Expressive Type Systems for Logic Programming Languages

David Jeffery

Submitted in total fulfilment of the requirements of
the degree of Doctor of Philosophy

February, 2002

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010
Australia

Abstract

The aim of this thesis is the design of a type system for an industrial strength logic programming language. The type system we describe has been implemented for the Mercury programming language, in the Melbourne Mercury compiler.

We begin by presenting a simple higher order extension of the Mycroft-O'Keefe [70] type system. We then use this type system as a basis for two significant extensions.

The first extension is the adaption of a type class system similar to that found in some modern functional languages in the context of higher order logic programming. We give a set of typing rules which both provide a formal definition of type correctness and define the source-to-source transformation we have used to implement type classes. This transformation is simple and effective, and can be easily shown to preserve Mercury's mode, determinism and uniqueness correctness properties.

The second extension is to allow existentially quantified type variables in the types of function symbols and of predicates. This represents the most significant contribution of this thesis. We then formally define the type system that results from the combination of both type classes and existential quantification of type variables. The two type system extensions are quite orthogonal. As a result, the definition of type correctness in the combined system is a fairly simple combination of the definitions for the individual systems. However, the mechanisms contributed by the two systems combine synergistically; the resulting type system is extremely expressive.

We then show how the type system we have designed allows the programmer to take advantage of many object oriented design techniques. We give an in depth analysis of object oriented design and isolate the mechanisms which are likely to result in reusable and maintainable software systems. We show that our type system allows the programmer to directly express these kinds of designs and discourages the use of the kinds of object oriented designs which reduce maintainability by introducing unnecessary implementation dependencies.

We show that these principles apply in a direct and simple manner to the modelling of component interfaces as supported by modern component frameworks such as CORBA [26]. We present MCORBA, an implementation of a CORBA binding for Mercury. This interface is bi-directional, allowing the programmer to write CORBA components in Mercury and to use existing CORBA components from within Mercury programs.

Acknowledgements

This research has been made possible by the financial support of the Commonwealth of Australia in the form of an Australian Postgraduate Award.

I would like to thank all the members of the Mercury group for providing such a pleasant and stimulating research environment, and for the enduring friendships we have formed over the years. My time with the group is something that I will never forget. Special thanks go to my supervisor Zoltan Somogyi, who knew when to let me follow my own (often ill-founded) course of research and when to drag me kicking and screaming back onto the path. I also owe a great debt to Fergus Henderson for his tireless hours spent helping me out with innumerable aspects of the implementation described in this thesis. Thanks also go to Tyson Dowd for *all* the good times (and there have been quite a few), to Mark Brown, without whom I would have drowned in a sea of mathematics (and sobriety) long ago, to Peter Ross, who has been on the other side of the world for quite some time but has never been that far away, and to Tom Conway, who taught me to think twice before eating an @.

I have also benefited greatly from the input of many other people at Melbourne over the years. I would like to thank Harald Søndergaard for introducing me properly to type theory (and also for knowing when to answer my questions and when to send me out of his office with an armful of books and papers), Anthony Senyard, who (aside from everything he has done for me as a friend) has proof read and given invaluable comments on much of this thesis, and Peter Stuckey, who has been an amazing sounding-board for ideas. I have also learnt a lot from many discussions with Martin Sulzmann, Peter Schachte and Kevin Glynn, who all provided helpful comments on earlier versions of this work.

Most of this thesis was written while I was working for the HAL group at Monash University. I am much indebted to my employers María García de la Banda, and Kim Marriott not only for their generosity with my time (which made writing this thesis possible), but more importantly for showing me how mee goreng was meant to be.

One day soon I will finally get to meet Ralph Becket from MSR Cambridge, who has provided detailed and extremely useful feedback on over half this thesis. It feels a little odd to owe a man so many pints without having ever met him in person.

Outside of work, I would like to thank my family for their patience and support over the last few years, and my many friends for all the fun we've had. Owen de Kretser and Matthew Saul have (in their own separate ways) provided much of that fun. I've got this feeling that's a trend that will continue for some time. My housemates Justin Drerup, Cameron Grant and Sian Scott deserve to be specially thanked for dealing with a housemate who would come and go at the strangest of hours over the last few months.

Without Alfred, Roy, Barry and all of their friends (you know who you are), I would never have stayed sane over the last four and a half years. (Big ups, Brand.)

Finally, I would like to thank Vanessa Toh. Without her support and encouragement (albeit from a continent away for most of the write-up) this thesis would never have been finished.

dgj
Melbourne, August 2001

Contents

List of Figures

Preface

This thesis comprises 7 chapters, the first and last of which are an introduction and a conclusion respectively. Chapter 2 is based partly upon [31] and the introductory material from [18] although the more formal material is new. The informal material from chapter 3 is based upon [42], and the more formal material is based upon the longer technical report version of the same paper [41], although the chapter represents a significant re-write of the published material. Chapter 4 is unpublished material. Chapter 5 is also unpublished material. Some of the motivational material from that chapter is based upon a lecture series presented by myself and Zoltan Somogyi during September 1998 at The University of Melbourne. Chapter 6 is based largely upon [40].

I certify that,

(i) other than as specified in the previous paragraph, this thesis comprises only my original work;

(ii) due acknowledgement has been made in the text to all other material used;

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, appendices and footnotes.

Chapter 1

Introduction

The discipline of software engineering emerged in the late 1960s as an attempt to combat what was then called the "software crisis" [84, 97]. This software crisis arose as a result of the arrival of computer hardware which was able to support application programs whose complexity had previously only been dreamt of. Existing software development procedures and technologies did not scale to the task of developing systems which were orders of magnitude larger and more complex than the systems to which they had previously been applied. The contribution of software engineering to software development practice has been the formation of sophisticated methodologies and processes for constructing software, as well as technologies which support those methodologies.

Although the nature of software has changed radically, three and a half decades later we are arguably still in the midst of a software crisis. While the start of the art in software development process and technology has advanced in leaps and bounds, the complexity of the software that we produce has increased at a similar, if not greater, rate. In the future, we can only expect this trend to continue; software systems will continue to become more complex, and as such there will be a continued need for systematic development processes and associated implementation technologies which help to manage these complexities.

The aspect of the software development process this thesis is concerned with is the design of a programming language for large scale software development. We can identify three important characteristics of programming languages which have been, at least to some extent, successful in combating implementation complexity:

1. They allow the programmer to concentrate on the important details of a problem and ignore other, less important details.

2. They allow a program to be decomposed into sub-programs in a way that promotes reuse, and in such a way that the complexity of the program remains manageable during maintenance.

3. They allow a program to interact with other software systems in a way that does not result in unnecessary complexity.

Logic programming allows the programmer to express programs at a very high level. Logic programming is an example of the declarative programming paradigm: a logic program (which can be viewed as a specification in predicate logic) states *what* the solution to a problem is, but leaves the details of *how* to calculate that solution to the implementation of the language. In addition, logic programming languages have much more useful formal semantics than their imperative counterparts. As a consequence, analysis of logic programs is much simpler, enabling compilers to perform aggressive optimisations. This allows the programmer to be less concerned with the performance of a program, and more concerned with its correctness and maintainability.

While logic programming languages have successfully been used in niche applications such as artificial intelligence, logic programming has thus far failed to make a significant impact on the wider software development market. One reason for this is that logic programming languages have not traditionally been designed with the needs of large scale software development in mind.

The logic programming language Mercury [31] is an exception to this rule. Mercury has been specifically designed with the needs of building large, robust software systems in mind. The most notable feature of Mercury is that it requires a program to pass several static analyses, including type, mode, determinism and uniqueness checking. This improves reliability, since many kinds of errors simply cannot happen in successfully compiled Mercury programs. This approach also improves productivity, since many errors that would otherwise require manual debugging to locate are found automatically by the compiler.

The price of enforcing these checks is a loss of flexibility, which makes prototyping and the construction of small programs more difficult as well as making some styles of programming impossible. This tradeoff is common in other engineering disciplines. Consider the case of an automobile with independent steering mechanisms for each of its wheels [14] — while this is certainly a more flexible design than that of an ordinary car, no sane automobile engineer would produce such an unsafe design in practice. Similarly, we believe that the loss of flexibility that comes as a consequence of rigorous static checking in a programming language is more than made up for by the improved reliability, and ultimately the improved programmer productivity, that it provides.

While logic programming languages successfully allow the programmer to express solutions at a very high level (characteristic 1), we believe that traditional logic programming languages have in many ways been less successful than imperative languages in addressing characteristics 2 and 3.

Object oriented architectures have proven very successful in the construction of maintainable, large scale, software systems [92]. However, despite many efforts to integrate object-oriented programming concepts into logic programming [12, 103, 111] no system has successfully done so and maintained either a simple semantics or the ability to fit into a strong typing framework. The ability to take advantage of many object oriented design techniques while maintaining strong typing and a useful formal semantics would be extremely useful for the logic programmer.

Component software frameworks have proven to be important mechanisms for communication amongst software systems and for software deployment generally. Building software by combining in-house and third party components improves software quality at the same time as supporting rapid development. Component software development is predicted by many to become the basis for most major software development in the near future [104]. Allowing logic programs to interoperate with component frameworks in a natural manner would greatly increase the usefulness of logic programming languages, and would be likely to allow a much wider group of programmers to take advantage of the benefits of logic programming. Development of software by assembling components is fundamentally different to traditional software development in several ways. Components come from third parties and as such, fault isolation and safety are of even greater importance. Component software allows the potential of *late integration*; rather than integrating and testing the individual modules of a program before deployment, code for software components are often not integrated until a program is actually run. As such, the ability to statically verify correctness of a component is likely to become increasingly desirable in a component market. Given the high level nature of logic programming languages, they are ideal for the task of building reliable components for late integration. As the importance of component software increases, we can expect the importance of a component-enabled logic programming language to increase.

In this thesis, we seek to answer the following question: is it possible to design a higher order logic programming language which allows the programmer to:

- take advantage of the object oriented design techniques which we can identify as likely to lead to robust and maintainable programs

- interoperate with component software frameworks (allowing both the construction of components and the use of existing components) in a manner that is natural and relatively simple

while maintaining a strong and static type system?

To answer this question with a resounding “yes”, we present a strong and static type system designed to support large scale software development for higher order logic programming languages which features two significant extensions over existing type systems for logic programming languages:

- A system of *type classes*. A type class specifies an interface consisting of one or more methods. A type can be declared to be a member of a type class by specifying the implementations of the methods for that type class, and functions can be declared to take as arguments values of any type that is a member of a given type class. This capability allows the programmer to take advantage of object oriented *interface inheritance* techniques.

- *Existential typing*, which allows the programmer to build heterogeneous collections of objects which all satisfy some type class interface. This is analogous to the ability of a programmer using a traditional object oriented language to build collections of objects which are all derived from the same abstract base class.

We formally define this type system, and show how this type system allows the programmer to take advantage of the object oriented design techniques which are likely to lead to robust and maintainable programs. We also show how this type system allows the programmer to interoperate with the CORBA [26] component framework by both implementing CORBA components and using existing CORBA components.

1.1 Overview of thesis

The remainder of this chapter presents the background material which places the contents of this thesis in context.

Chapter 2 describes the Mercury programming language, and defines an abstract syntax for this higher order logic programming language. We then use this abstract syntax to present a higher order version of the Mycroft-O'Keefe [70] type system, which is used as the basis for the type systems that we present in the subsequent chapters.

Chapter 3 presents our type class system. We provide an intuitive description of how the system works and provide a formal definition of type correctness. This is the first time that the type class system of functional languages has been adapted to logic programming.

Chapter 4 presents our existential type system. We provide an intuition behind how the system works and provide a formal definition of type correctness. We then show the type system that results from combining these two orthogonal type system extensions. The type system we present is both more expressive than any existing strong type system for a logic programming language (this being the first such type system to allow existential types) and more expressive in many ways than existential type systems of functional languages.

Chapter 5 examines the principles of good object oriented design, and shows how our type system encourages the programmer to write programs which obey these principles. Although type classes and existential types have in the past often been related to object oriented design, this is (to our knowledge) the first in-depth analysis on the subject which relates these concepts to a concrete object oriented design methodology.

Chapter 5 describes MCORBA, a bi-directional interface between Mercury and the CORBA component framework. The system makes heavy use of type classes and existential types. MCORBA was the first bi-directional interface between a strongly typed declarative language and a component framework, and remains the only such interface which takes advantage of the power of type classes and existential types to model software components.

Throughout the presentation, we have made an effort to ensure that the bulk of the material could be comprehended by the reader without a great deal of mathematical sophistication. Where possible, the presentation of the intuitions behind our type system have been separated from the more mathematically rigorous sections. Some readers may prefer to skip those sections. The overall flow of the presentation should not be affected.

1.2 Background

1.2.1 Logic programming

Logic is a formalisation of reasoning which has been studied in various forms since antiquity, providing us with a method for presenting methodical and rigorous arguments.

The field of logic programming began in the early 1970s as an offshoot of work on automatically proving theorems in the logic known as *predicate calculus*

[24] using a computer. Kowalski showed that predicate calculus can be used as a programming language, which was a radical idea at the time, given that predicate calculus had previously only ever been applied in computer science as a specification language [49].

Since that time, the theory of logic programming has become very well understood [55]. Many different logic programming languages have been implemented, one of the most influential of which has been Prolog ("programming in logic") [79, 101]. Prolog has been implemented by many different groups in many subtly different guises. However, the language was made the subject of an ISO Standard [39] in 1994 in an attempt to increase portability of programs amongst different implementations of the language.

Prolog has been successfully applied in many projects, particularly within the field of artificial intelligence, where it is particularly well suited to modelling many different kinds of knowledge representation. As a high level language with an inbuilt search mechanism, it greatly reduces the burden of writing many kinds of software.

The importance of introducing higher order programming to logic programming was recognised in the late 1980s with the introduction of HiLog [10], which has a higher order syntax but uses a first order semantics, and λProlog [71], which is a logic programming language based upon higher order logic [72].

While early logic programming languages were untyped, type systems have been designed into some modern logic programming languages such as λProlog (whose type system is described in [73]) and Gödel [35] (a logic programming language based on many sorted logic).

1.2.2 Type systems

A variable in a program can take a range of different values in a program. A *type* of a variable places bounds on these values. For example a variable with type `bool` might only be allowed to take the values `true` and `false`, while a variable of type `integer` could only take values like `1`, `2`, `-62431` and so forth. Some expressions make sense for values of certain types but not for others. For example, if `X` and `Y` are variables of type `bool`, then the expression `and(X,Y)` would be well-defined, but this might not be so if either of `X` or `Y` were of a type other than `bool`.

Languages in which variables are assigned a type, and which ensure that variables only take values of their assigned type, are called *typed languages*. Conversely, languages which do not place any restrictions upon the values that variables can take are called *untyped languages*. (Another useful way of thinking about untyped languages is they only support one "universal" type which all values are a member of.) In an untyped programming language, operations may be applied to values which are inappropriate. The effect of doing so depends upon the language in question, but is typically one of:

- a run-time fault, such as a segmentation fault or bus error
- an exception being thrown
- some fixed arbitrary value

Alternatively, the effect may simply be undefined.

A *type system* [7] is the part of a language responsible for assigning types to variables and expressions, and for using that information to accept or reject a program. The primary purpose of a type system is robustness — a type system prevents certain kinds of errors (which we will refer to as *type errors*) from occurring at run time. There are two other important purposes of a type system:

- Types aid modularity. By specifying the types of the arguments (and return value) of the operations, the programmer is able to provide information to other programmers about how the operations of a module are to be used. Types provide a *partial specification* for software structures.
- Types encourage more disciplined and regular software designs by ensuring that the structure of data is well defined.

Not all syntactically valid expressions are type correct. For example, while syntactically valid, an expression which multiplies a floating point number by a string would typically result in a type error. The same is true of an expression which tries to apply a number as a function to another number. It is the job of a type system to distinguish between such "bad" expressions whose meaning is ill-defined and the "good expressions" with well-defined meanings. From an operational point of view, the bad expressions can cause run-time type errors while the good expressions cannot.

However, all but the most trivial definition of badness will lead to the situation where deciding whether an expression is good or bad is an undecidable problem. We are faced with a choice: either we must

- reject some good expressions,
- accept some bad expressions,

- or accept the fact that we cannot expect our type checking algorithm to terminate in all cases.

Allowing bad expressions past our type checker is not an acceptable solution; we believe the software developer must be able to have confidence that programs will not cause run-time type errors. Similarly, a potentially non-terminating type-checker is not acceptable for a general purpose programming language.

We are left with the conclusion that, in order to remain safe, our system must reject some good expressions. A strong type system must make a conservative approximation of the run time behaviour of the program, accepting a program only if it is certain not to cause a type error. Figure 1.1 illustrates this situation:

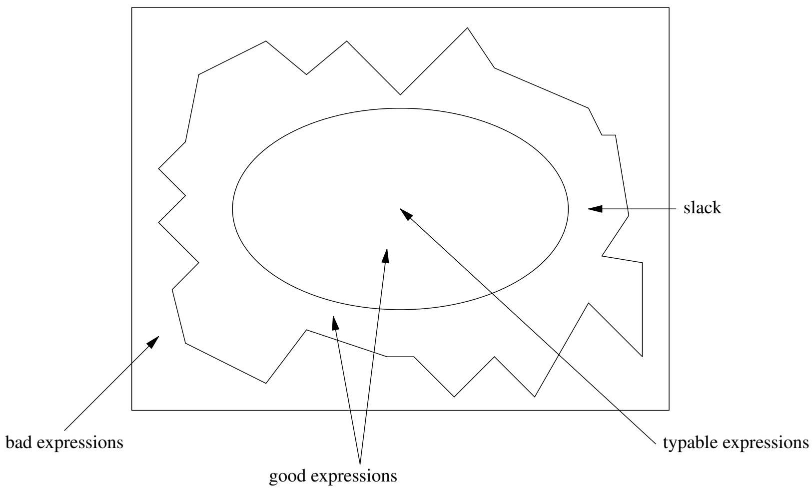

Figure 1.1: Good and bad expressions

The "slack" in the diagram represents those expressions that, while being good expressions, are not accepted by the type system. One of the primary aims of type system design is to minimise this slack, and in particular to include as few "useful" programs in this slack as possible.

Types capture, in a fairly simple manner, the range of values that a variable can take. We distinguish types from other kinds of annotations about variables and their values in a program. These annotations could range from informal comments to formal specifications of a program's behaviour (for use by a theorem prover). Types lie somewhere in between these two extremes. Unlike comments, types can be automatically checked. Unlike formal specifications, types do not allow the programmer to specify arbitrary properties of a program. As a consequence, we can design a type system so that the type checker is guaranteed to terminate. The same is not true of a theorem prover which attempts to prove the correctness of an arbitrary formal specification.

