} \tau' & \Rightarrow & \{\tau = \tau'\} \quad \text{Field types} \\
\\
\tau_0 \to \tau_1 \leq_{\mathsf{meth}} \tau'_0 \to \tau'_1 & \Rightarrow & \{\tau_0 \leq_{\mathsf{ArgVec}} \tau'_0, \tau_1 = \tau'_1\} \\
\tau_0 \to \tau_1 \leq_{\mathsf{meth}} \alpha & \Rightarrow & \{\alpha = \alpha_0 \to \alpha_1, \tau_0 \leq_{\mathsf{ArgVec}} \alpha_0, \tau_1 = \alpha_1\} \\
\alpha \leq_{\mathsf{meth}} \tau_0 \to \tau_1 & \Rightarrow & \{\alpha = \alpha_0 \to \alpha_1, \alpha_0 \leq_{\mathsf{ArgVec}} \tau_0, \alpha_1 = \tau_1,\} \\
\alpha \leq_{\mathsf{meth}} \alpha' & \Rightarrow & \{\alpha \leq_{\mathsf{meth}} \alpha'\} \quad \text{No change} \\
\\
\tau_1 \times \ldots \times \tau_n \leq_{\mathsf{ArgVec}} \tau'_1 \times \ldots \times \tau'_n & \Rightarrow & \{\tau_1 \leq_{\mathsf{Arg}} \tau'_1, \ldots, \tau_n \leq_{\mathsf{Arg}} \tau'_n\} \\
\tau_1 \times \ldots \times \tau_n \leq_{\mathsf{ArgVec}} \alpha & \Rightarrow & \{\alpha = \alpha_1 \times \ldots \times \alpha_n, \tau_1 \leq_{\mathsf{Arg}} \alpha_1, \ldots, \tau_n \leq_{\mathsf{Arg}} \alpha_n\} \\
\alpha \leq_{\mathsf{ArgVec}} \tau_1 \times \ldots \times \tau_n & \Rightarrow & \{\alpha = \alpha_1 \times \ldots \times \alpha_n, \alpha_1 \leq_{\mathsf{Arg}} \tau_1, \ldots, \alpha_n \leq_{\mathsf{Arg}} \tau_n\} \\
\alpha \leq_{\mathsf{ArgVec}} \alpha' & \Rightarrow & \{\alpha \leq_{\mathsf{ArgVec}} \alpha'\} \quad \text{No change} \\
\tau \leq_{\mathsf{ArgVec}} \tau' & \Rightarrow & \{\tau \leq_{\mathsf{Arg}} \tau'\} \quad \text{Non-tuple types} \\
\\
\tau \leq_{\mathsf{Arg}} \tau' \quad \text{Either type is JavaType} & \Rightarrow & \{\tau \leq_w \tau'\} \\
\tau \leq_{\mathsf{Arg}} \tau' \quad \text{Either type is non-JavaType} & \Rightarrow & \{\tau = \tau'\} \\
\tau \leq_{\mathsf{Arg}} \tau' \quad \text{Can't distinguish} & \Rightarrow & \{\tau \leq_{\mathsf{Arg}} \tau'\} \quad \text{No change} \\
\\
c \leq_{\mathsf{sup}} \mathsf{super}(c) & \Rightarrow & \{\} \quad \text{Eliminate} \\
c \leq_{\mathsf{sup}} \alpha & \Rightarrow & \{\alpha = \mathsf{super}(c)\} \\
\alpha \leq_{\mathsf{sup}} c & \Rightarrow & \{\alpha \leq_{\mathsf{sup}} c\} \quad \text{No change} \\
\alpha \leq_{\mathsf{sup}} \alpha' & \Rightarrow & \{\alpha \leq_{\mathsf{sup}} \alpha'\} \quad \text{No change}
\end{array}
$$

member in question must be a field. As there is no subtyping on fields, remove the constraint and add $\tau = \tau'$ to the extended substitution. Method subtyping is like a simplified version of $\leq_{\mathsf{memb}}$ in which the member must be a method.

In Java, methods can have several arguments. In MLj, this is simulated by methods having tuples as arguments. We have argument subtyping on each of the elements of the tuple, this means that the argument must either be exactly the right type, or a JavaType and a subtype of the required argument type. Given constraints of form $\tau \leq_{\mathsf{ArgVec}} \tau'$, if either type is known to be a tuple, then the other must be a tuple of the same length, and argument subtyping applies to all tuple fields. If either type is known not to be a tuple, then argument subtyping applies to the types.

Given constraints of form $\tau \leq_{\mathsf{Arg}} \tau'$, if either type is known to be a JavaType then widening applies (this is handled by the extended substitution), otherwise the types must be exactly equal.

We use the definition in Figure 7.1 and information in the extended substitution to tell whether a type is a JavaType. A typical case which cannot be distinguished is α `option`.

If we know the subclass of τ in a constraint of the form $\tau \leq_{\mathsf{sup}} \tau'$, then we can instantiate the immediate superclass. The converse does not hold, however, as super-classes can have an arbitrary number of immediate subclasses. (The open world assumption prevents us from instantiating the subclass even if only one candidate exists — others may be added at a future date).

Cast subtyping constraints, $\tau \leq_{\mathsf{Cast}} \tau'$ or $\tau \geq_n \tau'$, are retained and checked after the other constraints have been solved. As they can involve either widening or narrowing, they do not help to solve the constraint set.

7.5.3 'Has' Constraints

'Has' constraints provide problems because of the arbitrary overloading of methods. If we know the class in a 'has' constraint then we try each of the possible types for the member, thus constraint solving can produce several solutions (see Section 7.3). This part of the algorithm is currently under development. It is necessary to consider in particular how to deal with 'has' constraints involving a variable with known bounds. For example if we have upper and lower bounds on α we can make certain assumptions about the methods it might have, and it may be necessary to try types for the methods in order to establish which class it is.

7.5.4 Selecting the Correct Solution

The result of solving the constraints is a set of extended substitutions, each of which satisfies the constraints. The final stage of type inference is to decide which (if any) substitution is to be used. The substitutions are applied to the method subtyping constraints in the original set of constraints and the results are compared to see which substitution gives the most specific method for *every* constraint. It may be that no substitution is appropriate: some may be incomparable as in the examples which had two equally reasonable types.

This is the point at which constraint solving can fail (i.e. discover that the program is not well-typed) and the point at which error messages must be produced.

7.6 Type Errors in MLj

Constraint solving can result in any number of solutions, each solution is a set of unsolved constraints and an extended substitution. There are a number of possible outcomes.

- There are no solutions. This means that there is an inconsistency somewhere. The program is rejected.

- There are no solutions without unsolved constraints. This means that there is not enough information in the program to resolve the types. The program is rejected.

- There are one or more solutions without unsolved constraints, and it is possible to pick one which gives the most specific method for every constraint. In this case the program can be accepted (but the programmer may still need assistance understanding why the solution was selected).

- There are one or more solutions without unsolved constraints, but it was not possible to select a single one. The program is rejected.

Each of the three ways in which the program can be rejected, and the case in which the program is accepted are considered in the sections below. The methods suggested below represent proposals for the implementation and have not been tried.

7.6.1 Inconsistent Programs

If there are no solutions to a set of constraints, then the program can be said to be inconsistent. This could occur if an object is required to have a type in two different branches of the type hierarchy, as seen in the example below

```
(* Class A has m : Integer -> unit
   Class B has m : String -> unit *)
(* Create new instances of A and B. *)
val a = A() val b = B()
fun f x = (a.#m x ; b.#m x)
```

There is no type which can be assigned to `x` because the constraints $\alpha_x \leq_w$ `Integer` and $\alpha_x \leq_w$ `String` are inconsistent. This is similar to type errors that can occur without subtyping, for example

```
fun a x = print (Int.toString x)
fun b x = print x

fun f x = (a x ; b x)
```

While it is true that similar mistakes should be announced with similar messages, current error messages for the pure ML version of the program are generally unsatisfactory. Most implementations of compilers will announce the second application as incorrect (as `x` will have been established as an `int` by the first application). Using unification of substitutions (Section 4.1.2) we will be able to find the inconsistency between the two uses of `x`.