In addition, a type system differs from a theorem prover in that a type system should also be *transparent* to the programmer; the programmer should be able to predict whether or not a program is well-typed under the type system.

Specifying a type system

Type systems are complex and subtle. An informal definition of a type system is often not precise enough to unambiguously allow an implementation to be derived from it. There are many examples of different compilers for the same languages accepting different sets of programs because of an under-specified type system. Aside from providing an unambiguous specification, formalisation of a type system through the use of precise notations and definitions leads to the possibility of analysing those definitions to give us increased confidence in their appropriateness.

The standard method for specifying a type system is the definition of a "has-type" relation of the form:

$$\mathtt{t} : \tau$$

denoting that term t has type τ. The definition of this relation invariably needs to take into account the types of variables and other symbols. This information is captured in an *environment*. The exact form that an environment can take depends upon the type system in question. Environments are used in *typing judgements*, or simply *judgements*, of the form:

$$E \vdash \mathtt{t} : \tau$$

which states that, given the assumptions in the environment E, term t has type τ. For example, consider a simple type system where a typing environment is simply a mapping from identifiers to their type. In this system, the following judgement would hold:

$$\{\mathtt{X} \mapsto \mathtt{bool}, \mathtt{Y} \mapsto \mathtt{bool}, \mathtt{and} \mapsto ((\mathtt{bool} \times \mathtt{bool}) \rightarrow \mathtt{bool})\} \vdash \mathtt{and}(\mathtt{X}, \mathtt{Y}) : \mathtt{bool}$$

This judgement states that, assuming that x and y are of type bool and that and is a function which accepts two bools as arguments and returns a bool, then the expression $\mathtt{and}(\mathtt{X}, \mathtt{Y})$ is of type bool.

A type system is typically defined in terms of a set of *typing rules* which define which typing judgements are valid in that type system. Typing judgements are written in the form:

$$\text{APP}\,\frac{\begin{array}{c}E \vdash t : (\tau_1 \times \ldots \times \tau_n) \to \tau \\ E \vdash t_i : \tau_i \quad (i \in 1 \ldots n)\end{array}}{E \vdash t(t_1, \ldots, t_n) : \tau}$$

The judgement beneath the horizontal line is the *conclusion* of the rule. The judgements above the line are the *premises* of the rule. A typing rule states that if each of the premises holds, then the conclusion holds. The type rule given above is concerned with the typing of a function application in a simple type system. The typing rule concerns the application of a function, denoted in this case by t to its arguments, denoted t_1,...,t_n.

The study of type systems and their properties is typically based upon the theory of λ-*calculus* [2]. While λ-calculus does not itself provide the facilities expected of a programming language, it captures the aspects of functional programming essential to reasoning about a type system.

One of the most influential type systems for functional programming languages has been the Hindley-Milner [66] type system. This type system was defined for a language based upon a typed, first order λ-calculus, extended with a '`let`' construct, which allowed a simple form a *parametric polymorphism*, meaning that the type of a function may contain *type variables* which may be instantiated with different types each time a function is called. This has proven to be a powerful software reuse mechanism. The type system was originally implemented for the functional programming language ML [67], but has since been used in various forms by many programming languages, both functional and otherwise.

Many different forms of λ-calculi have been studied as models of different programming language features. Second order calculi model languages which permit function parameters which are themselves fully polymorphic functions (and the parameters of those fully polymorphic functions may themselves be fully polymorphic functions, ad infinitum). Various λ-calculi which model the type systems of object oriented programming languages have been studied in great detail [21, 85]. Various other language extensions such as type classes [27, 43, 47, 75, 82] and existential types [51] have also traditionally been formulated in a form of λ-calculus.

Given the connection between functional programming and λ-calculus, it is not surprising that modern functional programming languages typically have much richer type systems than other programming languages.

1.2.3 Types in logic programming

While functional languages have a typed mathematical system as their basis, logic programming languages are based on the untyped theory of predicate calculus. As a consequence, logic programming languages have traditionally been untyped. Nonetheless, it has been shown that types play an important role in the programmer's intended meaning of a logic program [74].

Early work on types in logic programming concentrated on *descriptive* type systems [11, 68, 106, 112]. A descriptive type system is one which does not seek to distinguish between well-typed and ill-typed programs; instead, it seeks to simply approximate the structure of a program for use by an optimising compiler.

The Mycroft-O'Keefe [70] prescriptive type system integrated the essence of the Hindley-Milner type system into a first order logic programming language. The presentation included a proof of soundness based upon the operational semantics of SLD resolution. Rather than presenting a semantic domain which included a notion of types, they showed that each step of SLD resolution (in the standard untyped semantic domain) preserved their notion of type correctness.

Several explicitly typed semantics for logic programming have been presented [30, 34, 50]. The semantics of Lakshman and Reddy [50] was also used to prove soundness of the Mycroft-O'Keefe type system.

Several type systems have been defined for higher order programming with parametric polymorphism [9, 10, 28, 35, 73]. Extending parametric polymorphism with subtyping in the context of logic programming was considered in [23, 65, 89, 94, 95, 110]. Soft type systems for logic programming have been investigated in [15].

Recent work on types for logic programming have concentrated on implementation techniques for efficiently inferring or checking polymorphic types by integrating polymorphic type checking into an abstract interpretation framework [57], or by applying constraint solving techniques [13].

Most recently, a novel proof of soundness for a Mycroft-O'Keefe style type system (for first order logic programming) was given in [16], using a declarative semantics based upon derivation trees.

The type system we have designed and implemented is significantly more expressive than any of these type systems for logic programming languages. Beginning with a higher order version of the Mycroft-O'Keefe type system as a basis, we first adapt the type class system of functional languages into the logic programming setting and then present an existential type system which is significantly different in style to any existing type system for a functional or logic programming language.

We have implemented this type system in the Melbourne Mercury compiler. The type class system has been available since version 0.8 and the existential type system has been available since version 0.9. While we have not done a formal analysis of the computational complexity of the type system, it has been used extensively in practice and has performed extremely well.

The Melbourne Mercury compiler and the MCORBA system are both available for download from the Mercury home page:
`http://www.cs.mu.oz.au/mercury/`

Chapter 2

Typed Higher Order Logic Programming and Mercury

In this chapter we introduce the programming language Mercury, an example of the higher order logic programming scheme and the testbed for the type systems we describe in the following chapters. We give an abstract syntax for Mercury which we call *core Mercury*, and present a simple type system for core Mercury that we extend in the following chapters with type classes and existential types.

2.1 Introduction to Mercury

Mercury is a purely declarative, strongly typed higher order logic programming language intended for general purpose large-scale programming. We will give a brief summary below of the features of Mercury relevant to this thesis, but for a more detailed definition we refer the reader to the Mercury language reference manual[31].

The syntax of Mercury is very similar to the syntax of Prolog. The main difference is that Mercury programs contain declarations that document several aspects of the intended behaviour of the predicates in the program. These aspects include the types of the arguments, the direction of the flow of data among predicate arguments, and the number of solutions a predicate can have. The declarations are required for predicates that are exported from the module that defines them. They are optional for predicates that are local to their modules; the compiler will perform inference to replace any missing declaration. However, in large programs and in team projects programmers should write down declarations even when they don't have to, because declarations serve as a very useful form of documentation. Declarations in Mercury can be trusted to be correct because the compiler is required to verify the information they contain; if it cannot do so, it must reject the program.

Predicate and function definitions. A predicate or function is defined by a set of clauses. Variables are denoted by tokens beginning with a capital letter. "Don't-care" variables begin with an underscore ('_'). Function symbols, predicate and function names begin with a lower case letter. Explicit integer, floating point, character and string constants as well as other builtin operators are also supported. Conjunction is represented by a comma (`,`), and explicit disjunction by a semicolon (`;`). Each clause implicitly defines a disjunct; the Melbourne Mercury compiler internally flattens a set of clauses into a single explicit disjunction. Mercury also supports if-then-else, represented by either "`A -> B ; C`" or "`if A then B else C`". Note that unlike Prolog, Mercury does not allow the "else" case to be omitted.

The following example code defines a `length` predicate which relates a list and the length of that list. This would normally be used to calculate the length of a list given as input.

```
length([], 0).
length([_|Xs], N + 1) :-
    length(Xs, N).
```

This is equivalent to the definition:

```
length(V1, V2) :-
    (
        V1 = [],
        V2 = 0
    ;
        V1 = [_|Xs],
        length(Xs, N),
        V2 = N + 1
    ).
```

As in Prolog, the expression "`[X|Xs]`" is a syntactic convenience for "`.(X, Xs)`", where "`.`" represents the list constructor symbol. For example, "`[1,2,3]`" is shorthand for "`.(1,.(2,.(3,[])))`.

Unlike in Prolog, the expression "`N + 1`" denotes a function application which is evaluated to an integer. (In Prolog, the expression would be left unevaluated as a compound term.)

Modules. Mercury modules are divided into two sections: an `interface` section, which contains declarations for any entities that are exported from the module, and an `implementation` section, which contains any declarations that are private to that module as well as the code for all predicates in the module.

Entities can be module qualified using "__". For example, the `write` predicate from the `io` module can be unambiguously referred to as `io__write`.

Types. The programmer is able to declare new types using the "`:- type`" declaration.

Discriminated union types are types defined in terms of a set of alternative function symbols that can be used to construct values of that type:

```
:- type bool ---> true ; false.
:- type list(T) ---> [] ; [T | list(T)].
```

Each of these declarations introduces a new type constructor and lists the one or more function symbols (also called functors) that can be used to construct terms of that type, and lists the types of the arguments of those function symbols.

Note that type variables are denoted by capital letters. Parameters of the type constructor being defined must be distinct type variables.

We will return to the subject of the form of type definitions in quite some detail in later chapters.

Equivalence types define a simple type renaming:

```
:- type count == int.
:- type intlist == list(int).
:- type partial_function(A, B) == map(A, B).
```

Mercury treats an equivalence type as an abbreviation for the type on the right hand side of the definition; the two are equivalent in all respects in scopes where the equivalence type is visible.

Abstract types are types whose name is exported from a module, but whose implementation is not.

```
:- type foo.
```

An abstract type declaration in a module's interface section must be matched by a non-abstract declaration in the implementation section of that module.

The programmer declares the types of predicate and function arguments with a "`:- pred`" or "`:- func`" declaration. Each such declaration lists the type of each of the arguments of the predicate or function and, in the case of functions, the type of the return value.

```
:- func increment(int) = int.
:- pred append(list(T), list(T), list(T)).
```

Modes. Mode declarations state which arguments of a predicate are input and which are output[1].

```
:- mode permute(in, out).
```

Typically, a predicate will have only one mode, but Mercury allows multiple mode declarations for a predicate.

```
:- mode append(in, in, out).
:- mode append(in, out, in).
:- mode append(out, out, in).
```

In the case where a predicate has a single mode, combined pred/mode declarations are also allowed:

```
:- pred permute(list(T)::in, list(T)::out).
```

A single mode of a predicate is called a *procedure*.

For each call to a predicate, the compiler chooses the appropriate mode to call the predicate in. The compiler may also re-order conjunctions to ensure that the value of a variable is bound before it is used. We say that the *producer* of a variable is placed before its *consumers*.

Uniqueness. Modes of a predicate can also specify uniqueness properties. The mode `di`, or *destructive input* denotes an input value that is unique (has no other references to it) on input, and whose value may be clobbered during the predicate call. The corresponding output mode is `uo`, denoting output variables that are guaranteed to be unique.

```
:- mode update_value(di, uo).
```

Uniqueness is used to give a declarative semantics to I/O and other operations which have side effects. The Mercury `io` library defines a type `io__state` which is used to represent the "state of the world". Each I/O operation takes an `io__state` as a destructive input and produces an `io__state` as a unique output.

Determinism. Mercury attaches information about the number of possible solutions to each mode of a predicate[32]. The most important determinisms are `det`, which denotes exactly one solution, `semidet` which denotes zero or one, `nondet` which denotes zero or more, and `multi` which denotes one or more.

[1] More generally, a mode declaration gives the initial and final instantiations of each argument.

```
:- mode append(in, in, out) is det.
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
```

Higher order programming. Mercury supports higher order programming. If `P` is a predicate-valued variable,, a call to `P` with arguments `1`, `"hello"`, and `42.0` is represented by `P(1, "hello", 42.0)`. That is, the syntax is the same as for a first order call, with a variable representing the predicate or function being called. For example, we can define the familiar `map` predicate, which applies a predicate to each element of an input list yielding an output list of the return values of the supplied predicate as follows:

```
:- pred map(pred(T, T), list(T), list(T)).
:- mode map(pred(in, out) is det, in, out) is det.
map(_P, [], []).
map(P, [X|Xs], [Y|Ys]) :-
    P(X, Y),
    map(P, Xs, Ys).
```

Note that the mode of a higher order argument lists the modes of its arguments and the determinism of that higher order goal.

Higher order values can be created using the `pred(...)` or `func(...)` syntax. For example, we can use the `map` predicate above to calculate the square of the integers from one to five with the following goal:

```
    P = (pred(N::in, Square::out) is det :- Square = N*N),
    map(P, [1,2,3,4,5], SquaredValues)
```

In the current Mercury implementation, mode and determinism declarations are compulsory on definitions of higher order terms.

2.2 Core Mercury: an abstract syntax for Mercury

We now present an abstract syntax which captures the structure of the part of Mercury that deals with predicate definitions. We will refer to this abstract syntax as "core Mercury". The abstract syntax is given in figure 2.1.

Throughout this thesis we use vector notation to represent a sequence of entities. For example, to represent a sequence of x, we write $\overline{x}$, which is equivalent to $\langle x_1, \ldots, x_n \rangle$. We use $\langle\rangle$ to represent the empty vector, and $|\overline{x}|$ to represent the number of elements in vector $\overline{x}$. Occasionally we will abuse this vector notation and treat a vector as a set. Whenever we do so, we assume that the elements of the vector are pairwise distinct.

We use the notation $\mathcal{P}X$ to denote a set whose elements are of type X.

$$
\begin{array}{llll}
\textsf{Variable} & v & & \\
\textsf{Function symbol} & f & & \\
\textsf{Predicate name} & \pi & & \\
\textsf{Term} & t & ::= & v \\
 & & | & f(\overline{t}) \\
 & & | & \lambda\overline{v}.G \\
\textsf{Goal} & G & ::= & \pi(\overline{t}) \\
 & & | & t(\overline{t}) \\
 & & | & t = t \\
 & & | & \texttt{true} \\
 & & | & \exists\overline{v}.G \\
 & & | & \neg\ G \\
 & & | & G \wedge G \\
 & & | & G \vee G \\
\textsf{Predicate} & C & ::= & \pi = \lambda\overline{v}.G \\
\textsf{Program} & P & ::= & \mathcal{P}(C)
\end{array}
$$

Figure 2.1: Core Mercury, an abstract syntax for Mercury

A program is represented as a set of predicates. Each predicate is a higher order expression where clauses are assumed to have been flattened into a single goal using explicit disjunctions. All variables are either introduced by a λ expression, or are explicitly existentially quantified.

In order to simplify the presentation, core Mercury contains no notion of functions — a function can be represented by a predicate with an extra argument corresponding to the function's return value. Using this process, a function of *arity* n (by which me mean that the function has n arguments) is converted into a predicate of arity $n + 1$.

The syntax of predicate bodies themselves is also simplified. For example, there is no explicit if-then-else construct; the Mercury goal `if A then B else C` is equivalent to the abstract syntax $(A \wedge B) \vee (\neg A \wedge C)$. Such issues have no impact on typing.

As an example of core Mercury, the following Mercury predicate definition:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

is equivalent to the abstract syntax:

$$
\begin{array}{ll}
\mathtt{append} = \lambda\mathtt{A},\mathtt{B},\mathtt{C}. & (\mathtt{A} = [] \wedge \mathtt{B} = \mathtt{C}) \vee \\
 & \exists\mathtt{X},\mathtt{Xs},\mathtt{Zs}.(\mathtt{A} = [\mathtt{X}|\mathtt{Xs}] \wedge \mathtt{C} = [\mathtt{X}|\mathtt{Zs}] \wedge \mathtt{append}(\mathtt{Xs},\mathtt{B},\mathtt{Zs}))
\end{array}
$$

(although an automatic translation from the Mercury definition to the abstract definition given above would likely introduce several redundant variables.)

We use identifiers such as `X`, `Y`, `Z`, `Zs`, `A`, `B` to denote program variables. We use identifiers such as v, v_1, v', v_{out} as meta-variables which range over program variables.

Similarly, we use identifiers such as `T`, `T1`, `T2` to denote type variables. We use identifiers such as α, β, α_1, α' as meta-variables which range over type variables.

Throughout this presentation, we will use Mercury goal syntax and this abstract syntax interchangeably.

2.3 A polymorphic type system

Core Mercury makes no distinction between well-typed and ill-typed terms and goals. There are many programs that are well-formed according to our abstract syntax, but are semantically meaningless because they are ill-typed.

The intention is that a type system must be defined along with the abstract syntax. This type system defines a set of well-typed programs, and excludes all others.

As an example, we now present an extension of the Mycroft-O'Keefe[70] type system. The Mycroft-O'Keefe type system was originally designed as a type system for first-order Prolog. We also include support for higher order programming. The resulting type system is very similar indeed to the Hindley-Milner type system[66] used by many functional languages.

The type language for this type system is given in figure 2.2.

Type variable	α		
Type constructor	χ		
Simple type	τ	$::=$	α
		$\mid$	$\chi(\overline{\tau})$
		$\mid$	$\mathbf{pred}(\overline{\tau})$
Predicate type	σ	$::=$	$\forall\overline{\alpha}.\mathbf{pred}(\overline{\tau})$
Function symbol type	φ	$::=$	$\forall\overline{\alpha}.\overline{\tau} \longrightarrow \chi\overline{\alpha}$

Figure 2.2: Type language for a Mycroft-O'Keefe like type system

We distinguish between simple types (which are either type variables, types built by applying a type constructor to a vector of types, or higher order types built by applying the distinguished type constructor **pred** to a vector of types) and polymorphic types (predicate types and function symbol types). Polymorphic types may be be *universally quantified* — they may be instantiated into a simple type by binding the universally quantified type variables to simple types themselves. For example, the predicate type $\forall\alpha.\mathbf{pred}(\alpha, \mathtt{list}(\alpha))$ could be instantiated into $\mathbf{pred}(\mathtt{int}, \mathtt{list}(\mathtt{int}))$ or $\mathbf{pred}(\mathtt{list}(\mathtt{int}), \mathtt{list}(\mathtt{list}(\mathtt{int})))$.