The inconsistent extended substitution which constraint solving attempts to produce contains ideal information for an error message. It knows that both `Integer` and `String` must be higher than the type of `x` in the hierarchy and this can be announced to the programmer. For example

```
Type Error:
  x cannot be a subtype of Integer and subtype of String
```

If the constraints are augmented with explanations in the style of Duggan [Dug98] then the error message can be augmented with an explanation of where these constraints came from, e.g.

```
Type Error:
  x cannot be
    a subtype of Integer (required by "a.#m x")
  and
    a subtype of String (required by "b.#m x")
```

Note that the MLj program given can be made to type check correctly by adding coercions (`x :> Integer` and `x :> String`) but it will not be able to execute without a run-time exception. Suggesting coercions, therefore, will appear to get rid of the type error but will actually just delay its detection until run-time (and is therefore not desirable and against the "spirit of ML").

7.6.2 Unsolvable Programs

If there is no way to reduce all of the constraints to extended substitutions then the program is unsolvable. One reason for this happening is the open world assumption. The example given earlier will serve to illustrate this

```
_classtype C ()
  with
    qwexiphlibit () = ()
  end
fun f obj = obj.#qwexiphlibit ()
(* Program rejected. *)
```

In this case the programmer needs to know which identifier is missing information. For example

```
Type Error:
  Not enough information to infer type for obj.
```

It is important to stress that the problem is that there is not enough information to decide on a unique type and that there is not a conflict of types (so there is no type error in the conventional ML way).

In the spirit of spell-checking, in this case it could possibly be suggested that the type may be supposed to be `C` (this information can be obtained from the 'has' constraint) or a reminder that "classes cannot be inferred from method use" could be given to help the programmer. Listing the constraints known involving the type may also help the programmer.

Generic advice (as opposed to specific information about the program in question) is not seen in type error messages, but is seen in other domains (for example "check the spelling of the URL"). It would certainly be useful for programmers getting used to a new language, which is particularly relevant to MLj, but are likely to annoy more experienced programmers who will see the same advice repeatedly.

7.6.3 Ambiguous Programs

If there are several ways to reduce a set of constraints to extended substitutions, then the program is ambiguous. This is similar to (though more subtle than) the previous case of unsolvable programs.

One example of such a case is the identity method.

```
_classtype I ()
  with
    I x = x
  end
val i = I () (* New instance of class I *)
val r = i.#I (Integer 3)
(* Program rejected. *)
```

For this case it will probably be necessary to list some 'reasonable' types for the method in question. For example

```
Type Error:
  Cannot find unique best type for I.I
  Possibilities include
    Integer -> Integer
    Object option -> Object option
```

As in the case of unsolvable constraints, care should be taken to state that the problem is not an inconsistency, but a lack of information.

7.6.4 Accepted Programs

A program is accepted when the constraints can be reduced to extended substitutions, and a single solution selects more specific method calls than any other.

An error message does not need to be given in this case, but programmers may want to know which types were assigned and how the constraints were solved. Sometimes this information is essential for MLj: if a class is being exported for use with Java programs, the programmer needs to be sure what type all its methods have. This information can be obtained from compiled code, but it would be more convenient to have MLj report it.

The types assigned inside a program could also affect the dynamic semantics of a program. For example, one version of an overloaded method may have undesirable side effects and the programmer may wish to be sure that version is not used. The following class represents a user interface for a bomb.

```
_classtype BombControl ()
  with local
    val b = Bomb() (* Create a new bomb *)
  in
    control (o : Integer) = b.explode()
    control (o : String) = b.defuse()
  end
```

The dynamic behaviour of any instance of this class is decided by the types assigned at compile time.

As well as knowing the types inferred, a programmer will also be helped by information about how they were inferred (for example Duggan-style explanations). This will help the programmer to see where in the program to place coercions to get the desired types.

Chapter 8

Conclusions

The following four sections take stock of the previous four chapters of results. In addition to the chapters of results, Chapters 1–3 discussed the deficiencies of existing type inference technology and other author's proposed means for solving these deficiencies.

8.1 Order of Type Inference

Chapter 4 was based upon the observation that existing type inference algorithms (including W and M) are non-compositional and have asymmetries in the way that they treat subexpressions. It also noted that by changing the order in which subexpressions are examined, one can generate different (and more useful) error messages.

8.1.1 Asymmetry and U_S

The classic type inference algorithm W has an asymmetry, which can be described as a 'left-to-right bias' in its cases for application expressions and tuples. The asymmetry is that when type checking $e_0\ e_1$ against an environment Γ, first e_0 is checked against Γ to obtain a substitution S_0 and then e_1 is checked against the modified environment $S_0\Gamma$. Removing the asymmetry necessitates type checking both e_0 and e_1 against Γ and obtaining two substitutions S_0 and S_1. A type inference algorithm, W^{SYM}, is described which does have recursive calls of this form. W^{SYM} then uses a new function, U_S, to unify these two substitutions and obtain the same substitution as W.

Proofs of the soundness and completeness of U_S and W^{SYM} are in Appendix A.

W^{SYM} is also symmetric in the case for tuples and a symmetric version of M (called M^{SYM}) is given as well.

8.1.2 Other Syntactic Structures

As well as their asymmetries, type inference algorithms suffer from following the formal syntax of the language, rather than looking at programs in the same way as programmers. For example, programmers think of curried application expressions, $e_0\ e_2\ \ldots\ e_n$, as the application of several arguments to a single function, whereas language semantics and compilers view these expressions as nested application expressions, $(\ldots((e_0\ e_2)\ e_3)\ldots)e_n$. There are n unifications used to type check such expressions, corresponding to the n stages in the computation. Curried expressions are the source of many confusing error messages as these can be produced at any of the n unifications, all of which are related to subexpressions of what the programmer sees as a simple application. Modifying type inference so that there is only a single unification of a *function type* and *context type* (created from all the argument types) produces more cogent error messages. The treatment of curried expressions in this way is used in Chapter 6.

The type checking of SML structures and signatures also does not fit the programmer's expectations because it follows the syntax of the language rather than looking at programs in the same way as people.

8.2 Graphs for Type Inference

An alternative way of doing type inference is described in Chapter 5.

8.2.1 Graphs for Typeable and Untypeable Programs

Existing algorithms for type inference all fail when there is a type error and the methods proposed by other authors as means for producing better error messages and assistance for programmers are ways of dealing with failure. In contrast, this chapter introduces a form of graph which can capture information about the types in programs, whether or not they are typeable.

Type inference using these graphs takes place in two distinct phases. First a graph is generated for the program. Second the graph is analysed to discover whether the program was typed and what type (if any) it has.

8.2.2 Extracting Further Information from Graphs

The reason why graphs are useful is that as well as the basic information about whether a program is typeable and what type it has, other useful information about programs is contained in them. This is shown in Section 5.4 by the fact that work proposed by other authors as a means to help programmers can also be extracted from the graphs.

It is informally demonstrated that the type explanations generated by Dominic Duggan's SML/E program can be extracted from graphs, and shown that these explanations do not, in fact, meet the specification for 'correct type explanations' given by Duggan.

8.3 Repairing Mistakes with Type Isomorphisms

While Chapter 5 described a new way of performing type inference, Chapter 6 describes a completely new form of error message which can be generated by extensions to conventional type inference algorithms.

8.3.1 New Error Messages

The new error messages describe how the program can be modified to remove type errors. For example

```
Try changing
  map ([1, 2, 3], Int.toString)
To
  map Int.toString [1, 2, 3]
```

These messages directly address the need of programmers, which is to repair the program. Other information only helps programmers indirectly by explaining the types in programs in the hope that this helps them devise a way of repairing the program.

A prototype of these error messages has been implemented for the MLj compiler. The chapter also describes how the implementation can be integrated into other compilers. Extracts from the source code appear in Appendix B.

8.3.2 Design and Implementation

The new error messages are produced by the application of the theory of type isomorphisms. Type inference fails when two types fail to unify, that is when there is no substitution of types for type variables which can make the types equal. There may, however, be a substitution which can make the types isomorphic. Two types are isomorphic if there exist a pair of functions (morphisms) which can transform values from one type to the other and back again to the original value.

If the type of a function, and the type its context requires it to have are not unifiable to equality, but are unifiable modulo isomorphism then one of the morphisms (the repair morphism) can be used to repair the mistake in the program by converting the function to a similar function which fits into the context.

The new version of the expression shown in the error messages is obtained by partially evaluating the application of the repair morphism.

8.3.3 New Algorithms

The development of these new error messages required several new algorithms to be devised and applied.

- The type inference algorithm is modified to type check curried expressions with a single unification.
- An existing algorithm for unification modulo linear isomorphism is extended to produce morphisms as well as substitutions.
- In order to modify unification modulo linear isomorphism, it was also necessary to modify an algorithm for associative-commutative unification so that it produces morphisms.
- A partial evaluation semantics for applying repair morphisms was devised.

8.3.4 Future Investigations

This application of theory is an area ripe for future investigations (and useful implementations). One way in which the unification algorithms could be improved is to allow pseudo-isomorphisms as well as the linear isomorphisms currently implemented. These pseudo-isomorphisms could include such injective morphisms such as coercion to a list (which cannot be reversed and is not, therefore, an isomorphism). The description of the unification algorithms mentions where these changes can be added, but it remains to be seen which coercions correspond to common programming mistakes and will be of help to programmers.

Another way in which the implementation could be improved is by tracking down alternative sites where morphisms can be inserted (rather than simply trying to insert them where unification fails). There is a large body of work on locating type errors to draw from when investigating this.

Lastly, to fully realise its potential this work needs to be implemented in more compilers. These implementations should be accompanied by studies with users to establish the exact form of message which is most useful in practice.

8.4 Beyond Hindley-Milner: the MLj Type System

Most of the work in this thesis is based on the Hindley-Milner type system. Programming languages generally use some extension of this, a prime example of which is MLj.

MLj is an extension of Standard ML which offers integration with Java's class hierarchy. To accommodate this, the type system is augmented with a subtyping relation representing the hierarchy of Java classes.

Type inference in MLj involves a phase of constraint solving which introduces new forms of type errors. For example a program may be untypeable because there is no way to resolve its set of constraints, or because there are several distinct ways to resolve the constraints. Ways of helping programmers repair mistakes involving these problems are discussed in Chapter 7.

8.5 Closing Remarks

This thesis has shown that there are several ways in which type error messages can be improved, to the point of even suggesting ways of repairing programs containing such errors.

The usability of compilers is an important factor in their adoption and popularity, and also an important factor in the popularity of the programming languages which they compile. It is essential, therefore, that if strongly typed programming languages are to gain greater popularity and usage that the ideas in this thesis are applied to future products.

Appendix A

Proofs For Chapter 4

A.1 Proof of Theorem 5

For any pair of substitutions, S_0 and S_1, if $U_S(S_0, S_1)$ succeeds then it returns a unifying substitution. (U_S is defined in Figure 4.2.)

Proof

First show that S'_n unifies S_0 and S_1 over the domain $D_0 \cup D_1$n (i.e. unifies T_0 and T_1).

Proposition For all S'_i, $\forall \alpha \in D_0 \cup \{\alpha_1 \cdots \alpha_i\} : S'_i$ is well formed (the α_j terms are from D_1), and $S'_i S_0 \alpha = S'_i S_1 \alpha$.

Proof by induction on $i = 0, 1, \ldots, n-1$

Base case $S'_0 = T_0$

Only concerned with type-variables in support of S_0 and not in support of S_1, $\{\alpha : \alpha \in D_0 \wedge \alpha \notin D_1\}$, so $S'_0 S_0 \alpha = S_0 \alpha$ and $S'_0 S_1 \alpha = S_0 \alpha$.

Induction step $S'_{i+1} = \{\alpha_{i+1} \mapsto S'_i \tau_{i+1}\} S'_i$

$\alpha_{i+1} \notin FV(S'_i \tau_{i+1})$ (as otherwise the algorithm fails with an occurs error)

So the result is an idempotent substitution.

For all α in $\mathrm{supp}(S_0) \cup \{\alpha_1 \cdots \alpha_i\}$: $S'_{i+1} S_0 \alpha = S'_i S_0 \alpha = S'_i S_1 \alpha$ (by induction hypothesis) $= S'_{i+1} S_1 \alpha$

For α_{i+1}: $S'_{i+1} S_1 \alpha = S'_{i+1} \tau_{i+1} = S'_i \tau_{i+1}$ and $S'_{i+1} S_0 \alpha_{i+1} = S'_i \tau_{i+1}$

Now show that V_m unifies over the support $\mathrm{supp}(S_0) \cup \mathrm{supp}(S_1) \cup FV(S_0) \cup FV(S_1)$.

Proposition For all V_i, V_i is well formed, and $\forall\alpha \in D_0 \cup D_1 \cup FV(S_0) \cup FV(S_1) \cup \{\beta_1 \cdots \beta_i\} : V_i S_0 \alpha = V_i S_1 \alpha$

Proof induction on i

Base case V_0

Leaves all variables but those in $D_0 \cup D_1$ unchanged, unifies on those in $D_0 \cup D_1$.

Induction step $V_{i+1} = U(\tau_0, \tau_1)V_i$

Need only consider β_{i+1} since by induction hypothesis everything else is unified by V_i.

Since $U(\tau_0, \tau_1)$ unifies $V_i S_0 \beta_{i+1}$ and $V_i S_1 \beta_{i+1}$, V_{i+1} unifies over the appropriate support.

So V_m unifies S_0 and S_1 over the support $\mathrm{supp}(S_0) \cup \mathrm{supp}(S_1) \cup FV(S_0) \cup FV(S_1)$ and leaves all other type variables unchanged. Thus V_m unifies S_0 and S_1.

□

A.2 Proof of Theorem 6

If S'' unifies S_0 and S_1 then

1. *$U_S(S_0, S_1)$ succeeds returning S', and*
2. *there is some R such that $S'' = RS'$.*

(U_S is defined in Figure 4.2.)

Proof

The proof follows a similar pattern to the previous result.

First show a property for the substitutions $S'_0 \cdots S'_n$ — that they do exist (there are no occurs errors) and how they relate to S''.

Proposition Each S'_i exists (there are no occurs errors) and for each S'_i there is a substitution X_i such that $\forall\alpha \in D_0 \cup \{\alpha_1 \cdots \alpha_i\} : X_i S'_i S_0 \alpha = S'' S_0 \alpha$. All these substitutions are idempotent.

Proof by induction on i

Base Case $S_0' = \{\alpha \mapsto S_0\alpha : \alpha \in D_0\}$

Know that $\forall\alpha : S''S_0\alpha \geq S_0\alpha$

And $\forall\alpha : S_0'S_0\alpha = S_0\alpha$

So X_0 exists.

Induction Step $S_{i+1}' = \{\alpha_{i+1} \mapsto S_i'\tau_{i+1}\}S_i'$

First show there is no occurs error by showing $\alpha_{i+1} \notin FV(S_i'\tau_{i+1})$.

Since $S''S_1$ and X_i exist, the occurs error cannot happen.

And that $S_i'\tau_{i+1} > S''S_0\alpha_{i+1}$

Know $\tau_{i+1} > S''S_0\alpha$ and terms of S_i' only effect the limited support (which is $\subseteq \mathrm{supp}(S'')$). So, by I.H. this holds.

So, by I.H. result holds for appropriate support and X_{i+1} exists.

Also, we can see that $\mathrm{supp}(S_n') = D_0 \cup D_1$.

Now show a similar result for the sequence $V_1 \cdots V_m$

Proposition Each V_i exists (the unification succeeds and there are no occurs errors); and for each V_i, there is a Y_i such that $\forall\alpha \in D_0 \cup D_1 \cup \{FV(S_0\beta_x) \cup FV(S_1\beta_x) \cup \{\beta_x\} : 0 \leq x \leq i\}.Y_iV_iS_0\alpha = S''S_0\alpha$

Proof Induction on i

Base Case $V_0 = S_n'$

This comes directly from the previous result.

Induction Step $V_{i+1} = U(U_iS_0\beta_{i+1}, V_iS_1\beta_{i+1})U_i$

Must show that U succeeds. Since $S''S_0\beta_{i+1} = S''S_1\beta_{i+1}$, there must be a type which is the unification of the two types $V_iS_0\beta_{i+1}$ and $V_iS_1\beta_{i+1}$, so U succeeds.

There is no occurs error (for similar reason to those in the previous proof)

Since U is the most general unifier, Y_{i+1} exists.