We formalise the notion of instantiating a type scheme by defining *substitution.* A substitution is a partial function from type variables to types which provides the *bindings* of the variables in its domain. We use the notation $[\tau/\alpha]$ to denote the substitution which maps α to τ. By extension, we use $[\overline{\tau}/\overline{\alpha}]$ to denote the substitution which maps each element of the vector $\overline{\alpha}$ onto the corresponding element of the vector $\overline{\tau}$. We denote the application of the substitution s to the type τ by $\tau\ s$. The effect of applying a substitution to a type is defined as follows:

$$\begin{aligned}
\alpha\ s &= \text{if } \tau/\alpha \in s \text{ then } \tau \text{ else } \alpha \\
\chi(\tau_1, \ldots, \tau_n)\ s &= \chi(\tau_1\ s, \ldots, \tau_n\ s) \\
\mathbf{pred}(\tau_1, \ldots, \tau_n)\ s &= \mathbf{pred}(\tau_1\ s, \ldots, \tau_n\ s)
\end{aligned}$$

(We occasionally extend this definition in the obvious manner to other syntactic entities which contain types.)

Throughout the presentation, we make use of the function **tvars**, which returns the set of type variables in a type:

$$\begin{aligned}
\mathbf{tvars}\ \alpha &= \{\alpha\} \\
\mathbf{tvars}\ \chi(\tau_1, \ldots, \tau_n) &= \textstyle\bigcup_{i=1}^{n} \mathbf{tvars}\ \tau_i \\
\mathbf{tvars}\ \mathbf{pred}(\tau_1, \ldots, \tau_n) &= \textstyle\bigcup_{i=1}^{n} \mathbf{tvars}\ \tau_i
\end{aligned}$$

Our typing rules are presented with respect to an "environment", which represents information collected during type checking. The components of the environment are as follows:

- P: the types of predicate symbols. P maps a predicate name and arity into a polymorphic type.
  $P :: \pi \times \mathcal{N} \longrightarrow \sigma$

- F: the types of function symbols. F maps a function symbol name and arity into a polymorphic type.
  $F :: f \times \mathcal{N} \longrightarrow \varphi$

- V: the types of variables in a clause.
  $V :: v \longrightarrow \tau$

(Note that throughout this thesis we use $\longrightarrow$ to denote both total and functions).

These environment components are stored in a single environment:

$$E = (P, F, V)$$

When defining an environment component explicitly, we use the notation:

$$X = \{a \mapsto x, b \mapsto y, c \mapsto z\}$$

to denote that the domain of X contains a, b, and c and the corresponding values they map to are x, y and z.

Definition: $(E.X)$ means selection of the X environment component from the environment E.

Definition: $X_1 \oplus X_2$ means the union of X_1 and X_2. It is an error if the domains of X_1 and X_2 are not disjoint.

We also write $E \oplus X_0$ to mean the environment E but with the elements of X_0 added to its X field, where X is P, V, F or other fields in later sections.

We use the symbols $\overset{\text{goal}}{\vdash}$, $\overset{\text{val}}{\vdash}$, $\overset{\text{pred}}{\vdash}$ and $\overset{\text{decl}}{\vdash}$ to distinguish between judgements over goals, values, predicates and declarations respectively:

- $E \overset{\text{goal}}{\vdash} G$ denotes that G is a well typed goal with respect to E.
- $E \overset{\text{val}}{\vdash} t : \tau$ denotes that t is a well typed term of type τ with respect to E.
- $E \overset{\text{pred}}{\vdash} \pi = \lambda\overline{v}.G$ denotes that $\pi = \lambda\overline{v}.G$ is a well typed predicate definition with respect to E.
- $E \overset{\text{decl}}{\vdash} X \triangleright E'$ denotes that X is a well formed declaration with respect to E, and yields the new environment E' which takes the declaration into account.

$$\text{MO-CALL}\ \frac{\begin{array}{c} E.P(\pi, n) = \forall\overline{\alpha}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \\ E \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}] \end{array}}{E \overset{\text{goal}}{\vdash} \pi(t_1, \ldots, t_n)}$$

Figure 2.3: Typing rule for calls in Mycroft-O'Keefe based type system

The typing rule for calls is found in figure 2.3.

The MO-CALL rule is concerned with an atomic goal (a predicate call). Checking the type correctness of a predicate call amounts to looking up the type of the called predicate and finding a suitable type substitution which maps the types of the formal arguments (from the callee) to those of the actual arguments (from of the call).

$$\text{MO-HO-CALL}\ \frac{E \overset{\text{val}}{\vdash} t_i : \tau_i \qquad E \overset{\text{val}}{\vdash} t : \mathbf{pred}(\tau_1, \ldots, \tau_n)}{E \overset{\text{goal}}{\vdash} t(t_1, \ldots, t_n)}$$

$$\text{MO-UNIFY}\ \frac{E \overset{\text{val}}{\vdash} t_1 : \tau \qquad E \overset{\text{val}}{\vdash} t_2 : \tau}{E \overset{\text{goal}}{\vdash} t_1 = t_2}$$

$$\text{MO-TRUE}\ \frac{}{E \overset{\text{goal}}{\vdash} \texttt{true}}$$

$$\text{MO-SOME}\ \frac{V = \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \qquad E \oplus V \overset{\text{goal}}{\vdash} G}{E \overset{\text{goal}}{\vdash} \exists v_1 \ldots v_n . G}$$

$$\text{MO-NOT}\ \frac{E \overset{\text{goal}}{\vdash} G}{E \overset{\text{goal}}{\vdash} \neg G}$$

$$\text{MO-CONJ}\ \frac{E \overset{\text{goal}}{\vdash} G_1 \qquad E \overset{\text{goal}}{\vdash} G_2}{E \overset{\text{goal}}{\vdash} G_1 \wedge G_2}$$

$$\text{MO-DISJ}\ \frac{E \overset{\text{goal}}{\vdash} G_1 \qquad E \overset{\text{goal}}{\vdash} G_2}{E \overset{\text{goal}}{\vdash} G_1 \vee G_2}$$

Figure 2.4: Typing rules for goals in Mycroft-O'Keefe based type system

The typing rules for goals other than calls are found in figure 2.4.

MO-HO-CALL checks a higher order call. This rule is quite simple, since higher order values are monomorphic, not polymorphic. Although the type of the higher order value may include a type variable, that type variable is not universally quantified in the higher order value's type. Instead, that type variable is universally quantified in the type of the enclosing predicate.

For example, given the higher order call $\mathtt{P}(\mathtt{X},\mathtt{Y})$ and the variable type assignment

$$V_0 = \{\mathtt{P} \mapsto \mathbf{pred}(\mathtt{T1},\mathtt{T2}), \mathtt{X} \mapsto \mathtt{T1}, \mathtt{Y} \mapsto \mathtt{T2}\}$$

the following is a valid judgement according to MO-HO-CALL (regardless of the environment E, as long as $E \oplus V_0$ is a valid environment):

$$E \oplus V_0 \overset{\text{goal}}{\vdash} \mathtt{P}(\mathtt{X},\mathtt{Y})$$

The type variables T1 and T2 represent types that are not known (or, to put it another way, types that are decided in a context outside the one in which this goal exists).

Note that there are many possible variable type assignments with which the above goal could be judged valid. For example,

$$V_1 = \{\mathtt{P} \mapsto \mathbf{pred}(\mathtt{int},\mathtt{float}), \mathtt{X} \mapsto \mathtt{int}, \mathtt{Y} \mapsto \mathtt{float}\}$$

and

$$V_2 = \{\mathtt{P} \mapsto \mathbf{pred}(\mathtt{T1},\mathtt{T1}), \mathtt{X} \mapsto \mathtt{T1}, \mathtt{Y} \mapsto \mathtt{T1}\}$$

would both lead to valid judgements. However, the one we have used in the judgement is the *most general* valid type assignment.

As with all such typing systems, one is required to somehow magically guess the appropriate initial environments in order to derive a valid judgement. In the implementation of a type checker, however, no such magic is required. The type inference/checking algorithm uses unification while traversing the program to derive the most general environment.

The remaining typing rules for goals MO-UNIFY, MO-TRUE, MO-SOME, MO-NOT, MO-DISJ and MO-CONJ handle unifications, `true`, existentially quantified goals, negated goals, disjunctions, and conjunctions respectively.

The typing rules for values are found in figure 2.5.

The MO-TERM rule judges a valid typing for a term. For example, consider

$$\text{MO-TERM}\ \frac{E.F(f,n) = \forall\overline{\alpha}.\langle\tau_1,\ldots,\tau_n\rangle \longrightarrow \tau \qquad E \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}]}{E \overset{\text{val}}{\vdash} f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha}]}$$

$$\text{MO-VAR}\ \frac{E.V(v) = \tau}{E \overset{\text{val}}{\vdash} v : \tau}$$

$$\text{MO-HO-CONST}\ \frac{V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \qquad E \oplus V \overset{\text{goal}}{\vdash} G}{E \overset{\text{val}}{\vdash} \lambda v_1 \ldots v_n.G : \mathbf{pred}(\tau_1,\ldots\tau_n)}$$

Figure 2.5: Typing rules for values in Mycroft-O'Keefe based type system

the typing of the term `.(1, [])`: Given the function symbol typing

$$F_0 = F_{\mathsf{builtin}} \cup \{ \quad ([],0) \mapsto \forall\mathtt{T}.\langle\rangle \longrightarrow \mathtt{list(T)}, \quad ('.',2) \mapsto \forall\mathtt{T}.\langle\mathtt{T},\mathtt{list(T)}\rangle \longrightarrow \mathtt{list(T)} \quad \}$$

(where F_{builtin} maps built-in constants to their type, such as $('\mathtt{1}',0) \mapsto \langle\rangle \longrightarrow \mathtt{int}$) and the environment $E_0 = \{\ \{\}, F_0, \{\}\ \}$, the following judgements are valid by successive applications of the MO-TERM rule:

$$E_0 \overset{\text{val}}{\vdash} [] : \mathtt{list(int)}$$
$$E_0 \overset{\text{val}}{\vdash} \mathtt{1} : \mathtt{int}$$
$$E_0 \overset{\text{val}}{\vdash} .(\mathtt{1},[]) : \mathtt{list(int)}$$

The MO-VAR rule simply allows a variable's type to be looked up in the environment. The MO-HO-CONST rule judges a valid typing for a higher order goal.

The rule for a predicate definition is found in figure 2.6.

Note the similarity between this rule and MO-HO-CONST; the essential difference between a predicate definition and the definition of a higher order constant is that the predicate definition may be polymorphic.

Consider the definition of the predicate `p`:

```
p(P, X, Y) :-
    P(X, Y).
```

$$\text{MO-PRED}\ \frac{\begin{array}{l} E.P(\pi, n) = \forall\overline{\alpha}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \\ V = \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \\ E \oplus V \overset{\text{goal}}{\vdash} G \end{array}}{E \overset{\text{pred}}{\vdash} \pi = \lambda v_1 \ldots v_n.G}$$

Figure 2.6: Typing rule for predicate definitions in Mycroft-O'Keefe based type system

which is equivalent to

$$\mathtt{p} = \lambda \mathtt{P}, \mathtt{X}, \mathtt{Y}.\mathtt{P}(\mathtt{X}, \mathtt{Y})$$

in our abstract syntax.

Now, with

$$\begin{array}{l} P_1 = \{\mathtt{p} \mapsto \forall \mathtt{T1}, \mathtt{T2}.\mathbf{pred}(\mathbf{pred}(\mathtt{T1}, \mathtt{T2}), \mathtt{T1}, \mathtt{T2})\} \\ E_1 = \{P, \{\}, \{\}\} \end{array}$$

and with

$$V_1 = \{\mathtt{P} \mapsto \mathbf{pred}(\mathtt{T1}, \mathtt{T2}), \mathtt{X} \mapsto \mathtt{T1}, \mathtt{Y} \mapsto \mathtt{T2}\}$$

the following are valid judgements:

$$\begin{array}{l} E_1 \oplus V_1 \overset{\text{goal}}{\vdash} \mathtt{P}(\mathtt{X}, \mathtt{Y}) \\ E_1 \overset{\text{pred}}{\vdash} \mathtt{p} = \lambda \mathtt{P}, \mathtt{X}, \mathtt{Y}.\mathtt{P}(\mathtt{X}, \mathtt{Y}) \end{array}$$

The rule allows the predicate's type to be polymorphic by universally quantifying over type variables that are used to type the predicate's body.

The user can introduce new type constructors using type declarations of the form:

$$\textsf{Type declaration} \qquad \forall\overline{\alpha}.\chi(\overline{\alpha}) \longrightarrow \langle f_1(\overline{\tau_1}); \ldots; f_n(\overline{\tau_n})\rangle$$

The meaning of these rules is rather different to that of our previous rules; rather than *using* the elements of the environment E, these rules *build* the environment. Each declaration rule has a translation which is the new environment which results from processing the declaration. We use the notation $E_0 \overset{\text{decl}}{\vdash} X \triangleright E$ to mean that declaration X is correct with respect to E_0, and results in environment E. When processing the whole program, we start with an empty environment ($E = (\{\}, \{\}, \{\})$, then process all the declarations, threading the environment through as it is built. Note that because declarations may be mutually recursive, the environment that results from processing a single declaration may not be valid. However, the environment which results from processing *all* of the declarations must be a valid one. It is this environment from which the predicates are judged.

The typing rule in figure 2.7 defines a well-formed type declaration.

$$\text{MO-TYPE-DECL}\frac{\begin{array}{l}\mathbf{tvars}(\overline{\tau_i}) \subseteq \overline{\alpha} \\ F = \{(f_i, |\overline{\tau_i}|) \mapsto \forall\overline{\alpha}.\overline{\tau_i} \longrightarrow \chi(\overline{\alpha})\}\end{array}}{E \overset{\text{decl}}{\vdash} \forall\overline{\alpha}.\chi(\overline{\alpha}) \longrightarrow \langle f_1(\overline{\tau_1}); \ldots; f_n(\overline{\tau_n})\rangle \triangleright E \oplus F}$$

Figure 2.7: Typing rule for type declarations in Mycroft-O'Keefe based type system

This well-formedness rule ensures that all type variables in a function symbol definition are universally quantified at the head of the type declaration and adds to the environment the types of all the function symbols of the type being defined.

The user can also declare the type of a predicate with a declaration whose abstract representation is given below:

Pred declaration	$\pi :: \sigma$

The rule in figure 2.8 defines a well-formed predicate declaration. The effect of a `pred` declaration is to ensure that the type of a predicate is as declared.

$$\text{MO-PRED-DECL}\frac{\begin{array}{l}\sigma = \forall\overline{\alpha}.\mathbf{pred}(\overline{\tau}) \\ \mathbf{tvars}(\overline{\tau}) = \overline{\alpha} \\ P = \{(\pi, |\overline{\tau}|) \mapsto \sigma\}\end{array}}{E \overset{\text{decl}}{\vdash} \pi :: \sigma \triangleright E \oplus P}$$

Figure 2.8: Typing rule for pred declarations in Mycroft-O'Keefe based type system

This type system is essentially the same as the type system of Mercury before type classes and existential types were introduced. In the following chapters, we present type classes and existential types as extensions of this type system.

Chapter 3

Type Classes

The type systems of modern functional languages such as Haskell [27], Gofer [46] and Concurrent Clean [76] have significant extensions to the Hindley-Milner type system. Their most important extension is their support for constrained genericity through *type classes*. A type class specifies an interface consisting of one or more methods. A type can be declared to be a member of a type class by specifying the implementations of the methods of that type class, and functions can be declared to take as argument values of any type that is a member of a given type class. This capability makes it significantly easier to express solutions to many common problems, and promotes code reuse.

Type classes give functional languages the kind of expressiveness that object-oriented constructs give to procedural languages. Just as a function in (say) C++ can operate on an object belonging to any sub-class of a given class, a function in Haskell can operate on any value whose type is a member of a given type class. Both the C++ and Haskell functions can invoke the operations associated with that class or type class. This means that the programmer does not have to write separate versions of the code of this function for each kind of object/value it needs to be able to operate on. Experience has shown that this kind of capability is extremely useful in the construction of large software systems.

The main difference between type classes in functional languages and classes in most object-oriented languages is that type classes provide inheritance only for interfaces, not for the implementations of methods. (We discuss this further in Chapter 5.) Even without inheritance of method implementations, the programmer can arrange for two implementations of a method to share code by having those implementations call the same functions, possibly in different contexts or with different arguments. This is because with type classes, the implementation of each method is explicitly under the control of the programmer.

This explicit control also allows and in fact requires the programmer to resolve the kinds of problems that arise in object-oriented systems with multiple inheritance when a class inherits conflicting implementations of a method from two or more of its super-classes.

We examine to the subject of object oriented programming using type classes in chapter 5.

Incorporating type classes in a logic programming language provides some new challenges. In this chapter, we explain how we have extended the type system defined in section 2.3 to include support for type classes, and show how this system fits in with the rest of the Mercury language. We show that type classes integrate very nicely with Mercury's mode, determinism and uniqueness systems, and describe how our implementation works.

As a purely declarative logic programming language designed to support the construction of large software systems, the fact that Mercury did not support constrained genericity or the use of object-oriented programming techniques was a significant drawback in this application area. We therefore decided to integrate type classes into Mercury. Type classes are cleaner and have simpler semantics than previous proposals for adding object-oriented constructs to logic languages [12, 111], and unlike many of those proposals, they do not introduce possibilities for any new kinds of run-time errors.

In section 3.1 we outline how type classes can be introduced into a typed higher order logic programming language. We discuss the issue of type correctness and describe the source to source transformation used by our system to implement type classes. In section 3.3, we consider the interactions between the type class system and the rest of Mercury, and show that our type class mechanism integrates in a simple and elegant manner with Mercury's mode, determinism and uniqueness systems.

We then present a set of formal rules that define type correctness with respect to our system of type classes. These rules rigorously describe the transformations that provide both an implementation for type classes and a semantics for programs using type classes in terms of programs that do not.

3.1 Type classes in a typed logic programming language

We now consider the problem of adding type classes to a higher order logic programming language and a Mycroft-O'Keefe style type system, as discussed in chapter 2.1. We will use the syntax of Mercury, but for now we will restrict our attention to issues involving types. We defer discussion of the effects of Mercury's mode, determinism and uniqueness systems upon type classes until section 3.3.

Syntactically, the type class system introduces three new constructs to the language, namely `typeclass` declarations, `instance` declarations, and type class constraints.

3.1.1 Type class declarations

A `typeclass` declaration introduces a new type class, and defines the interface to which a vector of one or more types must conform in order to be considered a member of that type class. The interface is a set of *methods*, i.e. predicate or function declarations. (We don't discuss functions further. Internally the Mercury compiler transforms functions into predicates, so they do not cause any additional difficulties.)

For example, the declaration:

```
:- typeclass x_y_point(T) where [
    pred get_x(T, float),
    pred get_y(T, float),
    pred set_x(T, float, T),
    pred set_y(T, float, T)
].
```

introduces the type class `x_y_point/1` (where the "`/1`" indicates that the arity of the type class is one), and says that a type `T` belonging to this type class must provide four predicates with the specified signatures to implement the methods `get_x/2`, `get_y/2`, `set_x/3` and `set_y/3`.

Most type class declarations have one type variable as their parameter. However, we also allow a type class declaration to have two or more type variables as parameters. Such *multi-parameter* type classes allow the program to clearly specify relationships among several types, via one or more operations in the type class whose signature includes more than one parameter of the type class. This capability increases the expressive power of the system, and is particularly useful in a language that supports constructor classes [44, 82]. (We do not address the issue of constructor classes in this thesis.)

3.1.2 Instance declarations

An `instance` declaration says that a particular vector of types is a member of a given type class, and specifies the predicates that implement each of the methods listed in the interface of that type class. For example, given the declarations

```
:- type cartesian ---> cart(float, float).

:- pred cart_get_x(cartesian, float).
:- pred cart_get_y(cartesian, float).
:- pred cart_set_x(cartesian, float, cartesian).
:- pred cart_set_y(cartesian, float, cartesian).
```

the `instance` declaration

```
:- instance x_y_point(cartesian) where [
    pred(get_x/2) is cart_get_x,
    pred(get_y/2) is cart_get_y,
    pred(set_x/3) is cart_set_x,
    pred(set_y/3) is cart_set_y
].
```

makes the `cartesian` type a member of the `x_y_point` type class.

An `instance` declaration gives a type for each parameter of the type class. These types must have the form of a type constructor applied to zero or more type variables, and each of these type variables may occur only once in the whole type vector. A program may not contain more than one declaration declaring a particular vector of type constructors to be an instance of a given type class. These restrictions ensure that there are no overlapping instance declarations, i.e. for a given type class there is at most one instance declaration that may be applied to any given vector of types.

3.1.3 Type class constraints

Polymorphic predicates have type variables in their signatures. In the absence of a type class constraint on the predicate declaration, the predicate may be called with actual values of any types which are instances of the expected (polymorphic) type. In the presence of one or more type class constraints on the predicate declaration, the predicate can only be called with values whose types satisfy all the constraints.

For example, in

```
:- pred draw_line(T1, T2, io__state, io__state)
                  <= x_y_point(T1), x_y_point(T2).

draw_line(P1, P2, IO0, IO) :-
    get_x(P1, X1),
    get_y(P1, Y1),
    move_pen(X1, Y1, IO0, IO1),
    pen_down(IO1, IO2),
    get_x(P2, X2),
    get_y(P2, Y2),
    move_pen(X2, Y2, IO3, IO4),
    pen_up(IO4, IO).
```

the type class constraint (also called class context)

```
    <= x_y_point(T1), x_y_point(T2)
```

says that the type variables `T1` and `T2` can only stand for types that are members of the `x_y_point` type class. This is why `draw_line` is allowed to invoke the methods of that type class on values such as `P1` and `P2` whose concrete types it does not know at compile time. Note that `P1` and `P2` may be of different types, but both types must be members of the `x_y_point` type class.

Each type class constraint is the name of a type class applied to one or more type variables. All these type variables must appear in the types of the arguments of the predicate, but a type variable may appear in more than one constraint and/or more than once in one constraint.

Type class constraints may also appear on `typeclass` and `instance` declarations. Constraints on type classes introduce the notion of "super-classes". Any vector of types declared to be an instance of a sub-class must also be declared to be an instance of the super-class. For example,

```
:- typeclass coloured_point(T) <= x_y_point(T) where [
    pred get_colour(T, colour)
].
```

introduces a new type class, `coloured_point`, describing points which have an associated colour, of which `x_y_point` is a super-class. The methods defined by the coloured_point type class include all those defined by the x_y_point type class, as well as get_colour. Any type declared to be an instance of coloured_point must also be declared to be an instance of x_y_point.

Constraints appearing on instance declarations form an *implied instance*. For example, consider the following declaration of a type class of types that may be printed:

```
:- typeclass portrayable(T) where [
    pred portray(T, io__state, io__state)
].
```

The programmer could declare instances such as

```
:- instance portrayable(int) where [
    pred(portray/3) is io__write_int
].
```

(io__write_int is defined in the Mercury standard library.) However, when it comes to writing the instance declaration for a type such as list(T), we want to be able to print out the list elements using the portray/3 for the particular type of the list elements. This can be achieved by placing a type class constraint on the instance declaration, as in the following example:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
    pred(portray/3) is portray_list
].
```

This states that for every type T that is a member of portrayable, the type list(T) is also a member.

Due to the class constraint on the instance declaration, portray_list is able to assume that the list elements are members of the portrayable type class, and can thus invoke its operations on them:

```
:- pred portray_list(list(T), io__state, io__state)
                    <= portrayable(T).

portray_list([], IO, IO).
portray_list([X | Xs], IO0, IO) :-
    portray(X, IO0, IO1),
```

```
    io__write_char(' ', IO1, IO2),
    portray_list(Xs, IO2, IO).
```

Note that Mercury also allows clauses to be written "inline" in an `instance` declaration. For example, instead of writing `portray_list` as a separate predicate, we could have written:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
    (portray([], IO, IO)),
    (portray([X | Xs], IO0, IO) :-
        portray(X, IO0, IO1),
        io__write_char(' ', IO1, IO2),
        portray(Xs, IO2, IO))
].
```

3.1.4 Type correctness

In this section we give an informal definition of type correctness in our system. We give a more formal definition in section 3.2.

There are two aspects to the type correctness of a program in our system: correctness of the clauses and correctness of the `instance` declarations. We will first consider the type correctness of the program clauses.

The program clauses are type correct if the program (ignoring type classes) is type correct under the higher order version of the Mycroft-O'Keefe type system (as we described in section 2.1) and if, in addition, the clauses' type class constraints are satisfied. Each clause's type class constraints are satisfied if the type class constraints on each predicate call in the clause are satisfied. For a type class constraint on a predicate call to be satisfied, one of the following conditions must hold:

1. The same constraint appears in the type signature of the caller (after the appropriate renaming of type variables).

2. The constraint is upon a vector of types where there is an instance declaration that declares a vector of types that is at least as general as this vector of types to be a member of the desired type class, and all type class constraints on this instance declaration are also satisfied (applying these rules recursively).

3. The constraint is for a class S, where another class C has been declared as a sub-class of S, and the corresponding constraint for the sub-class C

is satisfied (again applying these rules recursively).

There are two aspects to the type correctness of the `instance` declarations.

First, any type class constraints on the `typeclass` declaration that the `instance` declaration is trying to satisfy must also be satisfied. For example, given the type class declaration

```
:- typeclass c1(T) <= c2(T) where [
     ...
].
```

the instance declaration

```
:- instance c1(int) where [
     ...
].
```

can only be correct if there is also an instance declaration for `c2(int)`.

Second, the implementations of the type class methods provided by the `instance` declaration must also be type correct with respect to the method type declarations in the corresponding `typeclass` declaration; that is, the type of the instance method must be at least as general as the type declared for that method in the `typeclass` declaration (with the instance types substituted for the class type variables). Our original implementation checked this condition as a separate step in the type checker. The drawback of this approach was that it effectively required us to reproduce all the checks performed during type checking, including class constraint satisfaction.

We instead use a more elegant means: for each instance method, we introduce an auxiliary predicate whose type is the type of the corresponding class method with the instance types substituted for the class type variables, and whose type class constraints are the union of the class method's constraints and the constraints from the head of the instance declaration. Then we simply type-check these auxiliary predicates in the same manner that we do for ordinary predicates.

For example, consider the following declarations:

```
:- typeclass c1(T) where [
    pred m(T, T2) <= c2(T2)
].

:- instance c1(list(T)) <= c(T) where [
    pred(m/2) is p
].
```

In order to check the type correctness of p as an implementation of m, we introduce an auxiliary predicate, and let its type correctness be checked as usual:

```
:- pred method_for__c1___list__arity1__m_2(list(T), T2)
              <= (c2(T2), c(T)).

method_for__c1___list__arity1__m_2(X, Y) :-
    p(X, Y).
```

This reduces the problem of type-checking instance declarations to the problem of type-checking predicate declarations. It also allows the actual instance method (in this case `p`) to be less constrained or more polymorphic than the expected type of the method (the signature declared for the introduced predicate `method_for__c1___list__arity1__m_2` is this expected type).

3.1.5 Translation

For Haskell, the semantics of a program using type classes is defined by a simple source-to-source transformation to a program which does not use type classes [27, 108]. This transformation is also used in the implementations of the language. We take a similar approach here. In this section, we give an informal description of the translation leaving a more formal approach until section 3.2.

For each constraint in the type signature of a predicate, an extra argument is added to the head of that predicate. Corresponding to this, each call to a predicate must supply one extra argument for each constraint on the called predicate. Each of these extra arguments contains a dictionary that specifies, for each method in the interface of the relevant type class, the implementation of that method for the actual type(s) corresponding to the vector of type variables in the type class constraint. In addition, the dictionary contains a dictionary for each super-class constraint from the type class declaration. The implementation of a method is represented by a higher-order constant. For a type class such as `portrayable` with one method, class methods are implemented by extracting the relevant field from the dictionary and performing a higher order call on the extracted goal. For example, given the `typeclass` declaration

```
:- typeclass portrayable(T) where [
    pred portray(T, io__state, io__state)
].
```

from section 3.1.3, the implementation of the `portray` method is:

```
portray(PortrayDict, X, IO0, IO) :-
    PortrayDict = dictionary(Impl),
    Impl(X, IO0, IO).
```

Calls to the `portray` method actually call this predicate.

For each of the ways of satisfying a type class constraint listed in section 3.1.4, there is a corresponding method for obtaining the dictionary corresponding to that constraint.

1. When the constraint is satisfied because the same constraint appears in the type signature of the caller (after the appropriate renaming of type variables), the dictionary to be passed to the callee can come from one of the extra arguments added to the head of the caller.

2. When the constraint is satisfied by applying an instance declaration, the dictionary can be produced by a goal that corresponds to that instance declaration.

3. When the constraint is for a class S, where another class C has been declared as a sub-class of S, and the corresponding constraint for the sub-class C is satisfied, the dictionary for the type class S can be extracted from the dictionary for C.

The first simply involves passing the correct dictionary by keeping track of which dictionary variables correspond to which class constraints. To illustrate the second way, consider the two instance declarations of the type class portrayable above. The declaration asserting that integers are portrayable has no type class constraints on it, so the goal corresponding to this declaration is simply

```
IntDict = dictionary(write_int)
```

Instance declarations with constraints on them correspond to goals with one input argument for each constraint. The declaration asserting that a list of `T`s is portrayable has a type class constraint requiring `T` to be portrayable. Therefore its goal has one input argument giving the dictionary for `T`, from which it constructs the dictionary for `list(T)`

```
ListDict = dictionary(
            portray_list(TDict))
```

where `portray_list(TDict)` represents a closure with one partially applied argument. This would be used in situations such as the following, where we build a list whose elements have a type whose identity is unknown but is known to be portrayable. The dictionary for this type is in the variable `ElemDict`:

```
    ...
    Xs = [Elem1, Elem2, Elem3],
    ListDict = dictionary(portray_list(ElemDict)),
    portray(ListDict, Xs, IO0, IO),
    ...
```

To illustrate the third way of getting hold of dictionaries, consider the following predicate:

```
:- pred draw_colour_point(T, io__state, io__state)
                        <= coloured_point(T).

draw_colour_point(CP, IO0, IO) :-
    get_colour(CP, C),
    set_pen_colour(C, IO0, IO1),
        % Just draw a single point
    draw_line(CP, CP, IO1, IO).
```

The `<= x_y_point(T)` type class constraint from the call to `draw_line` is satisfied because its super class constraint `<= coloured_point(T)` is known to be satisfied.

The transformation of this predicate extracts the dictionary corresponding to the constraint `<= x_y_point(T)` from the dictionary for the constraint `<= coloured_point(T)`:

```
draw_colour_point(ColourPointDict, CP, IO0, IO) :-
    get_colour(ColourPointDict, CP, C),
    set_pen_colour(C, IO0, IO1),
    ColourPointDict = dictionary(XYDict, _GetColour),
    draw_line(XYDict, XYDict, CP, CP, IO1, IO).
```

(Note that the transformation of this call to `draw_line` involves passing two copies of the same dictionary to `draw_line`. This is because `draw_line` allows the points passed to it to have different types (as long as each type satisfies the `x_y_point` constraint), and thus requires a dictionary for each type. In this case, we are passing two values of the same type, so we pass the same dictionary twice.

3.2 Typing and translation rules

In this section we give a set of rules which both define the notion of type-correctness in our type system, and give a translation from our source language (a typed, higher order logic programming language) to our target language (an untyped, higher order logic programming language). In section 3.3, we consider the problem of adding type classes to the full Mercury programming language, and examine the challenges posed by Mercury's strong mode, determinism and uniqueness systems. The rules defined here are used by the Melbourne Mercury compiler for both type checking and type inference.

We begin by defining the type language of our source language, and defining a syntax for our source and target languages. We then give a set of rules which define type correctness of source programs in our system, and which simultaneously define a translation into our target language. This formalism is in the spirit of [70], but our rules resemble those for Haskell in [27] more closely. While we have attempted to follow the conventions of [27] wherever possible, our rules are necessarily more complicated as we handle multi-parameter rather than single-parameter type classes, and allow the types of type class methods to take (possibly constrained) polymorphic types rather than just monomorphic types.

Figure 3.1 shows the syntax of types, which are an extension of the higher order Mycroft-O'Keefe type system from figure 2.2 with type class constraints. Figure 3.2 shows the syntax of the declarations that we are adding to the language. Since our target language has no types or declarations, its syntax is confined to that shown previously in figure 2.1.

Type variable	α		
Type constructor	χ		
Class name	κ		
Simple type	τ	$::=$	α
		$\mid$	$\chi\overline{\tau}$
		$\mid$	$\mathbf{pred}(\overline{\tau})$
Context	θ	$::=$	$\langle \kappa_1\overline{\alpha}_1, \ldots, \kappa_n\overline{\alpha}_n \rangle$
Qualified type	σ	$::=$	$\forall\overline{\alpha}.\mathbf{pred}(\overline{\tau}) \Leftarrow \theta$
Function symbol type	φ	$::=$	$\forall\overline{\alpha}.\overline{\tau} \longrightarrow \chi\overline{\alpha}$

Figure 3.1: Syntax of types in the source language

The only addition to our type language is that we now associate class contexts with predicate types. We refer to a predicate type with a non-empty class context as a *qualified type*.

Declaration	::=	Pred declaration
	|	Type declaration
	|	Type class declaration
	|	Instance declaration
Pred declaration	::=	$\pi :: \sigma$
Type declaration	::=	$\forall\overline{\alpha}.\chi(\alpha) \longrightarrow \langle f_1(\overline{\tau_1}); \ldots ; f_n(\overline{\tau_n})\rangle$
Type class declaration	::=	`typeclass` $\kappa\overline{\alpha} \Leftarrow \theta$ `where` [(Pred declaration)$_1, \ldots,$ (Pred declaration)$_n$]
Instance declaration	::=	`instance` $\kappa\overline{\chi\overline{\alpha}} \Leftarrow \theta$ `where` [(Predicate)$_1, \ldots,$ (Predicate)$_n$]

Figure 3.2: Syntax of declarations in the source language

Throughout the presentation, we use the same conventions regarding variable and meta-variable naming as chapter 2, but add the prefix "`D`" to program variables which are introduced to hold dictionaries, and similarly add the prefix "d" to meta-variables which denote entities added as part of the transformation (including variables which are introduced to hold dictionaries and higher order goals which are introduced to produce dictionaries).

Our typing rules are again presented with respect to an environment. The components of this environment are as follows:

- P: the types of predicate symbols:
  $P :: \pi \times \mathcal{N} \longrightarrow \sigma$

- F: the types of function symbols:
  $F :: f \times \mathcal{N} \longrightarrow \varphi$

- V: the types of variables in a predicate:
  $V :: v \longrightarrow \tau$

- LI: local instance environment, which is a mapping from type class constraints to the local variables which represent their dictionaries:
  $LI :: \kappa\overline{\alpha} \longrightarrow v$

- I: instance environment, which is a mapping from type class constraints to the higher order goals which are used to generate them. This encodes both implied instances and super classes:
  $I :: \forall\overline{\alpha}.\kappa\overline{\tau} \Leftarrow \theta \longrightarrow \lambda\overline{v}.G$

- C: class environment, which is a mapping from type class name and arity to the super-classes and methods of the type class:

$C :: \kappa \times \mathcal{N} \longrightarrow \forall \overline{\alpha}.P \Leftarrow \theta$
(where P represents the methods and θ represents the super-classes of the class)

These environment components are stored in a single environment: $E = (P, F, LI, I, C, V)$.