So U_S succeeds with $S' = V_m$ and, $Y_mU_mS_0\alpha = S''S_0\alpha$ for all α in the support above. Since all other type-variables are invariant under $Y_mU_mS_0$, it is trivial to provide a substitution, R', such that $R'Y_mU_mS_0 = S''S_0$. So R exists and is $R'Y_m$.

$\square$

A.3 Proof of Theorem 7

If $W^{SYM}(\Gamma, e)$ succeeds with (S, τ) then there is a derivation of $S\Gamma \vdash e : \tau$. (W^{SYM} is defined in Figure 4.3.)

Proof Damas [Dam85] gives a proof of this theorem for W by induction on the structure of e. As W^{SYM} differs from W only in the case of application and tuples, it is sufficient to prove these cases. The tuple case is omitted here.

Case $e = e_0 e_1$

By the induction hypothesis, we know $W^{SYM}(\Gamma, e_0) = (S_0, \tau_0)$ and $S_0\Gamma \vdash e_0 : \tau_0$; and $W^{SYM}(\Gamma, e_1) = (S_1, \tau_1)$ and $S_1\Gamma \vdash e_1 : \tau_1$

And since $W^{SYM}(\Gamma, e)$ succeeds, that $S' = U_S(S_0, S_1)$, $V = U(S'\tau_0, S'\tau_1 \rightarrow \beta)$

From the definition of U_S, let $S = S'S_0 = S'S_1$, and from that of U we know $VS'\tau_1 \rightarrow \beta = VS'\tau_0$.

And the final result $W^{SYM}(\Gamma, e) = (VS, V\beta)$.

Must show that there is a derivation of $VS\Gamma \vdash e_0 e_1 : V\beta$

The derivation will end

$$\frac{VS\Gamma \vdash e_0 : VS'\tau_1 \rightarrow V\beta \qquad VS\Gamma \vdash e_1 : VS'\tau_1}{VS\Gamma \vdash e_0 e_1 : V\beta}$$

We already know (from I.H. above) that derivations of $S_0\Gamma \vdash e_0 : \tau_0$ and $S_1\Gamma \vdash e_1 : \tau_1$ exist, so by proposition 2 of [DM82] derivations of $S'S_0\Gamma \vdash e_0 : S'\tau_0$ and $S'S_1\Gamma \vdash e_1 : S'\tau_1$ also exist.

So derivations of $VS\Gamma \vdash e_0 : VS'\tau_0$ (the type here is $VS'\tau_1 \rightarrow V\beta$) and $VS\Gamma \vdash e_1 : VS'\tau_1$ also exist .

So the derivation of $VS\Gamma \vdash e_0 e_1 : V\beta$ exists.

The other cases are a simple analogue of ones which have been shown by Damas, so the theorem holds in general. □

A.4 Proof of Theorem 8

Given Γ *and* e*, let* Γ' *be an instance of* Γ *and* η *be a type scheme such that* $\Gamma' \vdash e : \eta$.
Then

1. $W^{SYM}(\Gamma, e)$ *succeeds*

2. *If* $W^{SYM}(\Gamma, e) = (P, \pi)$ *then for some* R*:* $\Gamma' = RP\Gamma$*, and* η *is a generic instance of* $R\overline{P\Gamma}(\pi)$.

(W^{SYM} *is defined in Figure 4.3.)*

Proof Damas provides a proof for this theorem for W on the structure of the derivation of $\Gamma' \vdash e : \eta$. As W^{SYM} differs from W only in the case that $e = e_0 e_1$, it is sufficient to present only the inductive step for this case.

Case $e = e_0 e_1$

The derivation ends

$$\frac{\Gamma' \vdash e_0 : \tau' \to \tau \qquad \Gamma' \vdash e_1 : \tau'}{\Gamma' \vdash e_0 e_1 : \tau}$$

for some τ'.

By the induction hypothesis we know that $W^{SYM}(\Gamma, e_0)$ succeeds, call the result (S_0, π_0)

By condition 2 of the induction hypothesis there is a substitution R_0 such that $\Gamma' = R_0 S_0 \Gamma$ and $\tau' \to \tau$ is a generic instance of $R_0 \overline{S_0\Gamma}(\pi_0)$.

Let $\alpha_1 \cdots \alpha_n$ be the generic type-variables in π_0 (these occur in π_0 but are not free in $S_0\Gamma$).

R_0 leaves all α_i unchanged since it is minimal (which means $\text{supp}(R_0) \subseteq FV(S_0\Gamma)$).

Since $\tau' \to \tau$ is a generic instance of $R_0(\forall \alpha_1 \cdots \alpha_n . \pi_0)$ (the scheme here is the closure $\overline{S_0\Gamma}(\pi_0)$), and R_0 leaves all $\alpha_1 \cdots \alpha_n$ unchanged: there are types $\tau_1 \cdots \tau_n$ such that $\tau' \to \tau = (R_0 + \{\alpha_i \mapsto \tau_i\})\pi_0$.

Likewise, for e_1: R_1, $\beta_1 \cdots \beta_m$ and $\tau'_1 \cdots \tau'_m$ exist, and $\tau' = (R_1 + \{\beta_j \mapsto \tau'_j\})\pi_1$.

First, show that $W_S(S_0, S_1)$ succeeds. To do this exhibit a substitution which is a unification of S_0 and S_1.

Note that $R_0 S_0 \Gamma = \Gamma' = R_1 S_1 \Gamma$, so $\forall \alpha \in FV(\Gamma) : (R_0 S_0)\alpha = (R_1 S_1)\alpha$.

And note that $\text{supp}(S_0) \subseteq \text{supp}(\Gamma) \cup \text{new}_0$ (where new_0 is the set of new type variables produced by $W^{SYM}(\Gamma, e_0)$.). Similarly $\text{supp}(S_1) \subseteq \text{supp}(\Gamma) \cup \text{new}_1$.

Let $S = \{\alpha \mapsto R_0 S_0 \alpha : \alpha \in FV(\Gamma)\} + \{\alpha \mapsto R_0 S_0 \alpha : \alpha \in \text{new}_0\} + \{\alpha \mapsto R_1 S_1 \alpha : \alpha \in \text{new}_1\}$.

S is a unification of S_0 and S_1. Since a unified substitution exists, $U_S(S_0, S_1)$ will terminate

and returns some unifying substitution S', and there is some S'' such that $S = S''S'S_0$.
Now it is necessary to show $U(S'\pi_0, S'\pi_1 \to \beta)$ (β is new) succeeds. To do this, exhibit a unifying substitution.
Let $U_0 = \{\alpha_i \mapsto \tau_i, \beta_j \mapsto \tau'_j, \beta \mapsto \tau\} + S''$.
First show that U_0 is a well formed substitution and then that it is a unifying substitution.
The type variables $\alpha_1 \cdots \alpha_n, \beta_1 \cdots \beta_n$ and β are all distinct. β does not occur in supp(S'').
Must also show none of $\alpha_1 \cdots \alpha_n, \beta_1 \cdots \beta_n$ are in supp(S''). supp$(S'') \subseteq FV(\Gamma) \cup \text{new}_0 \cup \text{new}_1$, so none of the α_i, β_j occur in this support. So U_0 is a well formed substitution.
We know $\tau' \to \tau = (R_0 + \{\alpha_i \mapsto \tau_i\})\pi_0$ so $\tau' \to \tau = U_0S'\pi_0$. Also $\tau' = (R_1 + \{\beta_j \mapsto \tau'_j\})\pi_1$ so $\tau' \to \tau = U_0(S'\pi_1 \to \beta)$. So U_0 is unifying substitution for $S'\pi_0$ and $S'\pi_1 \to \beta$.
Since a unifying substitution exists, $U(S'\pi_0, S'\pi_1 \to \beta)$ succeeds and returns V, and there is some substitution V' such that $U_0 = V'V$.
It remains to show that the result $W(\Gamma, e) = (VS, V\beta)$ satisfies condition 2.
Must show that there is some substitution R such that $\Gamma' = RVS'S_0\Gamma$ and τ is a generic instance of $R\overline{VS'S_0\Gamma}(\tau)$. It is clear that such an R can be constructed, so this condition is satisfied.

Since the induction case $e = e_0e_1$ holds, and the other cases are a simple analogue of ones which have been shown by Damas, the theorem holds in general. $\square$

Appendix B

Source Code for Chapter 6

Chapter 6 described how to use unification modulo linear isomorphisms to find ways to rearrange programs that repair type errors, and how to use partial evaluation to produce error messages from these.