Throughout the rest of this chapter, we will make use of the following example environment, which corresponds to some of the `typeclass` and `instance` declarations defined earlier in this chapter.

$$
\begin{array}{rl}
P_0 = & \{ \\
& \quad \begin{array}{ll}
(\mathtt{set\_x}, 3) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{float}, \mathtt{T}) \Leftarrow \mathtt{x\_y\_point}(\mathtt{T}), \\
(\mathtt{get\_x}, 2) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{float}) \Leftarrow \mathtt{x\_y\_point}(\mathtt{T}), \\
(\mathtt{set\_y}, 3) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{float}, \mathtt{T}) \Leftarrow \mathtt{x\_y\_point}(\mathtt{T}), \\
(\mathtt{get\_y}, 2) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{float}) \Leftarrow \mathtt{x\_y\_point}(\mathtt{T}), \\
(\mathtt{get\_colour}, 2) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{colour}) \Leftarrow \mathtt{coloured\_point}(\mathtt{T}), \\
(\mathtt{portray}, 3) & \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{io\_\_state}, \mathtt{io\_\_state}) \\
& \qquad\qquad \Leftarrow \mathtt{portrayable}(\mathtt{T})
\end{array} \\
& \} \\
F_0 = & \{ \\
& \quad \begin{array}{ll}
([], 0) & \mapsto \forall \mathtt{T}.\langle\rangle \longrightarrow \mathtt{list}(\mathtt{T}), \\
('.', 2) & \mapsto \forall \mathtt{T}.\langle \mathtt{T}, \mathtt{list}(\mathtt{T})\rangle \longrightarrow \mathtt{list}(\mathtt{T})
\end{array} \\
& \} \\
I_0 = & \{ \\
& \quad \begin{array}{ll}
\forall \mathtt{T}.\mathtt{portrayable}(\mathtt{list}(\mathtt{T})) \Leftarrow \mathtt{portrayable}(\mathtt{T}) & \mapsto dt_{p\_list} \\
\mathtt{portrayable}(\mathtt{int}) & \mapsto dt_{p\_int} \\
\forall \mathtt{T}.\mathtt{x\_y\_point}(\mathtt{T}) \Leftarrow \mathtt{coloured\_point}(\mathtt{T}) & \mapsto dt_{cp\_xyp}
\end{array} \\
& \}
\end{array}
$$

where dt_{p_list}, dt_{p_int} and dt_{cp_xyp}, which represent higher order goals for producing dictionaries from the corresponding `instance` or `typeclass` declarations, are defined as follows:

$$
\begin{array}{ll}
dt_{p\_list} & = \lambda \mathtt{DV1}, \mathtt{DVout}. \\
& \qquad (\mathtt{DVout} = \mathtt{dictionary}(\lambda \mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{portray\_list}(\mathtt{DV1}, \mathtt{A}, \mathtt{B}, \mathtt{C}))) \\
dt_{p\_int} & = \lambda \mathtt{DVout}.(\mathtt{DVout} = \mathtt{dictionary}(\lambda \mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{write\_int}(\mathtt{A}, \mathtt{B}, \mathtt{C}))) \\
dt_{cp\_xyp} & = \lambda \mathtt{DV1}, \mathtt{DVout}.(\mathtt{DV1} = \mathtt{dictionary}(\mathtt{DVout}, \_))
\end{array}
$$

$$
\begin{array}{lll}
C_0 = & \{ & \\
& (\texttt{x\_y\_point}, 1) & \mapsto \forall \texttt{T}. \\
& & \{ \\
& & \quad (\texttt{get\_x}, 2) \mapsto \mathbf{pred}(\texttt{T}, \texttt{float}), \\
& & \quad (\texttt{get\_y}, 2) \mapsto \mathbf{pred}(\texttt{T}, \texttt{float}), \\
& & \quad (\texttt{set\_x}, 3) \mapsto \mathbf{pred}(\texttt{T}, \texttt{float}, \texttt{T}), \\
& & \quad (\texttt{set\_y}, 3) \mapsto \mathbf{pred}(\texttt{T}, \texttt{float}, \texttt{T}), \\
& & \}, \\
& (\texttt{coloured\_point}, 1) & \mapsto \forall \texttt{T}. \\
& & \{ \\
& & \quad (\texttt{get\_colour}, 2) \mapsto \mathbf{pred}(\texttt{T}, \texttt{colour}) \\
& & \} \\
& & \Leftarrow \texttt{x\_y\_point}(\texttt{T}), \\
& (\texttt{portrayable}, 1) & \mapsto \forall \texttt{T}. \\
& & \{ \\
& & \quad (\texttt{portray}, 3) \mapsto \\
& & \qquad\qquad \mathbf{pred}(\texttt{T}, \texttt{io\_\_state}, \texttt{io\_\_state}) \\
& & \} \\
& \} & \\
E_0 = & (P_0, F_0 \cup F_{\mathsf{builtin}}, \{\}, I_0, C_0, \{\}) &
\end{array}
$$

We now present the typing/transformation rules. In these rules, we use the notation $E \vdash X \rightsquigarrow X'$ to indicate that X is well typed with respect to E, and it is transformed into X'.

As in chapter 2, we use the symbols $\overset{\text{goal}}{\vdash}$, $\overset{\text{val}}{\vdash}$, $\overset{\text{pred}}{\vdash}$ and $\overset{\text{decl}}{\vdash}$ to distinguish between judgements over goals, values, predicates and declarations respectively:

- $E \overset{\text{goal}}{\vdash} G \rightsquigarrow G'$ denotes that G is a well typed goal with respect to E, and is transformed into G'.
- $E \overset{\text{val}}{\vdash} t : \tau \rightsquigarrow t'$ denotes that t is a well typed term of type τ with respect to E, and is transformed into t'.
- $E \overset{\text{pred}}{\vdash} \pi = \lambda\overline{v}.G \rightsquigarrow \pi' = \lambda\overline{v'}.G'$ denotes that $\pi = \lambda\overline{v}.G$ is a well typed predicate definition with respect to E, and is transformed into $\pi' = \lambda\overline{v'}.G'$.
- $E \overset{\text{decl}}{\vdash} X \triangleright E' \rightsquigarrow P$ denotes that X is a well formed declaration with respect to E, and yields the new environment E' which takes the declaration into

account, and P is a set of predicates which appear in the transformed program.

We introduce one new form of judgement: we use $E \overset{\text{dict}}{\vdash} \theta \rightsquigarrow dt$ to denote that the type class constraint θ is satisfied under E and is transformed into the higher order term dt which produces a dictionary corresponding to that type class constraint.

3.2.1 Goals, values and predicate definitions

$$\text{TC-CALL}\ \frac{\begin{array}{l} E.P(\pi, n) = \forall\overline{\alpha}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \Leftarrow \langle \kappa_1\overline{\alpha_1}, \ldots \kappa_k\overline{\alpha_k}\rangle \\ E \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}] \rightsquigarrow t'_i \\ E.LI(\kappa_i\overline{\alpha_i}[\overline{\tau'}/\overline{\alpha}]) = dv_i \end{array}}{E \overset{\text{goal}}{\vdash} \pi(t_1, \ldots, t_n) \rightsquigarrow \pi_n(dv_1, \ldots, dv_k, t'_1, \ldots, t'_n))}$$

Figure 3.3: Type class typing/transformation rules for calls

The typing and transformation rule for an atomic goal (a predicate call) is given in figure 3.3.

Checking the type correctness of a predicate call amounts to:

- looking up the type of the called predicate and finding a suitable type substitution which maps the types of the callee's arguments to those of the caller;
- checking that the constraints on the type of callee, (with the substitution applied so they are expressed on the types of the arguments of the call), are satisfied. We check satisfaction of the type class constraints by looking them up in the LI component of the environment. LI maps type class constraints which are known to be satisfied to variables containing their dictionaries.

The transformation for a predicate call is:

- Transform the terms being passed in the call.
- Add the variables which contain the type class dictionaries for the type class constraints to the front of the atom being called.

For example, given

$$\begin{aligned} LI = &\quad \{\mathtt{x\_y\_point(T)} \mapsto \mathtt{DV}\} \\ V = &\quad \{\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{T}\} \end{aligned}$$

and with E_0 defined as above, the following is a valid judgement:

$$E_0 \oplus LI \oplus V \overset{\text{goal}}{\vdash} \mathtt{set\_x(X, 3.0, Y)} \leadsto \mathtt{set\_x(DV, X, 3.0, Y)}$$

Typing and transformation rules for goals other than calls are given in figure 3.4.

TC-HO-CALL checks a higher order call. This rule, like MO-HO-CALL is quite simple, since higher-order types do not have any type class constraints. Higher order terms are still monomorphic, so there are no universally quantified type variables to constrain. However, type variables in the higher order type may be constrained in the enclosing scope where the type variable is universally quantified.

The rules TC-UNIFY, TC-TRUE, TC-SOME, TC-NOT, TC-DISJ and TC-CONJ are analogous to their Mycroft-O'Keefe counterparts.

The rules concerned with values, TC-TERM, TC-VAR and TC-HO-CONST can be found in figure 3.5. These check the types of terms, variables and higher order constants. Only TC-TERM is essentially the same as the rule for our Mycroft-O'Keefe style type system, although the type class transformation is applied to the arguments of the term in question. When checking lambda goal using TC-HO-CONST, we augment the environment with one variable of the appropriate type for each argument of the lambda expression, since these variables are visible from inside the lambda goal but not from the outside. The transformation of the lambda goal may introduce new variables representing type class dictionaries. Some of these may be nonlocal to the lambda goal, i.e. they may occur outside the lambda goal as well. This is fine, since this may happen even without the transformation; all these nonlocal variables will be included in the closure generated for the lambda expression.

For example, given

$$\begin{aligned} LI &= \{\mathtt{x\_y\_point(T)} \mapsto \mathtt{DV}\} \\ V &= \{\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{T}\} \end{aligned}$$

and with E_0 defined as above, the typing of the lambda expression

$$\lambda\mathtt{X}, \mathtt{Y}.\mathtt{set\_x(X, 3.0, Y)}$$

$$\text{TC-HO-CALL}\ \frac{E \overset{\text{val}}{\vdash} t_i : \tau_i \rightsquigarrow t'_i \qquad E \overset{\text{val}}{\vdash} t : \mathbf{pred}(\tau_1, \ldots, \tau_n) \rightsquigarrow t'}{E \overset{\text{goal}}{\vdash} t(t_1, \ldots, t_n) \rightsquigarrow t'(t'_1, \ldots, t'_n)}$$

$$\text{TC-UNIFY}\ \frac{E \overset{\text{val}}{\vdash} t_1 : \tau \rightsquigarrow t'_1 \qquad E \overset{\text{val}}{\vdash} t_2 : \tau \rightsquigarrow t'_2}{E \overset{\text{goal}}{\vdash} t_1 = t_2 \rightsquigarrow t'_1 = t'_2}$$

$$\text{TC-TRUE}\ \frac{}{E \overset{\text{goal}}{\vdash} \texttt{true} \rightsquigarrow \texttt{true}}$$

$$\text{TC-SOME}\ \frac{V = \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \qquad E \oplus V \overset{\text{goal}}{\vdash} G \rightsquigarrow G'}{E \overset{\text{goal}}{\vdash} \exists v_1 \ldots v_n . G \rightsquigarrow \exists v_1 \ldots v_n . G'}$$

$$\text{TC-NOT}\ \frac{E \overset{\text{goal}}{\vdash} G \rightsquigarrow G'}{E \overset{\text{goal}}{\vdash} \neg G \rightsquigarrow \neg G'}$$

$$\text{TC-CONJ}\ \frac{E \overset{\text{goal}}{\vdash} G_1 \rightsquigarrow G'_1 \qquad E \overset{\text{goal}}{\vdash} G_2 \rightsquigarrow G'_2}{E \overset{\text{goal}}{\vdash} G_1 \wedge G_2 \rightsquigarrow G'_1 \wedge G'_2}$$

$$\text{TC-DISJ}\ \frac{E \overset{\text{goal}}{\vdash} G_1 \rightsquigarrow G'_1 \qquad E \overset{\text{goal}}{\vdash} G_2 \rightsquigarrow G'_2}{E \overset{\text{goal}}{\vdash} G_1 \vee G_2 \rightsquigarrow G'_1 \vee G'_2}$$

Figure 3.4: Type class typing/transformation rules for goals

involves the following judgements:

$$E_0 \oplus LI \oplus V \overset{\text{goal}}{\vdash} \texttt{set\_x}(\texttt{X}, 3.0, \texttt{Y}) \rightsquigarrow \texttt{set\_x}(\texttt{DV}, \texttt{X}, 3.0, \texttt{Y})$$

$$E_0 \oplus LI \overset{\text{val}}{\vdash} \lambda\texttt{X}, \texttt{Y}.\texttt{set\_x}(\texttt{X}, 3.0, \texttt{Y}) : \mathbf{pred}(\texttt{T}, \texttt{float}, \texttt{T}) \rightsquigarrow \lambda\texttt{X}, \texttt{Y}.\texttt{set\_x}(\texttt{DV}, \texttt{X}, 3.0, \texttt{Y})$$

$$\text{TC-TERM}\;\frac{\begin{array}{l} E.F(f,n) = \forall\overline{\alpha}.\langle\tau_1,\ldots,\tau_n\rangle \longrightarrow \tau \\ E \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}] \rightsquigarrow t'_i \end{array}}{E \overset{\text{val}}{\vdash} f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha}] \rightsquigarrow f(t'_1,\ldots,t'_n)}$$

$$\text{TC-VAR}\;\frac{E.V(v) = \tau}{E \overset{\text{val}}{\vdash} v : \tau \rightsquigarrow v}$$

$$\text{TC-HO-CONST}\;\frac{\begin{array}{l} V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \\ E \oplus V \overset{\text{goal}}{\vdash} G \rightsquigarrow G' \end{array}}{E \overset{\text{val}}{\vdash} \lambda v_1 \ldots v_n.G : \mathbf{pred}(\tau_1,\ldots\tau_n) \rightsquigarrow \lambda v_1 \ldots v_n.G'}$$

Figure 3.5: Type class typing/transformation rules for values

$$\text{TC-PRED}\;\frac{\begin{array}{l} E.P(\pi,n) = \forall\overline{\alpha}.\mathbf{pred}(\tau_i,\ldots,\tau_n) \Leftarrow \langle\kappa_1\overline{\alpha_1},\ldots,\kappa_k\overline{\alpha_k}\rangle \\ LI = \{\kappa_1\overline{\alpha_1} \mapsto dv_1,\ldots,\kappa_k\overline{\alpha_k} \mapsto dv_k\} \\ V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \\ E \oplus LI \oplus V \overset{\text{goal}}{\vdash} G \rightsquigarrow G' \end{array}}{E \overset{\text{pred}}{\vdash} \pi = \lambda v_1,\ldots,v_n.G \rightsquigarrow \pi_n = \lambda dv_1,\ldots,dv_k,v_1,\ldots,v_n.G'}$$

Figure 3.6: Type class typing/transformation rule for predicate definitions

The rule TC-PRED (in figure 3.6) is concerned with the typing of a predicate definition. Intuitively, the rule says that the body of the predicate is checked (and transformed) under the assumption that the head variables of the predicate have their expected types, and that the dictionaries for each of the predicate's type class constraints can be found in the variables introduced in LI.

The TC-PRED closely resembles the TC-HO-CONST rule, the difference being that a predicate definition may be assigned a qualified type rather than a simple type. As such, predicate bodies are transformed to take an extra variable to hold the variable for each class constraint on the predicate's signature.

During the transformation the predicate name is annotated with its original arity so that in adding extra arguments we do not make two predicates which were originally of the same name but different arity into predicates of the same name and same arity.

Consider the definition of the predicate $\mathtt{p}$:

$$\mathtt{p} = \lambda \mathtt{A}, \mathtt{B}.\mathtt{set\_x}(\mathtt{A}, 3.0, \mathtt{B})$$

Now, given

$$\begin{aligned} P = &\quad \{\mathtt{p} \mapsto \forall \mathtt{T}.\mathbf{pred}(\mathtt{T}, \mathtt{T})\} \\ LI = &\quad \{\mathtt{x\_y\_point}(\mathtt{T}) \mapsto \mathtt{DV}\} \\ V = &\quad \{\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{T}\} \end{aligned}$$

and with E_0 as previously defined, the typing of $\mathtt{p}$ involves the following judgements (with the resulting transformations shown):

$$\begin{aligned} E_0 \oplus P \oplus LI \oplus V \overset{\text{goal}}{\vdash} &\quad \mathtt{set\_x}(\mathtt{X}, 3.0, \mathtt{Y}) \\ &\quad \rightsquigarrow \mathtt{set\_x}(\mathtt{DV}, \mathtt{X}, 3.0, \mathtt{Y}) \\ E_0 \oplus P \overset{\text{pred}}{\vdash} &\quad \mathtt{p} = \lambda \mathtt{X}, \mathtt{Y}.\mathtt{set\_x}(\mathtt{X}, 3.0, \mathtt{Y}) \\ &\quad \rightsquigarrow \mathtt{p}_2 = \lambda \mathtt{DV}, \mathtt{X}, \mathtt{Y}.\mathtt{set\_x}(\mathtt{DV}, \mathtt{X}, 3.0, \mathtt{Y}) \end{aligned}$$

3.2.2 Dictionaries

$$\text{TC-LI-ADD}\;\frac{\begin{array}{c} LI = \{\kappa\overline{\tau} \mapsto dv\} \\ E \overset{\text{dict}}{\vdash} \kappa\overline{\tau} \rightsquigarrow dt \\ E \oplus LI \overset{\text{goal}}{\vdash} G \rightsquigarrow G' \end{array}}{E \overset{\text{goal}}{\vdash} G \rightsquigarrow \exists dv.(dt(dv) \wedge G')}$$

Figure 3.7: Type class typing/transformation rule for dictionary introduction

Figure 3.7 contains the rule TC-LI-ADD, which uses the fact that a particular type class constraint is satisfiable to check the type correctness of a goal. A constraint that is satisfiable can be placed in the LI component of the environment which is used to check the goal. The transformation of the original goal is a conjunction of the goal which produces the dictionary for the constraint being satisfied and the transformed original goal.

Figure 3.8 shows the rules which handle satisfaction and translation of type class constraints themselves.

The rule TC-DICT-I satisfies a type class constraint by using an instance declaration or the super-class relation from the I. Any universally quantified type variables from the entry in I are instantiated with types from the context in which this constraint is being satisfied.

$$\text{TC-DICT-I}\ \frac{E.I(\forall\overline{\alpha}.\kappa\overline{\tau} \Leftarrow \theta) = dt}{E \overset{\text{dict}}{\vdash} (\kappa\overline{\tau} \Leftarrow \theta)[\overline{\tau'}/\overline{\alpha}] \rightsquigarrow dt}$$

$$\text{TC-DICT-REL}\ \frac{\begin{array}{l} E \overset{\text{dict}}{\vdash} \kappa\overline{\tau} \Leftarrow \langle \kappa_1\overline{\tau}_1, \ldots, \kappa_k\overline{\tau}_k \rangle \rightsquigarrow dt \\ E.LI(\kappa_i\overline{\tau}_i) \rightsquigarrow dv_i \end{array}}{E \overset{\text{dict}}{\vdash} \kappa\overline{\tau} \rightsquigarrow \lambda dv.dt(dv_1, \ldots, dv_k, dv)}$$

Figure 3.8: Type class typing/transformation rules for type class constraints

The rule TC-DICT-REL allows a constraint to be satisfied if it can be satisfied under a set of constraints (using an implied instance or super-class, the proof of which will involve TC-DICT-I), and in turn those constraints can be solved.

By way of example, consider the following steps in checking the following call:

$$\texttt{portray}([1, 2, 3], \texttt{IO0}, \texttt{IO})$$

The steps involve producing a dictionary `portrayable(list(int))`, which in turn requires a dictionary for the constraint `portrayable(int)`. Given

$$\begin{array}{rl} V = & \{\texttt{IO0} \mapsto \texttt{io\_\_state}, \texttt{IO} \mapsto \texttt{io\_\_state}\} \\ LI_1 = & \{\texttt{portrayable(int)} \mapsto \texttt{DV1}\} \\ LI_2 = & \{\texttt{portrayable(list(int))} \mapsto \texttt{DV2}\} \end{array}$$

and with $E_1 = E_0 \oplus V$ (where E_0 is as previously defined),

(1) $E_1 \oplus LI_1 \overset{\text{dict}}{\vdash} \texttt{portrayable(list(int))} \Leftarrow \texttt{portrayable(int)} \rightsquigarrow dt_{\texttt{list}}$

(2) $E_1 \oplus LI_1 \overset{\text{dict}}{\vdash} \texttt{portrayable(list(int))} \rightsquigarrow dt_{\texttt{list}}$

(3) $E_1 \oplus LI_1 \oplus LI_2 \overset{\text{goal}}{\vdash} \texttt{portray}([1, 2, 3], \texttt{IO0}, \texttt{IO})$
$\rightsquigarrow \texttt{portray}(\texttt{DV2}, [1, 2, 3], \texttt{IO0}, \texttt{IO})$

(4) $E_1 \overset{\text{dict}}{\vdash} \texttt{portrayable(int)} \rightsquigarrow dt_{\texttt{int}}$

(5) $E_1 \oplus LI_1 \overset{\text{goal}}{\vdash} \texttt{portray}([1, 2, 3], \texttt{IO0}, \texttt{IO})$
$\rightsquigarrow \exists \texttt{DV2}.(dt_{\texttt{list}}(\texttt{DV1}, \texttt{DV2}) \wedge \texttt{portray}(\texttt{DV2}, [1, 2, 3], \texttt{IO0}, \texttt{IO}))$

(6) $E_1 \overset{\text{goal}}{\vdash} \texttt{portray}([1, 2, 3], \texttt{IO0}, \texttt{IO})$
$\rightsquigarrow \exists \texttt{DV1}.(dt_{\texttt{int}}(\texttt{DV1}) \wedge \exists \texttt{DV2}.(dt_{\texttt{list}}(\texttt{DV1}, \texttt{DV2}) \wedge$
$\texttt{portray}(\texttt{DV2}, [1, 2, 3], \texttt{IO0}, \texttt{IO})))$

(dt_{p_int} and dt_{p_list} , defined earlier in this chapter, represent the higher order expressions to produce dictionaries for the type class constraints portrayable(int) and portrayable(list(T)) $\Leftarrow$ portrayable(T) respectively.)

(1) is true by TC-DICT-I. (2) follows from (1) by TC-DICT-REL. (3) is true by TC-PRED (we have skipped the simple steps of checking the types of the arguments being passed to portray). (4) is true by TC-DICT-I. (5) follows from (2) and (3) by TC-LI-ADD. (6) follows from (4) and (5) by TC-LI-ADD.

3.2.3 Type class and instance declarations

$$\text{TC-TYPECLASS}\ \frac{\begin{array}{l}\sigma_i = \forall\overline{\alpha}_i.\mathbf{pred}(\overline{\tau_i}) \Leftarrow \theta_i \\ C = \{(\kappa, |\overline{\alpha}|) \mapsto \\ \qquad\qquad \forall\overline{\alpha}.\{(\pi_1, |\tau_1|) \mapsto \sigma_1, \ldots, (\pi_n, |\tau_n|) \mapsto \sigma_n\} \\ \qquad\qquad\quad \Leftarrow (\theta_{\kappa 1}, \ldots \theta_{\kappa l})\} \\ P = \{(\pi_i, |\tau_i|) \mapsto \forall\overline{\alpha}_i, \overline{\alpha}.\mathbf{pred}(\overline{\tau_i}) \Leftarrow (\kappa\overline{\alpha}), \theta_i)\} \\ I = \{\forall\overline{\alpha}.\theta_{\kappa i} \Leftarrow \kappa\overline{\alpha} \mapsto \lambda dv_1, dv_2.\mathtt{project}_i^{l+n}(dv_1, dv_2)\}\end{array}}{\begin{array}{l}E \overset{\text{decl}}{\vdash} \mathtt{typeclass}\ \kappa\overline{\alpha} \Leftarrow (\theta_{\kappa 1}, \ldots \theta_{\kappa l}) \\ \qquad\qquad \mathtt{where}\ [\pi_1 :: \sigma_1, \ldots, \pi_n :: \sigma_n] \\ \triangleright E \oplus C \oplus P \oplus I \\ \rightsquigarrow \\ \{\pi_i = \lambda dv, dv_1, \ldots, dv_{|\theta_i|}, v_1, \ldots, v_{m_i}. \\ \qquad\quad \exists dv_{imp}.\mathtt{project}_{l+i}^{l+n}(dv, dv_{imp}) \wedge \\ \qquad\quad dv_{imp}\langle dv_1, \ldots, dv_{|\theta_i|}, v_1, \ldots, v_{m_i}\rangle\}\end{array}}$$

Figure 3.9: Type class typing/transformation rules for typeclass declarations

The remaining rules deal with typeclass and instance declarations. We use the notation:

$$E \overset{\text{decl}}{\vdash} X \triangleright E' \rightsquigarrow P$$

to denote the X is a well-formed declaration with respect to the environment E, producing the new environment E' (which takes X into account). The judgement also yields a set of predicates, P, which are inserted into the transformed program. (The transformation part of the judgement is only necessary for typeclass declarations as they are the only declarations which cause new predicates to appear in the transformed program. All other declarations insert information into the environment, but are not transformed into any code directly).

Our rules make use of predicate $\mathtt{project}_n^m/2$, which takes its first (input) argument (which is a dictionary with m entries), and binds its second (output) argument to the nth entry.

$$\texttt{project}_n^m = \lambda \texttt{DVin}, \texttt{DVout}.\exists \texttt{V}_1, \ldots, \texttt{V}_m. \\ (\texttt{DVin} = \texttt{dictionary}(\texttt{V}_1, \ldots, \texttt{V}_m) \wedge \texttt{DVout} = \texttt{V}_n)$$

The rule for `typeclass` declarations,TC-TYPECLASS, is given in figure 3.9. There are four premises above the line for this rule. The first is simply deconstructing the type scheme for each of the type class' methods for use by the later premises. The second, third and fourth premise each construct a new environment component which appears in the environment produced by the judgement below the line. The environment components produced are:

- C, which contains a single entry which records both the signatures of the methods for this class and the constraints which appear on this type class declaration.

- P, which contains an entry for a predicate corresponding to each method of this type class. The signature of this predicate is simply the declared signature of the method, with one additional class constraint. This additional class constraint corresponds to the class being defined.

- I, which contains an entry for each super-class of the class. Each of these entries contains a goal which retrieves the super-class' dictionary from the dictionary for this type class by `project`ing out the appropriate component.

The declaration also yields a predicate in the transformed program for each of the class' methods. Each of these predicates simply `project`s out the appropriate higher order goal from the dictionary corresponding to this type class and applies the appropriate arguments to it.

For example, given the following `typeclass` declaration:

```
:- typeclass coloured_point(T) <= x_y_point(T) where [
    pred get_colour(T, colour)
].
```

the following judgement holds:

$$
\begin{array}{ll}
E_0 \overset{\text{decl}}{\vdash} & \texttt{typeclass coloured\_point(T)} \Leftarrow \texttt{x\_y\_point(T)} \\
& \qquad\qquad \texttt{where } [\texttt{get\_colour} :: \textbf{pred}(\texttt{T}, \texttt{colour})] \\
& \triangleright E_0 \oplus C \oplus P \oplus I \\
& \leadsto \\
& \{\texttt{get\_colour} = \lambda \texttt{DV}, \texttt{V1}, \texttt{V2}.\exists \texttt{DVimp}.\texttt{project}^2_2(\texttt{DV}, \texttt{DVimp}) \wedge \\
& \qquad\qquad \texttt{DVimp}(\texttt{V1}, \texttt{V2})\}
\end{array}
$$

where

$$
\begin{aligned}
C &= \{(\texttt{coloured\_point}, 1) \mapsto \forall \texttt{T}.\{(\texttt{get\_colour}, 2) \mapsto \textbf{pred}(\texttt{T}, \texttt{colour})\} \\
&\qquad\qquad \Leftarrow \texttt{x\_y\_point(T)}\} \\
P &= \{\texttt{get\_colour}, 2) \mapsto \forall \texttt{T}.\textbf{pred}(\texttt{T}, \texttt{colour}) \Leftarrow \texttt{coloured\_point(T)}\} \\
I &= \{\texttt{x\_y\_point(T)} \Leftarrow \texttt{coloured\_point(T)} \mapsto \lambda \texttt{DV1}, \texttt{DV2}.\texttt{project}^2_1(\texttt{DV1}, \texttt{DV2})\}
\end{aligned}
$$

$$
\text{TC-INSTANCE}\;
\frac{
\begin{array}{ll}
(1) & \overline{\tau} = \langle \chi_1 \overline{\alpha_1}, \ldots, \chi_k \overline{\alpha_k} \rangle \\
(2) & E.C(\kappa, k) = \\
& \forall \overline{\alpha}.\{(\pi_1, |\overline{v_1}|) \mapsto \sigma_1, \ldots, (\pi_m, |\overline{v_m}|) \mapsto \sigma_m\} \Leftarrow \theta' \\
(3) & \theta = \langle \theta_1, \ldots, \theta_l \rangle \\
(4) & \theta' = \langle \theta'_1, \ldots, \theta'_{l'} \rangle \\
(5) & LI = \{\theta_1 \mapsto dv_1, \ldots, \theta_l \mapsto dv_l\} \\
(6) & E \oplus LI \overset{\text{dict}}{\vdash} \theta'_i[\overline{\tau}/\overline{\alpha}] \leadsto dt'_i \\
(7) & P_i = \{\pi'_i \mapsto \sigma_i[\overline{\tau}/\overline{\alpha}]\} \\
(8) & E \oplus LI \oplus P_i \overset{\text{pred}}{\vdash} \pi'_i = \lambda \overline{v}_i.G_i \leadsto \pi'_{n_i} = dt_i \\
(9) & I = \{\forall \overline{\alpha_1}, \ldots, \overline{\alpha_k}.\kappa \tau_1, \ldots, \tau_k \Leftarrow \theta \\
& \quad \mapsto \\
& \quad \lambda dv_1, \ldots, dv_l, dv_{out}.\exists dv'_1, \ldots, dv'_{l'}. \\
& \quad\; dt'_1(dv'_1) \wedge \ldots \wedge dt'_{l'}(dv'_{l'}) \wedge \\
& \quad\; dv_{out} = \texttt{dictionary}(dv'_1, \ldots, dv'_{l'}, dt_1, \ldots, dt_m)\}
\end{array}
}{
\begin{array}{l}
E \overset{\text{decl}}{\vdash} \texttt{instance } \kappa \overline{\tau} \Leftarrow \theta \\
\qquad\qquad \texttt{where } [\pi_1 = \lambda \overline{v}_1.G_1, \ldots, \pi_m = \lambda \overline{v}_m.G_m] \\
\triangleright E \oplus I
\end{array}
}
$$

Figure 3.10: Type class typing/transformation rules for `instance` declarations

The rule for `instance` declarations (TC-INSTANCE) is given in figure 3.10. The purpose of each line in this rule is as follows:

(1) checks that each of the elements of the vector of types provided as an

instance in this declaration is a type constructor applied to a vector of distinct type variables.

(2) retrieves the method signatures and super-class constraints for the type class for which this declaration is providing an instance.

(3) deconstructs the constraint from this instance declaration into its constituent constraints.

(4) deconstructs the super class constraint for the type class for which this declaration is providing an instance into its constituent constraints.

(5) constructs an environment component which records each of the individual constraints from (3) in a new dictionary variable.

(6) checks that each of the super-class constraints for this type class is satisfied by the vector of types provided as an instance in this declaration. In doing so, it is allowed to assume that the constraints on this instance declaration have been satisfied. (The proof of satisfaction of those constraints is provided each time that this instance declaration is applied).

(7) constructs an environment component for each class method which maps a mangled predicate name (which we represent by π'_i) to the expected type of that method for this instance.

(8) checks that the goals provided as implementations of the type class methods are well typed. Each method is allowed to assume that dictionaries for the constraints on this instance declaration are found in the locations specified in LI, constructed on line (4). While it is not actually possible for a predicate definition to reference variables outside the scope of that predicate, this does not actually cause a problem; we only use the transformed predicate body as a higher order goal in the dictionary that we construct for this instance, and the variables in LI *are* in scope in that dictionary.

(9) constructs an instance environment containing all the information corresponding to this declaration. It records the vector of types provided as an instance in this declaration, the constraints on this instance declaration and a higher order goal which generates a dictionary corresponding to this declaration. The goal has an argument corresponding to each of the individual constraints on this declaration, and has one argument corresponding to the dictionary produced. The names of the variables

corresponding to the individual constraints on this declaration are the same as the names used on line (4). The higher order goals produced by the transformation on lines (5) and (7) expect to find the appropriate dictionaries in these variables, and whenever this instance declaration is applied, the appropriate dictionaries are passed into those variables in the appropriate arguments. The goal corresponding to this instance declaration itself is a conjunction of the goals which produce the dictionaries corresponding to the super-classes of this class and a unification which constructs the dictionary (which contains all the super-class dictionaries and a higher order goal corresponding to each method).

Lines (6) and (10) check that the instance is correct (and line (1) enforces a simple syntactic condition).

For example, the correctness of the instance declaration:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
    pred(portray/3) is portray_list
].
```

is checked as follows:

First, this class has no super classes, so there is no proof required for line (6). Second, given

$$\begin{aligned} P = & \quad \{(\mathtt{portray}', 3) \mapsto \mathtt{T}.\mathbf{pred}(\mathtt{list(T)}, \mathtt{io\_\_state}, \mathtt{io\_\_state})\} \\ LI = & \quad \{\mathtt{portrayable(T)} \mapsto \mathtt{DV}\} \end{aligned}$$

and with E_0 as previously defined, it follows fairly simply that

$$\begin{aligned} E_0 \oplus LI \oplus P \overset{\text{pred}}{\vdash} \quad & \mathtt{portray}' = \lambda\mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{portray\_list}(\mathtt{A}, \mathtt{B}, \mathtt{C}) \\ & \leadsto \\ & \mathtt{portray}'_3 = \lambda\mathtt{DV}, \mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{portray\_list}(\mathtt{DV}, \mathtt{A}, \mathtt{B}, \mathtt{C}) \end{aligned}$$

Therefore, the following judgement holds:

$$\begin{aligned} E_0 \overset{\text{decl}}{\vdash} \mathtt{instance} \quad & \mathtt{portrayable(list(T))} \Leftarrow \mathtt{portrayable(T)} \\ & \mathtt{where}\ [\mathtt{portray} = \lambda\mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{portray\_list}(\mathtt{A}, \mathtt{B}, \mathtt{C})] \\ & \triangleright E \oplus I \end{aligned}$$

where

$$
\begin{aligned}
I = \quad & \{ \\
& \forall \mathtt{T}.\mathtt{portrayable}(\mathtt{list}(\mathtt{T})) \Leftarrow \mathtt{portrayable}(\mathtt{T}) \\
& \qquad \mapsto \lambda \mathtt{DV}, \mathtt{DVout}. \\
& \qquad\qquad \mathtt{DVout} = (\mathtt{dictionary}(\lambda \mathtt{A}, \mathtt{B}, \mathtt{C}.\mathtt{portray\_list}(\mathtt{DV}, \mathtt{A}, \mathtt{B}, \mathtt{C}))) \\
& \}
\end{aligned}
$$

As can be seen in the resulting instance environment, the dictionary corresponding to this instance declaration contains a single method that "captures" the dictionary for the constraint on the instance declaration.

Each time that a dictionary corresponding to this `instance` declaration is constructed, the dictionary corresponding to the `portrayable(T)` constraint is supplied, and this dictionary becomes part of the closure corresponding to the `portray` method.

3.3 Type classes in Mercury

Our treatment so far has been concerned only with type checking and inference. We now consider the problem of integrating the type system presented above into Mercury, a purely declarative logic programming language that has not only a strong type system but strong static mode, determinism and uniqueness systems as well [31]. In the context of the full language, our system of type classes must be a little more complicated.

Our main concern is the need to ensure that a call to a class method, where the actual code called is not chosen until run time, does not result in a mode, determinism or uniqueness error. To make this possible, we extend the syntax of `typeclass` declarations:

```
:- typeclass x_y_point(T) where [
    pred get_x(T, float),
    mode get_x(in, out) is det,
    ...
].
```

Each method of the type class also has a `mode` declaration which gives information about the modes and uniqueness of the arguments and the determinism of that mode of the predicate. As with all Mercury predicates, there may be more than one `mode` declaration for each method.

Checking the correctness of the methods in an `instance` declaration is no longer just a matter of checking the type correctness of the method with respect to the corresponding type class method; it is also necessary to check that the instance method is mode, uniqueness and determinism correct. Our solution to this is quite elegant; the approach that we adopted earlier for type checking the instance declaration via program transformation gives us these things "for free".

Consider the `method_for__c1___list__arity1__m_2` predicate introduced to check the correctness of the instance declaration in section 3.1.4. The introduction of `method_for__c1___list__arity1__m_2` also allows us to check the mode, determinism and uniqueness correctness of `p` as an implementation of `m`. For example, if `c1` included the mode declarations:

```
:- typeclass c1(T) where [
    pred m(T, T2) <= c2(T2),
    mode m(in, out) is det,
    mode m(out, in) is nondet
].
```

then for the following instance declaration:

```
:- instance c1(list(T)) <= c(T) where [
    pred(m/2) is p
].
```

we add the following to the generated program:

```
:- pred method_for__c1___list__arity1__m_2(list(T), T2)
       <= (c2(T2), c(T)).
:- mode method_for__c1___list__arity1__m_2(in, out) is det.
:- mode method_for__c1___list__arity1__m_2(out, in) is nondet.