This work has been added to the MLj compiler as described in Section 6.4 and this appendix contains extracts from the source code. The full implementation is available in source code form from `http://www.dcs.ed.ac.uk/home/bjm/` and the software is distributed according to the following restrictions

The implementation consists of a new module for generating error messages, and a small number of changes to existing MLj source code to call these. The new parts of the program are intended to be suitable for integration with any type inference system written in Standard ML. The changes to the existing code are described in Section B.1 and the excerpts from the new parts are in Section B.2. At the end of the appendix is a short description of how to test the system (Section B.3).

B.1 Changes to Existing MLj Source Code

The main change to the existing MLj source code was to get type inference to follow curried expressions and to call the new routines when it finds a type error in application expressions. It was also necessary to disable error message printing in the existing unification routine. Below is the modified type inference routine for curried expressions (taken from the structure ElabCore)

```
(* This is the 'curried form' version of the Rule 8 code *)

| App(func, arg) =>
      let
          (* If func is an application then we have a
           curried expression.  The following function
           finds the curried function and the list of
           arguments. (It also stores the locations of the
           individual applications.) *)

          fun splitFunc ((loc, App(e, e')), eList, locList) =
                splitFunc(e, e'::eList, loc::locList)
            | splitFunc (e, eList, locList) = (e, eList, locList)

          val (e0, es, locs) = splitFunc(func, [arg], [loc])

          (* Type check the function expression *)
          val e0Result = infExpOrJava C e0
      in
          case e0Result of
              Result (e0', t0) =>
                  (* The function is not a Java static method. *)
                  let

                      (* Type check arguments *)
                      val esAndTs = map (infExp C) es

                      (* Build context type from
                         types of the arguments *)
                      val beta = SMLTy.freshType()
                      fun build ((e', t), (es', t')) =
                          (e'::es', SMLTy.funType(t, t'))
                      val (es', ts) =
                          List.foldr build ([], beta) esAndTs

                      (* we now have
                       e0' - elaborated e0
                       es' - list of other elaborated expressions
                       t0  - type of e0
                       ts  - context type *)

                      (* Convert the relevant types into the form
```

```
            for type debugging. *)
           val t0LI = ConvertTypes.convertType t0
           val tsLI = ConvertTypes.convertType ts

           (* Do the unification *)
           val (_, unifyErr) =
               unifyNoReport ((SOME loc,
                               "actual function type",
                               t0),
                              (SOME loc,
                               "context requires function type",
                               ts))

           (* Generate error message if necessary. *)
           val _ = if unifyErr then
               let
                   val _ =
               (case TyDebug.tydebug(t0LI, tsLI, e0::es) of
                    SOME s =>
                        (SMLTy.error
                         (Error.error (loc, s), []))
                  | NONE =>
                        (SMLTy.error
                         (Error.error
                          (loc,
                           ("Type error found at application expression\n      "^
                            (TyDebug.mlToString exp))),
                          [("Inferred function type",
                            t0),
                           ("Inferred context type",
                            ts)]))
                        )
               in
                   ()
               end
                   else
                       ()

           (* Construct the elaborated expression *)
           val e' = List.foldl
               (fn (fromEs, e') => T.App(e', fromEs))
               e0' es'
       in
           (e', beta)
       end (* of non java case of Rule 8 *)

 | JavaResult (class, name) =>
       (* The function is a static method, so
          rebuild the curried application with
```

```
                a Java invocation at the front. *)
              let
                (* from the list of curried arguments,
                   get the argument for the static method,
                   and the other arguments. *)

                val locsAndEs = ListPair.zip (locs, es)

                val (loc, firstArg, args) =
                    case hd locsAndEs of (loc, (_, Tuple exps)) =>
                        (loc, exps, tl locsAndEs)
                  | (loc, first) =>
                        (loc, [first], tl locsAndEs)
                  (* Build the java invocation. *)

                val javaExp =
                    (loc,
                     Java(Java.Invoke,
                          SOME(loc, TyClass (ClassHandle.name class)),
                          SOME name, firstArg))

              (* Build the curried expression (with an invocation
               at the front. *)
              val exp =
                  List.foldl
                  (fn ((loc, a), f) => (loc, App(f, a)))
                  javaExp
                  args
              in
                infExp C exp
              end (* Of static method case of rule 8 *)

    end (* of rule 8 *)
```

B.2 The New Modules

The new source code includes

- A front end to the type error messages (called from the MLj type inference routine). See Section B.2.1.

- A generic representation of types and a routine to convert MLj types into the generic representation for unification. See Section B.2.2

- Representation of and partial evaluation of expressions involving morphisms. See Section B.2.3

- Unification modulo linear isomorphism and AC-unification, generating morphisms. See Section B.2.4.

- Various library structures for example, for pretty-printing and lazy streams.

B.2.1 The Front End (TyDebug)

Structure TyDebug contains the front end of the system. To integrate with another compiler, this will probably have to be changed to produce output which fits in with the particular representation of error messages.

```
structure TyDebug =
    struct

        val version = "v0.1"

        open UnifyModuloLinearIso LITypes

        (* This should really come with the syntax structures. *)
        fun mlToString e =
            Isomorphisms.toString (Isomorphisms.toMExp e)

        (* tydebug returns a string option.
           NONE of there is no way to debug (therefore a more traditional
           error message must be produced).
           SOME "try changing ... to ... or ... or ..." if there
           is a way to debug. *)
        fun tydebug(funcType, contextType, expList) =
            let

                val _ = Patterns.resetId()
                val s = unify (funcType, contextType)
                val l = Stream.toList s
                val len = length l

                val oldMExpList = map Isomorphisms.toMExp expList
                val oldExpString =
                    ToString.spaceSep (map Isomorphisms.toString oldMExpList)

                fun mkMessage l =
                    let
                        val intro =
                            (" Try changing\n    " ^
                             oldExpString ^
```

```
                        "\n  to\n    ")
                in
                    intro ^
                    (ToString.sep
                       "\n  or\n    "
                       (map
                          (fn (_,m) =>
                             Isomorphisms.toString
                             (Isomorphisms.applyToML (m, expList)))
                          l))
                end

        in
            if len = 0 then
                NONE (* No ways to debug *)
            else
                SOME (mkMessage l)

        end handle _ => NONE


    end
```

B.2.2 Representation of Types

The representation of types is in structure LITypes (the name was chosen to be distinct from the Types structure already in MLj) and there is a routine to convert MLj types into this representation in structure ConvertTypes.

B.2.2.1 LITypes

```
structure LITypes :
    sig


        datatype types =
            FUN of types * types
          | TYVAR of string
          | CON of string * types list (* e.g CON("list", [...]) *)
          | TUPLE of types list

        val app : types * types -> types (* append as tuples *)
        val concat : types list -> types (* Concatenate as tuples *)

        val occurs : (string * types) -> bool
```

```
        val resetTyvar : unit -> unit
        val newTyvar : unit -> types

        val toString : types -> string
    end =
struct

    datatype types =
        FUN of types * types
      | TYVAR of string
      | CON of string * types list
      | TUPLE of types list

    fun app (TUPLE l, TUPLE l') = TUPLE (l @ l')
      | app (t, TUPLE []) = t
      | app (TUPLE [], t) = t
      | app (t, TUPLE l) = TUPLE(t::l)
      | app (TUPLE l, t) = TUPLE(l @ [t])
      | app (t, t') = TUPLE[t, t']

    val concat = (List.foldl app (TUPLE [])) o List.rev

    fun occurs (v, TYVAR v') = v=v'
      | occurs (v, FUN(t1, t2)) = occurs (v, t1) orelse occurs (v, t2)
      | occurs (v, TUPLE l) = List.exists (fn t => occurs (v, t)) l
      | occurs (v, CON(s, l)) =  List.exists (fn t => occurs (v, t)) l

    local
        val next = ref 0
    in
        fun resetTyvar () = next := 0
        fun newTyvar () =
            (next := !next + 1;
             TYVAR ("a" ^ (Int.toString (!next))))
    end

    fun  toString (FUN(t1, t2)) =
    "("^(toString t1)^" -> "^(toString t2)^")"
      | toString (TYVAR n) = "'"^n
      | toString (CON(c, [])) = c
      | toString (CON(c, [t])) =
        (toString t) ^ " " ^ c
      | toString (CON(c, l)) =
        "(" ^ (ToString.commaSep (map toString l)) ^ ") " ^ c
      | toString (TUPLE []) = "()"
      | toString (TUPLE [t]) =
        "{1 : " ^ (toString t) ^ " }"
        (* This is weird because of ML tuple derived syntax *)
      | toString (TUPLE l) =
```

```
        let
            fun f [] = ""
              | f [t] = toString t
              | f (h::t) = (toString h) ^ " * " ^(f t)
        in
            "(" ^ (f l) ^ ")"
        end

end
```