method_for__c1___list__arity1__m_2(X, Y) :-
    p(X, Y).
```

The compiler treats an introduced predicate just like any other, so even though we introduced the predicate to do *type* checking, the appropriate phases of the compiler also perform mode, determinism and uniqueness checking. In addition, just as this allows the type of `p` to be more general than that of the introduced predicate, the mode, determinism and uniqueness may also be more general. For modes, this means that an *implied mode* may be used. For determinism, it means that the procedure may have a more strict determinism (e.g. it may be known never to fail even though it is allowed to). For uniqueness, this means that an argument may be returned that is guaranteed to be unique, even though this is not required.

3.3.1 Preservation of Mercury's static correctness properties

Armed with the knowledge that class method calls themselves cannot cause mode, determinism or uniqueness errors, it is simple to show that the transformations described in section 3.1.5 preserve mode, determinism and uniqueness correctness. The reasoning for this is as follows.

Our transformation does not introduce any extra references to any of the source program's variables, and thus cannot alter their mode or uniqueness properties. To show that the transformation preserves mode and uniqueness correctness, it is therefore sufficient to show that references to the *introduced* variables have no such errors.

First, the introduced variables are not required to be unique, which means uniqueness errors cannot arise. Second, for all the introduced variables, the producer of the variable is inserted before all consumers. The dictionary variables are always bound by the *caller* of the predicate, and the code to produce or extract the dictionary is always inserted before the call.

In the case of producing a dictionary from an instance declaration with constraints upon it, the code to produce the dictionary takes as argument the dictionaries corresponding to those constraints. This never introduces a mode error as, once again, the producer of the variable is always inserted before its consumer.

Any variables introduced by the transformation inside a negated context are either bound inside the negation (by a goal introduced by our transformation) and never used outside the negation, or are input to the negated goal (where the dictionary is passed from the head of a clause to a call in the body of the clause). Both alternatives preserve mode correctness. This also applies to the conditions of if-then-elses.

All the goals introduced by our transformation (those which construct dictionaries and those that select methods from dictionaries) have exactly one solution. Inserting these goals before other goals has no direct effect on determinism, since the goal $G_1 \wedge G_2$ has exactly the same determinism as G_2 if G_1 always has exactly one solution. It could possibly have an indirect effect if the inserted goal defines the value of a variable that is visible from outside the conjunction, since this might suppress Mercury's implicit pruning of nondeterministic goals with no outputs. However, the goals introduced by the transformation either produce no variables or produce variables whose only consumer is the transformed original goal, so this cannot happen. Therefore the determinism of a procedure is never changed by our transformation.

In the current version of Mercury, a higher-order value does not stand for a predicate; it stands for a procedure, one mode of a predicate. Therefore the dictionaries we produce have one entry not for each predicate in the interface of the type class, but for each mode of each predicate in the interface of the type class. Therefore, the code that performs a class method call must choose the correct procedure of the called method from the dictionary. This is not a problem, since the mode of the method call is known at compile time.

3.4 Typing the transformation

The description of the transformation that we have given in this chapter does not explicitly gives the types of predicates and values in the transformed program.

Nonetheless, preserving types in the transformation is conceptually a simple matter — we need merely assign a type to the dictionary for each class, and use those types to assign types to the introduced goals.

For example, the dictionary corresponding to the typeclass `x_y_point/1` could be represented by the following Mercury type:

```
:- type x_y_point(T)
        ---> dictionary(
                pred(T, float),     % get_x/2
                pred(T, float, T),  % set_x/3
                pred(T, float),     % get_y/2
                pred(T, float, T)   % set_y/3
        ).
```

The type class `coloured_point/1` would correspond to:

```
:- type coloured_point(T)
        ---> dictionary(
                x_y_point(T),       % parent class
                pred(T, colour)     % get_colour/2
        ).
```

The goal which extracts a dictionary for `x_y_point/1` would have type

$$\forall \mathtt{T}.\mathbf{pred}(\mathtt{coloured\_point(T)}, \mathtt{x\_y\_point(T)})$$

whereas the goal corresponding to the

$$\forall \mathtt{T}.\ \mathtt{portrayable(list(T))} \Leftarrow \mathtt{portrayable(T)}$$

instance declaration would have type

$$\forall \mathtt{T}.\mathbf{pred}(\mathtt{portrayable(T)}, \mathtt{portrayable(list(T))})$$

However, a complication arises when we consider the typing of a dictionary that corresponds to a typeclass that has a method which universally quantifies over type variables that are different to those quantified over by the type class itself:

```
:- typeclass c(T) where [
    pred m(T, list(T1), list(T1)) <= c2(T1)
].
```

The dictionary corresponding to this type class needs to include a polymorphic predicate. Although there is no way of defining such a type in Mercury, we could expect the type to look something like:

```
:- type c(T)
        ---> dictionary(
                all [T1] pred(T, list(T1), list(T1)) <= c2(T1)
        ).
```

The predicate, when unpacked from the dictionary, can be applied to *any* type `T1`. Universally quantifying over a type inside another universal quantifier (in this case, universally quantifying over `T1` inside the implicit universal quantifier over `T`) is called *second order* polymorphism.

A program which is type correct under a first order polymorphic type system using type classes is transformed into another program which is not typable under *that* type system.

3.5 Implementation

The Melbourne Mercury implementation has a sophisticated run time type information (RTTI) system [18]. For every type variable in a predicate's signature, an extra argument known as a "type-info" is added to that predicate. The type-info contains low-level information about the run-time representation of values of the type to which the type variable in question is bound. This information is used by the user-level dynamic typing and RTTI libraries, as well as by subsystems which require low-level knowledge of run-time representations, such as the Melbourne Mercury debugger [99]. The transformation to introduce type-infos is conceptually quite similar to the transformation to introduce type class dictionaries that we have described. In fact, if we consider type-infos to be dictionaries for some "universal" type class of which all types are a member, and consider every type variable in every predicate's signature as being constrained to be in this type class, then the transformation that results from introducing dictionaries for this universal type class is precisely the transformation used to introduce type-infos. As such, the type-info and dictionary mechanisms co-exist quite simply. Both transformations are implemented in a single pass of the compiler.

The main complication in implementing type classes in the Melbourne Mercury compiler is that the compiler's intermediate program representation uses first order types, whereas the transformation we have described requires second order types.

By contrast, the Glasgow Haskell Compiler uses a second order intermediate code representation [81]. This allows the transformation to be explicitly typed as in [27], (although the system shown in [27] does not allow type class methods to be polymorphic in type variables other than the type class' parameters).

However, to our knowledge second order type systems have never been explored in the context of logic programming. This remains an area for future research. Thus, rather than changing the Melbourne Mercury compiler's intermediate representation to use second order typing, we have instead implemented the class method call mechanism through an untyped layer. We first show a simplified version of this mechanism to show how our untyped layer allows us to bypass the second order polymorphism problem, then show a version (that we have actually implemented) which, at the price of a more complicated implementation, allows us to make the dictionary creation process much more efficient.

3.5.1 A simple untyped layer

Rather than explicitly defining a (troublesome) type for each dictionary, we can instead represent all dictionaries by a single built in type. Using this technique, dictionaries cannot be constructed in pure Mercury code. Instead, dictionaries are implemented using low-level constructs specific to the compiler's target language.

The methods in a dictionary are a constant for any given instance declaration, although constraints on an instance declaration mean that the closures corresponding to the methods may need to be built at run time from the dictionaries corresponding to those constraints.

A simple scheme might generate low-level code for each instance to construct a dictionary given the other dictionaries from the instance declaration's constraints by manually creating the partially applied (but still fully polymorphic) methods and storing these in a block of memory (whose size depends upon the type class in question). A reference to this block of memory would then be returned as the dictionary itself.

Calling a method from this dictionary would then be simple; even though the methods stored inside the dictionary are polymorphic, we extract the methods from the dictionary each time that we want to call them. As such, we can just use first order polymorphism to type the extraction process. For example, the transformation of the `do_to_list/3` predicate from our example above could be:

```
:- pred do_to_list(dictionary, T, T1, list(T1)).
do_to_list(DV, V1, V2, V3) :-
    extract_do_to_list_from_dictionary(DV, Imp),
    Imp(V1, V2, V3).
```

where the `extract_do_to_list_from_dictionary` predicate is implemented in the run time system, and extracts the appropriate partially applied method from the dictionary structure.

The type of `extract_do_to_list_from_dictionary` could simply be:

```
:- pred extract_do_to_list_from_dictionary(dictionary,
                                           pred(T, T1, list(T1))).
```

This type says that for any type `T1`, we can retrieve an implementation of `do_to_list/3` which can be applied to values of that type. It does not capture the fact that the method, once extracted, is still fully polymorphic. However, given that we know before it is extracted the type to which it will be applied, this does not pose a problem.

3.5.2 Avoiding excessive closure creation

The method outlined in the previous section does allow the type class transformation to work in the presence of polymorphic methods, but at the cost of requiring the compiler back end to handle dictionary construction and deconstruction specially. This is, of course, undesirable as it means that more work must be done when porting to another back end. Given that this price has been paid, we can reduce the overhead of the dictionary creation mechanism by taking a step further from the pure source-to-source transformation described earlier in this chapter.

One key inefficiency in both dictionary creation processes discussed so far (the pure source-to-source method, and the method using low level code in the run time system), is that for each dictionary, every method must have a closure containing all the dictionaries for the constraints on the `instance` declaration to which that dictionary corresponds, even if that method is not used over the lifetime of that dictionary. Dictionary construction time would be a linear function of the number of methods of a given type class. Figure 3.11 shows this structure.

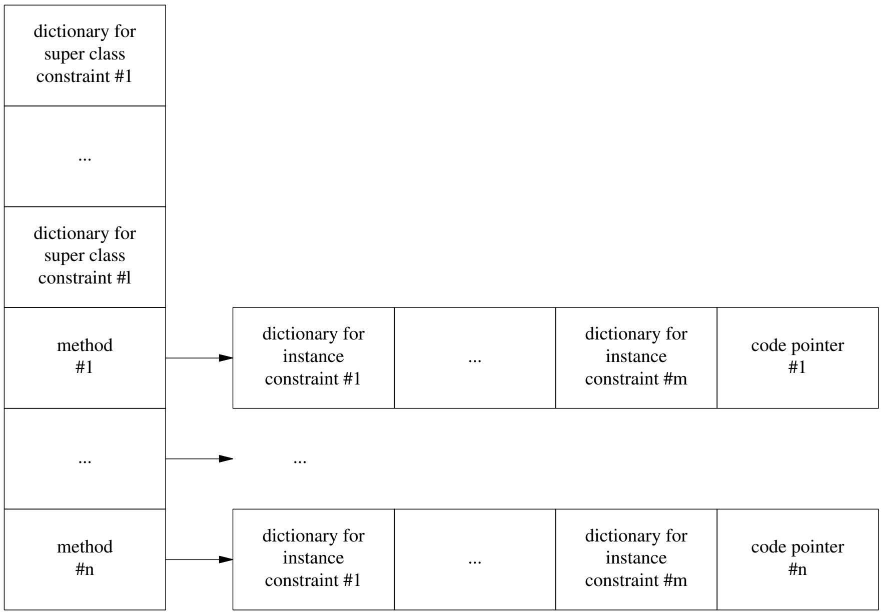

Figure 3.11: Simple low-level dictionary representation

Our solution is to construct just one closure-like object for each dictionary, and to use a different call mechanism for class methods than for other higher order calls. The resulting dictionary structure is shown in figure 3.12.

Rather than allocating memory for the dictionaries from instance constraints to each method, we do so just once for each dictionary. In addition, we separate the structure into two parts. The first part (shown vertically in figure 3.12) contains just the code pointers for the methods, and is static data for any given instance. The second part (shown horizontally) contains the dictionaries which are added dynamically. This allows us to avoid the overhead of allocating memory for the methods each time we create a dictionary.

Using a structure such as this, where the methods are not stored as true closures, means that we must use a specialised calling mechanism to pass the extra dictionaries at run time. Rather than unpacking the dictionary to retrieve the closure, returning this to Mercury, then calling the closure with the arguments specific to this call, we use a mechanism whereby the arguments specific to this call are passed with the dictionary to a routine which unpacks the extra

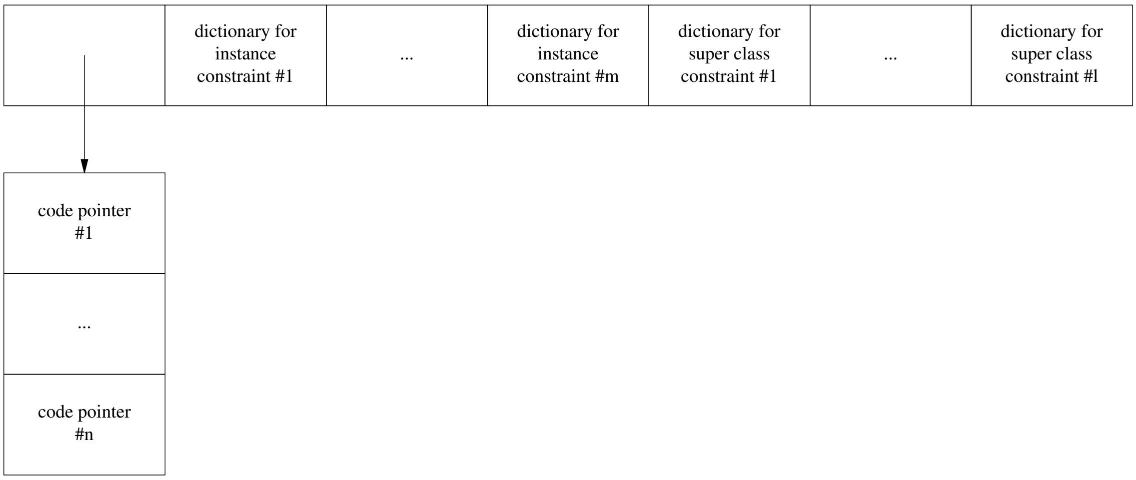

Figure 3.12: Optimised low-level dictionary representation

dictionaries and passes these along with the input arguments to the method implementation.

This does introduce extra complexity to the implementation, but we feel that this overhead is worth it, considering that, even with the simple approach outlined in the previous section, it is necessarily to add some type class specific mechanism to the back end. In the case of the Melbourne Mercury compiler's low-level C back end [100], the mechanism we have just described requires only 48 lines of extra code in the run time system.

3.6 Remarks

We have rigorously described a type class system for higher order logic programming. This type system has been implemented in the Melbourne Mercury compiler, and has been part of the Mercury distribution since version 0.8. We have given a set of typing rules which give a definition of type correctness and defined the source-to-source transformation used to implement the scheme.

The transformation produces a program which in general would require a second order type system. We have given details of an implementation which works in a first order typed back end by using a simple untyped layer. Second order typing in the context of logic programming remains a topic for future work.

We have shown that type classes add significant expressiveness to a logic programming language. Future work will look at making this system even more expressive by extending the system with constructor classes [44] and functional dependencies [48].

In the next chapter, we extend the type system we have described in this chapter with *existential typing* to yield an even more expressive type system.

Chapter 4

Existential Types

Modern programming languages typically provide some kind of abstract type mechanism. In Mercury, a type declaration in the interface of a module where the declaration does not list the function symbols of the type means that the name of the type is exported, but its implementation is not. Similarly, Haskell allows an algebraic type to be exported without its constructors while Standard ML and Miranda have an `abstype` construct which explicitly allows an abstract type to be declared. Such approaches only allow a single implementation of an abstract type.

Abstract types were given a sound theoretical basis by Mitchell and Plotkin [69] (and around the same time by Cardelli and Wegner [8]). This theory used second order lambda calculus to show that an abstract type can be represented by an *existentially quantified type variable.* We often say that a value whose type is an existentially quantified type variable is *existentially typed.*

This concept was extended by Läufer and Odersky, who define a system of first class abstract types based on existential types for both ML [53, 54] and Haskell [52] which provided strong static typing and type reconstruction. The system allows the programmer to provide multiple implementations of a given abstract type, and most importantly allows the construction of heterogeneous collections of different implementations of the same abstract type. This system was implemented in the ML implementation CAML-Light [63], and in the Chalmers Haskell B compiler. This mechanism also formed the basis for the implementation of existential types in the Glasgow Haskell compiler (although this is yet to become an official part of the Haskell language).

Existential types, in a slightly different form, were also implemented in Hope+C [80], although as noted in [51], the typing rules given in [80] are not sound.

In this chapter, we present an existential type system for higher order logic programming. Having presented type classes in the previous chapter as an extension to our Mycroft-O'Keefe style type system, we now present existential types as another extension. We first present existential types in their own right, and then show the extremely expressive system that results from combining both type classes and existential types.

This is, to our knowledge, the first such type system for a logic programming language. Furthermore, unlike other existing systems that support existential types, our system allows existential quantifiers on the types of functions and predicates, as well as on the types of data constructors. By careful handling of higher order values, we are able to avoid the type soundness problems typically associated with allowing existential quantifiers on the types of functions and predicates.

We present a concrete implementation of this system in the Melbourne Mercury compiler. We investigate the interactions between existential types and modes (data-flow) in logic programming, with particular attention to Mercury's static mode system, which performs data-flow analysis and code reordering.

This chapter begins by describing our system's support for existentially quantified components of data structures and existentially quantified types for the return values of predicates in turn, and gives a set of typing rules for the system. We then combine this system with the type class system defined in the previous chapter and give a set of typing rules for the whole system.

4.1 Existentially quantified function symbol types

The types of data constructors in our basic, Mycroft-O'Keefe based, type system are allowed to contain universally quantified type variables. For example, given the following Mercury type definition:

```
:- type list(T) ---> [] ; [T | list(T)].
```

the type variable '`T`' is implicitly universally quantified across the definition of both function symbols. As a consequence, the type of the elements inside the list is "exposed" in the type of the resulting list. A graphical representation of this type (a *type graph*) is shown in figure 4.1. The universally quantified type variable `T` can be instantiated with any type, having the effect of binding the type variables all the way down the type tree.

However, in many situations it is useful to be able to instantiate a polymorphic type with a variety of different types, but to have those types stay hidden outside the function symbol in which they are bound. The mechanism we propose for doing so is *existential quantification* of type variables. For example,

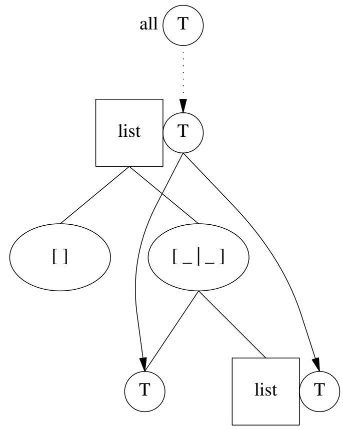

Figure 4.1: Type graph representing the `list(T)` type

we may want to create a type which stores values of any type, along with a predicate which converts those values into integers:

```
:- type to_int ---> some [T] f(T, pred(T, int)).
```

The definition of the type `to_int` declares the type to have a single function symbol `f`. `some [T]` denotes that the type variable `T` is existentially quantified across the function symbol definition. The function symbol `f` has two arguments: a value of some type, and a predicate which relates a value of that type to an integer. Given a value of type `to_int`, we know nothing about the type `T` of the first value inside it, other than the fact that it is the same type as is expected by the first argument of the predicate. For example, the following would be valid values of type `to_int`.