B.2.2.2 ConvertTypes

To integrate with another compiler, this routine must be customised.

```
structure ConvertTypes =
    struct

        open LITypes

        open SMLTy


        fun convertType t =
            (* Unfortunately, the signature SMLTY hides the datatype
             for MLj types.
             I tried removing the signature contraint from SMLTy, but
             it broke lots of other things. *)
            let
            in
                case fromTyVar t of
                    SOME v => TYVAR (TyVar.toString v)
                  | NONE =>
                case fromFunType t of
                    SOME (t, t') => FUN (convertType t, convertType t')
                  | NONE =>
                case fromConsType t of
                    SOME (l, c) => CON(TyName.toString c , map convertType l)
                  | NONE =>
                case fromRefType t of
                    SOME t => CON("ref", [convertType t])
                  | NONE =>
                case fromArrayType t of
                    SOME t => CON("array", [convertType t])
                  | NONE =>
                case fromProd t of
                    SOME l => TUPLE(map convertType l)
                  | NONE => CON(toString t, []) (* I think this must be a record *)
            end
```

```
    end
```

B.2.3 Representation of Morphisms and Rewriting (Isomorphisms)

Representation of morphisms and partial evaluation are in structure Isomorphisms. To integrate with another compiler, this will require minor changes (to deal with other abstract syntaxes). Currently the pretty-printer cannot deal with many MLj expressions.

```
structure Isomorphisms :
    sig

        (* Next a function to say whether a type constructor with
         a particular arity has a map function. *)
        val hasMap : string * int -> bool

        (* a morphism could be a morphism, or could be a morphism
         applied to some MLj expressions. *)

        type morphism

        val I : morphism
        val curry : morphism
        val uncurry : morphism
        val appUnit : morphism
        val mkLazy : morphism
        val mapFun : morphism * morphism -> morphism
        val mapCon : string * morphism list -> morphism
        val mapTuple : morphism list -> morphism
        val lambdaPat : Patterns.pattern * Patterns.pattern -> morphism
        val compose : morphism * morphism -> morphism

        (* This takes an ML expression and turns it into a
         morphism expression.
         This should only be used as a way to print ML expressions.
         to use them with morphisms, use applyToML. *)
        val toMExp : Syntax.Exp -> morphism

        (* This is the function for applying a morphism to and
         ML curried application and evaluating it.  It takes an
         ML expression list as input and returns a morphism list
         as a result (the morphism list contains the original
         syntax plus morphisms). *)
        val applyToML : morphism * Syntax.Exp list -> morphism

        val toString : morphism -> string
```

```
    end =
struct

    open Patterns

        datatype morphism =
            CURRY | UNCURRY
          | APP_UNIT | MK_LAZY
          | MAP_FUN of
            morphism * morphism
          | MAP_CON of string * morphism list
          | MAP_TUPLE of morphism list
          | LAMBDA_PAT of pattern * pattern
          | COMPOSE of morphism list
            (* "a o b" is [a, b], [] represents identity *)
          | CURRY_APP of morphism * morphism list
            (* [] represents identity *)
          | I
          | ML_EXP of Syntax.Exp
          | TUPLE_EXP of morphism list

    val maps =
        [(("list", 1), "List.map"),
         (("option", 1), "Option.map"),
         (("array", 1), "Array.map"),
         (("vector", 1), "Vector.map"),
         (("pair", 2), "mapPair")]

    fun getMap x = Option.map #2 (List.find (fn (x', _) => x=x') maps)

    fun hasMap x = List.exists (fn (x', _) => x=x') maps

    fun isI I = true
      | isI _ = false

    val curry = CURRY
    val uncurry = UNCURRY
    val appUnit = APP_UNIT
    val mkLazy = MK_LAZY

    fun mapFun (m, m') =
        if isI m andalso isI m' then
            I
        else
            MAP_FUN(m, m')

    fun mapCon (c, l) =
        if hasMap (c, length l) then
            if List.all isI l then
```

```
            I
        else
            MAP_CON(c, l)
    else
        raise Fail ("Internal error: attempt to create illegal map "^
                    c)

fun lambdaPat (p, p') =
    if p=p' then
        I
    else
        LAMBDA_PAT(p, p')

fun mapTuple l =
    if List.all isI l then
        I
    else if
        List.all (fn I => true | LAMBDA_PAT _ => true | _ => false) l then
        let
            val (args, results) =
                ListPair.unzip
                (map
                 (fn LAMBDA_PAT(a, r) => (a, r)
                   | I => let val i = newId() in (i, i) end
                   | _ => raise Fail "Internal Error mapTuple")
                 l)
        in
            lambdaPat(TUPLE args, TUPLE results)
        end
    else
        MAP_TUPLE l

fun compose1 (LAMBDA_PAT(p3, p4), (LAMBDA_PAT(p1, p2))::M) =
    let
        val S = unifyPat(p2, p3)
        val p1' = substPat S p1
        val p4' = substPat S p4
    in
        (lambdaPat(p1', p4'))::M
    end
  | compose1 (m, m') = m::m'

fun compose (I, m) = m
  | compose (m, I) = m
  | compose (M1, M2) =
    let
        val M1' = case M1 of COMPOSE l => l | m => [m]
        val M2' = case M2 of COMPOSE l => l | m => [m]
        val composed = List.foldl compose1 M2' (List.rev M1')
```

```
    in
       case composed of
           [] => I
         | [m] => m
         | l => COMPOSE l
    end


fun tupleStringByLen n =
    ToString.commaSep (List.tabulate (n, fn i => "a"^(Int.toString i)))

fun tupleStringByList l =
    ToString.commaSep (map (fn i => "a"^(Int.toString i)) l)

val symToString = JavaString.toMLString  o Symbol.toJavaString


fun toString CURRY = "curry"
  | toString UNCURRY = "uncurry"
  | toString APP_UNIT = "appUnit"
  | toString MK_LAZY = "mkLazy"
  | toString (MAP_FUN (i1, i2)) =
    ("mapFun(" ^
     (toString i1) ^ ", " ^
     (toString i2) ^ ")" )
  | toString (MAP_CON (c, ms)) =
    (case (getMap (c, length ms)) of
         NONE => "(raise Fail \"unknown map: " ^ c ^ "\")"
       | SOME s =>
             "(" ^ s ^
             "(" ^ (ToString.commaSep (map toString ms)) ^ ")" ^
             ")"
    )
  | toString (MAP_TUPLE l) =
    "mapTuple" ^ (Int.toString (length l)) ^
    " (" ^ (ToString.commaSep (map toString l)) ^ ")"
  | toString(LAMBDA_PAT (p, p')) =
    "(fn " ^ (patToString p) ^ " => " ^ (patToString p') ^ ")"
  |  toString I = "I"
  | toString (COMPOSE []) =
    "I"
  | toString (COMPOSE l) =
    "(" ^ (ToString.sep " o " (map toString l)) ^ ")"
  | toString (CURRY_APP (f, l)) =
    "(" ^ (ToString.spaceSep (map toString (f::l))) ^ ")"
  | toString (TUPLE_EXP l) =
    "(" ^ (ToString.commaSep (map toString l)) ^ ")"
  | toString (ML_EXP (_, e)) =
       (case e of
```

```
            Syntax.LongVid (Syntax.Short i) => symToString i
          | _ => "?" (* Don't know how to print this *) )


(* The next section involves actually evaluating morphism application. *)

(* Convert an MLj expression into a morphism expression. *)

fun toMExp (_, Syntax.Tuple l) = TUPLE_EXP (map toMExp l)
  | toMExp (_, Syntax.App (e0, e1)) =
    let
        val e0' = toMExp e0
        val e1' = toMExp e1
    in
        case e0' of
            CURRY_APP (f, args) => CURRY_APP(f, args@[e1'])
              | e0' => CURRY_APP(e0', [e1'])
    end
  | toMExp e = ML_EXP e


fun apply (I, (f::args)) = apply(f, args)

  (* CURRYING AND UNCURRYING *)

  | apply (CURRY, f::a1::a2::args) =
    apply(f, (TUPLE_EXP[a1, a2])::args)

  | apply (UNCURRY, f::(TUPLE_EXP[a1, a2])::args) =
    apply(f, a1::a2::args)

  (* INSERTING AND DELETING UNIT *)

  | apply (APP_UNIT, f::args) =
    apply(f, (TUPLE_EXP[])::args)

  | apply (MK_LAZY, f::(TUPLE_EXP[])::args) =
    apply(f, args)

  (* LAMBDA *)

  | apply (LAMBDA_PAT (p, p'), [a]) =
    let
        (* We try to match argument a to pattern p,
           then rewrite it using pattern p'.
           It may not be possible to do this, in
           which case just keep the lambda term here. *)
        fun match ((ID i, a), SOME dict)  = SOME ((i, a)::dict)
          | match ((TUPLE t, TUPLE_EXP a), SOME dict) =
```

```
        if length t<> length a then
            NONE
        else
            List.foldl match (SOME dict) (ListPair.zip (t, a))
      | match _ = NONE
    fun build dict (TUPLE l) = TUPLE_EXP(map (build dict) l)
      | build [] (ID i) = raise Fail "unbound variable"
      | build ((i, a)::dict) (ID i') =
        if i=i' then a else build dict (ID i')
in
    case match ((p, a), SOME []) of
        NONE => (CURRY_APP(LAMBDA_PAT (p, p'), [a]))
      (* or possibly return as a CURRY_APP *)
      | SOME d =>
            (build d p')
end

(* MAPS *)

| apply (MAP_FUN(m, m'), f::args) =
  apply(compose(m', (compose (f, m))), args)

| apply (MAP_TUPLE l, [TUPLE_EXP l']) =
  if length l = length l' then
      TUPLE_EXP(map apply (ListPair.zip (l, map (fn e => [e]) l')))
  else
      CURRY_APP(MAP_TUPLE l, [TUPLE_EXP l'])

(* COMPOSITION *)

(* Base cases for composition *)
| apply ((COMPOSE []), args) = apply(I, args)

| apply (COMPOSE [m], args) =
  apply(m, args)

(* Composition case *)
| apply ((COMPOSE (m::ms)), (e::es)) =
  let
      (* We rearrange an expression like
       (m o ms) e es
       into
       m (ms e) es *)

      val e' =
          (apply(COMPOSE ms, [e]))

      (* Next evaluate this *)
      val applyM = apply (m, e'::es)
```

```
        in
            applyM
        end

      (* This wee bit handling currying is required to get
         composition to work *)
      | apply (CURRY_APP(m, a), a') =
        apply (m, a@a')

      (* ERROR AND NO ACTION CASES *)

      | apply (m, []) =
        m

      | apply (m, a::args) =
        CURRY_APP(m, a::args)

    fun applyToML (morphism, mlExpList) =
        apply (morphism, map toMExp mlExpList)


end


(* The actual ML definitions of the isomorphism components. *)
structure Implementations =
    struct

        val I = fn i => i
        val curry = fn f => fn a1 => fn a2 => f(a1, a2)
        val uncurry = fn f => fn (a1, a2) => f a1 a2
        val appUnit = fn f => f ()
        val mkLazy = fn x => fn () => x
        val mapFun = fn (i1, i2) => fn f => i2 o f o i1
        fun mapTuple2 (f1, f2) (a1, a2) =
            (f1 a1, f2 a2)
        fun mapTuple3 (f1, f2, f3) (a1, a2, a3) =
            (f1 a1, f2 a2, f3 a3)
        fun mapTuple4 (f1, f2, f3, f4) (a1, a2, a3, a4) =
            (f1 a1, f2 a2, f3 a3, f4 a4)

        (* This is the only binary type constructor with a map. *)
        datatype ('a, 'b) pair = PAIR of 'a * 'b
        fun mapPair (f, g) (PAIR(a, b)) = (PAIR(f a, g b))

    end
```

B.2.4 Unification

The unification routines are too long for inclusion in full here. The files required are

- UnifyModuloLinearIso
- Rewrite
- UnifyAC
- GenerateMatrices
- UnifyACMatrices
- UnifyACUtilities
- UnifyEq
- Substitutions

These files should not require any changes to integrate them with another compiler.
Unification modulo linear isomorphism follows the algorithm exactly

```
structure UnifyModuloLinearIso :
    sig
        val unify :
            LITypes.types * LITypes.types ->
            (Substitutions.substitution * Isomorphisms.morphism) Stream.stream
    end =
struct

    open LITypes

    fun unify (t1, t2) =
        let
            val (t1', m1, m1') = Rewrite.rewrite t1
            val (t2', m2, m2') = Rewrite.rewrite t2

            val S = UnifyAC.unify (t1', t2')

            val S' =
                Stream.map
                (fn (s, m3) =>
                 (s, Isomorphisms.compose(m2', Isomorphisms.compose(m3, m1))))
                S
```

```
        in
            S'
        end

end
```

B.3 Using the Program

The program is compiled and invoked in the same way as the original version of MLj. It then presents the user with a prompt.

```
MLj 0.2c with Type Debugging by Bruce McAdam v0.1
Copyright (C) 1999 Persimmon IT Inc.

MLj comes with ABSOLUTELY NO WARRANTY. It is free software, and you are
welcome to redistribute it under certain conditions.
See COPYING for details.

This version incorporates Type Debugging Messages
Copyright (C) 2001 Bruce McAdam <bjm@dcs.ed.ac.uk>
The extensions come with ABSOLUTELY NO WARRANTY.  The extensions are
covered by a separate license agreement.
See COPYING_tydebug for details.
For further information see http://www.dcs.ed.ac.uk/home/bjm/
\
```

The easiest way to test it is to use the commands

`sourcepath` *dir*
`make` *file*

Where *dir*/*file*`.sml` is the file you wish to compile.

Bibliography

[App93] Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–430, 1993.

[AW93] A. Aiken and E.L. Wimmers. Type Inclusion Constraints and Type Inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM, June 1993.

[Bjø94] Nikolaj Skallerud Bjørner. Minimal typing derivations. In *Workshop on ML and its Applications*, pages 120–126. ACM SigPlan, 1994. Available as INRIA Technical Report No. 2265.

[BK99] Nick Benton and Andrew Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *International Conference on Functional Programming*, pages 126–147. ACM SIGPLAN, ACM Press, 1999.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming*, pages 129–140. ACM SIGPLAN, ACM Press, 1998.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, 1993.

[BS93] Mike Beaven and Ryan Stansifer. Explaining Type Errors in Polymorphic Languages. *ACM Letters on Programming Languages and Systems*, 2(1):17–30, March 1993.

[BS95] Karen L. Bernstein and Eugene W. Stark. Debugging Type Errors (Full Version). Technical report, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, U.S.A., November 1995. `http://www.cs.sunysb.edu/~stark/REPORTS/INDEX.html`.

[Car87] Luca Cardelli. Basic polymorphic type-checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[Car97] Luca Cardelli. *The Handbook of Computer Science and Engineering, Allen B. Tucker Jr. (Ed.)*, chapter 103 Type Systems, pages 2208–2236. CRC Press, 1997.

[Chi01] Olaf Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, pages 193–204. Association of Computing Machinery SIGPLAN, ACM Press, September 2001.

[Dam85] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K., April 1985. CST–33–85.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[Di 95] Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhäuser, 1995.

[DM82] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Ninth Annual Symposium on Principles of Programming Languages*, pages 207–212. Association of Computing Machinery, 1982.

[Dug98] Dominic Duggan. Correct type explanation. In *Workshop on ML*, pages 49–58. ACM SIGPLAN, 1998.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 1995.

[FF98] Matthias Felleisen and Daniel P Friedman. *The Little MLer*. MIT Press, 1998.

[Gil95] Stephen Gilmore. Designing for Proof. In *Mathematics of Software Quality*, 1995.

[Gil97] Stephen Gilmore. Programming in Standard ML '97: A Tutorial Introduction. Technical Report ECS–LFCS–97–364, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., 1997.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HHPW94] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. Technical Report FP–94–04, Department of Computing Science, University of Glasgow, 8-17 Lilybank Gardens, Glasgow, G12 8QQ, U.K., January 1994.

[Jim95] Trevor Jim. What are Principal Typings and What are They Good For? Technical Report MIT/LCS/TM–532, Massachusetts Institute of Technology, 77 Massachusetts Avenue Cambridge, MA 02139-4307, U.S.A., August 1995.

[JM93] L. Jategaonkar and J.C. Mitchell. Type Inference with Extended Pattern Matching and Subtypes. *Fund. Informaticae*, 19:127–166, 1993.

[JW86] Gregory F. Johnson and Janet A. Walz. A Maximum-Flow Approach to Anomaly Isolation in Unification-Based Incremental Type-Inference. In *13th ACM Symposium on Principles of Programming Languages*, pages 44–57. Association of Computing Machinery, ACM Press, 1986.

[Kah93] Stefan Kahrs. Mistakes and Ambiguities in the Definition of Standard ML. Technical Report ECS–LFCS–93–257,, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., April 1993.

[Kni89] Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[LC89] Partick Lincoln and Jim Christian. Adventures in Associative-Commutative Unification. *Journal of Symbolic Computation*, 8:217–240, 1989.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. In *20th Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 220–231. Association of Computing Machinery, ACM Press, 1993.

[Ler97] Xavier Leroy. *The Caml Light System, Documentation and User's Guide*. Institut National de Recherche en Informatique et Automatique, 0.74 edition, December 1997. Available from `http://caml.inria.fr/man-caml/index.html`.

[LRVD99] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml System, Documentation and User's Guide*. Institut National de Recherche en Informatique et Automatique, 2.02 edition, March 1999. Available from `http://pauillac.inria.fr/ocaml/htmlman/`.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a Folklore Let-Polymorphic Type Inference Algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[LY00] Oukseh Lee and Kwangkeun Yi. A Generalized Let-Polymorphic Type Inference Algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong Yusong-gu, Taejon 305-701, Korea, March 2000.

[McA97] Bruce J. McAdam. BigTypes in ML. In *International Conference on Functional Programming*, page 316. Association of Computing Machinery, June 1997. Poster abstract.

[McA98a] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, U.K.*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[McA98b] Bruce J. McAdam. On the Unification of Substitutions in Type-Inference. Technical Report ECS–LFCS–98–384, Laboratory for Foundations of Computer Science, The

University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., March 1998.

[McA99a] Bruce J. McAdam. A Data Structure for Representing Type Derivations. Technical Report ECS–LFCS–99–415, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., November 1999.

[McA99b] Bruce J. McAdam. Generalising Techniques for Type Explanation. In Greg Michaelson and Phil Trinder, editors, *Trends in Functional Programming 1999 (Proceedings of Scottish Functional Programming Workshop)*, pages 49–57. Intellect, 1999.

[McA01] Bruce McAdam. How to Repair Type Errors Automatically. In *Scottish Functional Programming Workshop, Stirling, U.K.*, pages 121–135, August 2001.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mit91] J.C. Mitchell. Type Inference with Simple Subtypes. *Journal of Functional Programming*, 1(3):245–286, 1991.

[Mit96] J.C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[MKB00] Bruce McAdam, Andrew Kennedy, and Nick Benton. Type Inference in MLj. In Stephen Gilmore, editor, *Trends in Functional Programming Volume 2 (Proceedings of Scottish Functional Programming Workshop)*, pages 159–171. Intellect, 2000.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. MIT Press, 1997.

[Nie01] Jakob Nielson. Error Message Guidelines, June 2001. ‘Alertbox’ column, available from `http://www.useit.com/`.

[NNH98] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1998.

[Nor98] Donald Norman. *The Invisible Computer*. MIT Press, 1998.

[NPS93] Paliath Narendran, Frank Pfenning, and Richard Statman. On the Unification Problem for Cartesian Closed Categories. In *8th Annual IEEE Symposium on Logic in Computer Science*, pages 57–63. IEEE, 1993.

[Pau96] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[PH98] Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/`, 1998.

[Pot96] F. Pottier. Simplifying Subtyping Constraints. In *International Conference on Functional Programming*, pages 122–133. ACM, 1996.

[PW93] Simon Peyton Jones and Phil Wadler. Imperative Functional Programming. In *20th ACM Symposium on Principles of Programming Languages*, pages 71–84. Association of Computing Machinery SIGPLAN, ACM Press, January 1993.

[Ram97] Norman Ramsey. Eliminating Spurious Messages. Technical Report CS–97–06, Department of Computer Science, University of Virginia, 151 Engineer’s Way, 204 Olsson Hall, Charlottesville, VA 22901, USA, 1997.

[Rit93a] Mikael Rittri. Finding the Source of Type Errors Interactively. In *Proc. El Wintermote*. Department of Computer Science, Chalmers University, 1993.

[Rit93b] Mikael Rittri. Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

[RT97] Laurence Rideau and Laurent Théry. Interactive Programming Environment for ML. Technical Report 3139, Inria, Domaine de Voluceau, Rocquencourt - B.P. 105, 78153 Le Chesnay Cedex, France, March 1997.

[Sha83] E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[Soo90] Helen Soosaipillai. An Explanation Based Polymorphic Type-Checker for Standard ML. Master's thesis, Department of Computer Science, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, U.K., 1990.

[Sti75] Mark E. Stickel. A Complete Unification Algorithm for Associative-Commutative Functions. In *4th Joint Conference on Artificial Intelligence*, pages 71–76, September 1975.

[Sti81] Mark E. Stickel. A unificiation algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, July 1981.

[Ten96] Edward Tenner. *Why Things Bite Back: Predicting the Problems of Progress*. 4th Estage, 1996.

[Tof89] Mads Tofte. Four Lectures on Standard ML. Technical Report ECS–LFCS–89–73, Laboratory for Foundations of Computer Science, The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, U.K., March 1989.

[Tur90] David A. Turner. Enhanced Error Handling for ML. CS4 Project, Department of Computer Science, The University of Edinburgh, May 1990.

[UIT96] *User Interfaces for Theorem Provers*, Eindhoven University of Technology, July 1996.

[Wan86] Mitchell Wand. Finding the Source of Type Errors. In *13th ACM Symposium on Principles of Programming Languages*, pages 38–43. Association of Computing Machinery SIGPLAN, ACM Press, 1986.

[WBL97] Jon Whittle, Alan Bundy, and Helen Lowe. An Editor for Helping Novices to Learn Standard ML. In *Proceedings of The First International Conference on Declarative Programming Languages in Education*, 1997.

[Whi96] Jon Whittle. A Qualitative and Quantitative Analysis of Errors Encountered by Novice ML Programmers. Informal Note 1132 from the Mathematical Reasoning Group, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, U.K., December 1996.

[Whi98] Jonathon N.D. Whittle. *Using Proofs as Programs to Build an Analogy Based Functional Program Editor*. PhD thesis, Division of Informatics, The University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, U.K., 1998.

[Yan97] Jun Yang. Visualization of the type system for functional programs. Technical Report Research Memorandum 97/13, Department of Computer Science and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, U.K., 1997.

[Yan99] Jun Yang. Explaining type errors by finding the source of a type conflict. In Greg Michaelson and Phil Trinder, editors, *Trends in Functional Programming 1999 (Proceedings of Scottish Functional Programming Workshop)*, pages 49–57. Intellect, 1999.

[Yan01] Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, U.K., July 2001.

[YM97] Jun Yang and Greg Michaelson. A visualisation of polymorphic type checking. Submitted to the Journal of Functional Programming, March 1997.

[YMT00] J. Yang, G. Michaelson, and P. Trinder. Helping students understand polymorphic type errors. In S. Alexander et al, editor, *Annual Conference of the LSTN Centre for Information and Computer Sciences*, pages 11–19. LTSN-ICS, August 2000.

[YMT01] Jun Yang, Greg Michaelson, and Phil Trinder. Human-like Explanations of Polymorphic Types. In Sharon Curtis, editor, *Scottish Workshop on Functional Programming*, pages 57–66, 2001.