```
f(42, (pred(A::in, B::out) is det :- B = A + 12))
f("hello", (pred(A::in, B::out) is det :- string__length(A, B)))
```

(where `string__length` is a library predicate which returns the number of characters in a string.)

A type graph for the `to_int` type is given in figure 4.2.

One of the most important benefits of this mechanism is that it allows us to build *heterogeneous collections*. For example, we can build a list where each element is of type `to_int`:

```
[f(42, (pred(A::in, B::out) is det :- B = A + 12)),
 f("hello", (pred(A::in, B::out) is det :- string__length(A,B)))]
```

The types of the elements inside the `f` function symbol of each element in the list is invisible outside the list elements.

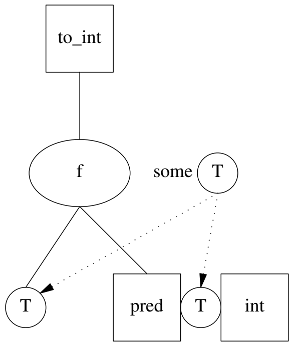

Figure 4.2: Type graph representing the `to_int` type

4.1.1 Construction and deconstruction

When constructing a list of any type `T`, we take a value of type `T` and a value of type `list(T)` to produce a value of type `list(T)`. Similarly, when deconstructing the list, we take a value of type `list(T)` and produce a value of type `T` and a value of type `list(T)`. We are even able to use the same syntax for construction and deconstruction. For example, the following `cons` predicate can be used to construct or deconstruct a list node:

```
:- pred cons(T, list(T), list(T)).
cons(X, Xs, [X|Xs]).
```

Existential types complicate this situation considerably. When constructing a value which contains an existentially quantified component, we can use a value of *any* type. On the other hand, when deconstructing such a value, the existentially quantified component that we retrieve is of some type *that we do not know*. With the introduction of existential types, construction and deconstruction become, from a typing point of view, quite different operations.

Consider our `to_list` example above, we can say that our constructor `f` when thought of as a function has type:

$$\mathtt{f} :: \forall \mathtt{T}.(\mathtt{T} \times \mathbf{pred}(\mathtt{T}, \mathtt{int})) \rightarrow \mathtt{to\_int}$$

However, the corresponding deconstruction function has type:

$$\mathtt{f}^{-1} :: \exists \mathtt{T}.\mathtt{to\_int} \rightarrow (\mathtt{T} \times \mathbf{pred}(\mathtt{T}, \mathtt{int}))$$

Given this, we are led to the conclusion that we must somehow know whether a function symbol is being used in a construction or a deconstruction before we can consider the type correctness of a goal.

Our solution is to provide a syntactic distinction between construction and deconstruction by annotating all constructions which involve an existentially quantified component with the keyword `new`. For example, the following goal constructs a new value of type `to_int` and assigns it to the variable `X`.

```
X = 'new f'(42, (pred(A::in, B::out) is det :- B = A + 12))
```

Similarly, we could construct a heterogeneous list:

```
Xs = ['new f'(42, (pred(A::in, B::out) is det :- B = A + 12)),
      'new f'("hello", (pred(A::in, B::out) is det :-
                                        string__length(A, B)))]
```

The syntax for deconstruction is unchanged:

```
X = f(V, P)
```

In this case, the value `X` is deconstructed into values whose types (for some unknown type `T`) are `T` and `pred(T, int)` respectively. It would be an error to try to unify `V` or `P` with values of any other type. For example, the following is not legal

```
Error = f(42, _)
```

because we unify a value of type `int` with a value whose type we do not know.

The following example (translated into Mercury from the ML example in [51]) defines a predicate which finds the smallest value in a heterogeneous list with respect to the integer value produced by the predicate in the `to_int` structure.

```
:- pred min(list(to_int), to_int).
min([X], X).
min([X1, X2 | Xs], Min) :-
    min([X2 | Xs], Min0),
    Min0 = f(V0, P0),
    P0(V0, N0),
    X1 = f(V1, P1),
    P1(V1, N1),
    ( N0 =< N1 ->
        Min = Min0
    ;
        Min = X1
    ).
```

We could use this predicate in a context such as:

```
...
Xs = ['new f'(42, (pred(A::in, B::out) is det :- B = A + 12)),
      'new f'("hello", (pred(A::in, B::out) is det :-
                                      string__length(A, B)))],
min(Xs, Min)
...
```

in which case, Min would become bound to

```
f("hello", (pred(A::in, B::out) is det :- string__length(A, B)))
```

4.2 Existentially quantified predicate types

The mechanism we have just described for deconstructing a value with existentially typed components is quite different to the mechanisms previously proposed in [51, 69]. Although they use different mechanisms, both those systems only allow such values to be deconstructed in a such manner that the existentially typed values are visible only within some limited scope enclosed by the deconstruction. Furthermore, to maintain type soundness no existentially quantified type variable is allowed to 'escape' from this scope unless wrapped inside an appropriate data constructor. This effectively means that no function is allowed to have an existentially typed return value.

In Läufer's system, a value can be deconstructed by pattern matching in a `let` expression. For example, given the ML data type:

```
datatype ToInt = f of 'a * ('a -> int)
```

then if `x` is a value of type `ToInt`, then the following is a well-typed expression of type `int`:

```
let val f(v,p) = x in
    p v
end
```

Inside the body of the let expression, the existentially typed values appear just like values of *universally quantified* types that have been passed in from some outside scope; they are simply values of some type that is unknown.

However, no existentially quantified type variable is allowed to appear in the result type of the body of the `let` expression, so the following is not legal since its type would be existentially quantified:

```
let val f(v,p) = x in
    v
end
```

In logic programming, rather than deconstructing a value for use in some nested expression, a value that is deconstructed is visible throughout the entire predicate body. This is a rather different mechanism, and as a result we cannot directly translate the existential type systems designed for functional languages into logic programming.

Our approach is to allow the existential types to 'escape'; we allow existentially quantified type variables to appear in the types of predicates (and functions), but by carefully handling the typing of higher order expressions, we are able to maintain type soundness.

For example, we allow the following predicate declaration, in which we deconstruct a value with existentially typed components, and return those components in separate, existentially quantified values:

```
:- some [T] pred deconstruct_to_int(to_int, T, pred(T, int)).
deconstruct_to_int(f(V, P), V, P).
```

Note that the corresponding predicate which constructs one of these values simply has the implicitly universally quantified type:

```
:- pred construct_to_int(to_int, T, pred(T, int)).
construct_to_int('new f'(V, P), V, P).
```

Calling a predicate with an existentially typed return value gives us a value whose type we do not know. For example, after the following call:

```
    ...
    deconstruct_to_int(X, V, P),
    ...
```

(where `X` is a value of type `to_int`), the type of variable `V` is unknown, and the only useful thing that can be done with it is to pass it as an argument to the predicate `P`.

Most importantly, each time we call the predicate `deconstruct_to_int`, we are returned values *of a different existentially quantified type.* For example, the following is not allowed:

```
    ...
    X1 = 'new f'(42, (pred(A::in, B::out) is det :- B = A + 12)),
    X2 = 'new f'("hello", (pred(A::in, B::out) is det :-
```

```
                                              string__length(A, B))),
    deconstruct_to_int(X1, V1, P1),
    deconstruct_to_int(X2, V2, P2),
    V1 = V2
    ...
```

The existentially quantified values cannot be unified as they may be of different types. The only way of comparing them is to compare the integers returned by the predicates stored inside the `to_int` structure.

```
    ...
    deconstruct_to_int(X1, V1, P1),
    deconstruct_to_int(X2, V2, P2),
    P1(V1, N1),
    P2(V2, N2),
    N1 = N2
    ...
```

At a call to a predicate with existentially quantified arguments, or at the deconstruction of a value with existentially quantified components, we simply make any existentially quantified type variable correspond to a fresh type variable which is not used elsewhere in the predicate definition. This type variable may not be bound anywhere. In particular, it may not be bound by a universal quantifier; if this type variable appears in the type of the enclosing predicate, it is existentially quantified.

A predicate may also be declared to have an existential type when, internally, there is a more specific type that could be assigned to it. For example, in the following predicate, we return a value of a known type (`int`) through an existentially typed parameter. The existential type simply abstracts away the concrete type.

```
:- some [T] pred p(T).
p(1).
```

Without careful handling of higher order programming, a type system such as this (which allows predicate types to contain existentially quantified type variables), could easily be unsound. Specifically, if the type of a value returned by a higher order goal is existentially typed, then the type checker is not able to statically distinguish between the different types that would be returned by different invocations of that goal.

For example, consider the following predicate which accepts a higher order goal and applies it to several values:

```
:- pred q(pred(int, T)).
q(P) :-
    P(1, X),
    P(2, Y),
    X = Y.
```

Now, if we also had the following predicate which binds its second argument to the integer '42' if its first argument is `1`, or to the string `"hello"` if its first argument is 2:

```
:- some [T] pred r(int, T).
r(N, V) :-
    (
        N = 1,
        X = 'new f'(42,
            (pred(A::in, B::out) is det :- B = A + 12))
    ;
        N = 2,
        X = 'new f'("hello",
            (pred(A::in, B::out) is det :- string__length(A, B)))
    ),
    X = f(V, _).
```

then we should not be allowed to write the following:

```
    ...
    P = (pred(A::in, B::out) :- r(A, B)),
    q(P)
    ...
```

as executing this would cause a run-time type error in `q/1` at the unification:

```
X = Y
```

The key observation is that the higher order goal `P` has a second argument of some type `T` *which is not fixed until the expression is fully applied.*

Our approach is simply to disallow existentially quantified type variables from appearing in the types of the arguments to higher order goals. We will make this concept precise in section 4.3.

4.2.1 Type data flow

A useful way of thinking about existential typing is in terms of the *data flow* of the types involved.

Universally quantified types exhibit very simple data flow: all types flow in a top-down manner. By this, we mean that a universally quantified type variable is always bound by the caller; the type is "passed" in from above it in the call tree. The same is true even if the value with a universally quantified type is an output. Consider the following definition of the `zip/3` predicate (from the Mercury standard library) which creates a list (the third argument) by alternating elements from the lists passed in the first two arguments. The definition is mutually recursive with an auxiliary predicate `zip2/3`.

```
:- pred zip(list(T), list(T), list(T)).
:- mode zip(in, in, out) is det.

zip([], Bs, Bs).
zip([A|As], Bs, [A|Cs]) :-
    zip2(As, Bs, Cs).

:- pred zip2(list(T), list(T), list(T)).
:- mode zip2(in, in, out) is det.

zip2(As, [], As).
zip2(As, [B|Bs], [B|Cs]) :-
    zip(As, Bs, Cs).
```

A data flow representation of these predicates is given in figure 4.3. In the figure, type variables are shown explicitly as arguments in angled brackets[1]. The dotted arrows show the direction of flow of type information. In this case, all type variables are universally quantified, so all type data flow is top-down.

Conversely, the existential quantification of type variables corresponds to *bottom-up* data flow of type information. As a consequence, existentially typed arguments are always outputs of a predicate — a variable's value cannot be determined before its type.

For example, consider the following predicate which unpacks a `to_int` and returns the unpacked value with a slightly modified higher order goal to manipulate that value.

```
:- some [T] pred double_to_int(to_int, T, pred(T, int)).
double_to_int(X, V, P) :-
```

[1]This corresponds very closely to our implementation; the Melbourne Mercury compiler implements polymorphism by introducing an extra argument called a "type-info" [18] to every predicate for each type variable that appears in that predicate's signature. A type-info effectively encodes all the information included in a "`:- type`" declaration.

```
zip(<T>, [], Bs, Bs).

zip(<T>, [A|As], Bs, [A|Cs]) :-

    zip2(<T>, As, Bs, Cs).
```

```
zip2(<T>, As, [], As).

zip2(<T>, As, [B|Bs], [B|Cs]) :-

    zip(<T>, As, Bs, Cs).
```

Figure 4.3: Type data flow for universal quantification

```
deconstruct_to_int(X, V, P0),
P = (pred(A::in, B::out) is det :- P0(A, B0), B = 2*B0).
```

The resulting bottom-up type data flow is shown in figure 4.4.

```
double_to_int(<T>, X, V, P) :-

    deconstruct_to_int(<T>, X, V, P0),

    P = (pred(A::in, B::out) :- P0(A, B0), B = 2*B0).
```

Figure 4.4: Type data flow for existential quantification

Finally, existentially typed function symbols conceptually correspond to a type being stored inside the function symbol itself. (The implementation in the Melbourne Mercury compiler does just this; a value representing the type is stored along with the existentially typed value). Figure 4.5 shows the type data flow involved in the `deconstruct_to_int/3` predicate.

Existential quantification adds to the language the possibility of bottom-up flow of type information. We can say that an existential quantifier is a *producer* of a type, whereas a universal quantifier is a *consumer*. The key to checking

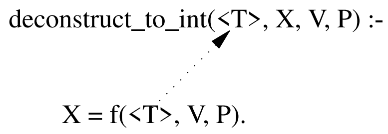

Figure 4.5: Type data flow for deconstruction

type correctness in a system with existential types is to ensure that there is exactly one producer of each existential type.

4.3 Typing rules

We now present a set of typing rules which give a rigorous definition of type correctness in our system.

First, our abstract syntax from figure 2.1 has to be extended to allow for `new` annotations on function symbols. Our new abstract syntax for terms is:

$$\begin{array}{lcll} \textsf{Term} & t & ::= & v \\ & & | & f(\bar{t}) \\ & & | & \texttt{new}\ f(\bar{t}) \\ & & | & \lambda\bar{v}.G \end{array}$$

The rest of the abstract syntax is unchanged.

4.3.1 Type language

Our modified type language is shown in figure 4.6. Once again, we distinguish between polymorphic types which may quantify type variables and simple types which do not. Predicates and function symbols are assigned polymorphic types, but terms are not.

$$\begin{array}{llll} \textsf{Type variable} & \alpha & & \\ \textsf{Type constructor} & \chi & & \\ \textsf{Simple type} & \tau & ::= & \alpha \\ & & | & \chi\overline{\tau} \\ & & | & \mathbf{pred}(\overline{\tau}) \\ \textsf{Predicate type} & \sigma & ::= & \forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\overline{\tau}) \\ \textsf{Function symbol type} & \varphi & ::= & \forall\overline{\alpha}.\exists\overline{\beta}.\overline{\tau} \longrightarrow \chi\overline{\alpha} \end{array}$$

Figure 4.6: Type language for a existential types

The main difference between the typing rules we present here and those from chapter 2 is that here we keep track of existentially quantified type variables produced in a goal, and ensure that each such type variable has exactly one producer. Our typing rules use judgements of the form:

$$(E, \Omega) \vdash X$$

to represent that X is well-typed with respect to the environment E (which is as defined in chapter 2), and where Ω is the set of existentially quantified variables that were produced by X. (We could have made Ω part of the environment, but almost every rule uses Ω so this structure leads to a slightly simpler presentation).

As in chapter 2, we use the symbols $\overset{\text{goal}}{\vdash}$, $\overset{\text{val}}{\vdash}$, $\overset{\text{pred}}{\vdash}$ and $\overset{\text{decl}}{\vdash}$ to distinguish between judgements over goals, values, predicates and declarations respectively:

- $(E, \Omega) \overset{\text{goal}}{\vdash} G$ denotes that G is a well typed goal with respect to E, and produces the type variable set Ω.
- $(E, \Omega) \overset{\text{val}}{\vdash} t : \tau$ denotes that t is a well typed term of type τ with respect to E, and produces the type variable set Ω.
- $E \overset{\text{pred}}{\vdash} \pi = \lambda\overline{v}.G$ denotes that $\pi = \lambda\overline{v}.G$ is a well typed predicate definition with respect to E.
- $E \overset{\text{decl}}{\vdash} X \triangleright E'$ denotes that X is a well formed declaration with respect to E, and yields the new environment E' which takes the declaration into account.

Note that $\overset{\text{pred}}{\vdash}$ judgements do not involve a set of produced type variables. This is because any existentially quantified type variables are captured explicitly in the type signature of the predicate in question.

We use $\Omega_1 \uplus \Omega_2$ to denote the disjoint union of sets Ω_1 and Ω_2:

$$\Omega = \Omega_1 \uplus \Omega_2 \equiv (\Omega = \Omega_1 \cup \Omega_2) \wedge (\Omega_1 \cap \Omega_2 = \{\})$$

$$\text{ET-CALL}\ \frac{E.P(\pi, n) = \forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \quad (E, \Omega_i) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}, \overline{\beta'}/\overline{\beta}]}{(E, \overline{\beta'} \uplus \biguplus_{i=1}^{n} \Omega_i) \overset{\text{goal}}{\vdash} \pi(t_1, \ldots, t_n)}$$

Figure 4.7: Typing rule for calls in existential type system

The typing rule for calls is found in figure 4.7.

The call rule ET-CALL, in addition to instantiating the called predicate's universally quantified type variables with types from the caller's context, maps the existentially quantified type variables from the called predicate onto type variables ($\overline{\beta'}$) which are recorded as being produced by this call. (The sequence of type variables $\overline{\beta'}$ are treated beneath the line as if they are a set. We thus require that the elements of this vector be pairwise distinct). The set of existentially quantified type variables produced by the call also includes any that are introduced by the arguments to the call. For example, given

$$\begin{aligned} P_0 = &\quad \{(\texttt{deconstruct\_to\_int}, 3) \mapsto \exists \texttt{T}.\mathbf{pred}(\texttt{to\_int}, \texttt{T}, \mathbf{pred}(\texttt{T}, \texttt{int}))\} \\ F_0 = &\quad \{(\texttt{f}, 2) \mapsto \exists \texttt{T}.\langle \texttt{T}, \mathbf{pred}(\texttt{T}, \texttt{int})\rangle \longrightarrow \texttt{to\_int}\} \\ V_0 = &\quad \{\texttt{X} \mapsto \texttt{to\_int}, \texttt{V} \mapsto \texttt{T}, \texttt{P} \mapsto \mathbf{pred}(\texttt{T}, \texttt{int})\} \\ \Omega_0 = &\quad \{\texttt{T}\} \end{aligned}$$

then with $E_0 = \{P_0, F_0, V_0\}$, we can make the following judgement:

$$(E_0, \Omega_0) \overset{\text{goal}}{\vdash} \texttt{deconstruct\_to\_int}(\texttt{X}, \texttt{V}, \texttt{P})$$

Note that the type variable $\texttt{T}$ appears in Ω_0 for this judgement, which means that $\texttt{T}$ is recorded as having been produced by this call.

The typing rules for goals other than calls are found in figure 4.8.

The higher order call rule ET-HO-CALL is simpler; higher order expressions do not take polymorphic types, and we do not allow the higher order goal to produce existential types. For example, with E_0 as defined above, the following judgement is valid:

$$(E_0, \{\}) \overset{\text{goal}}{\vdash} \texttt{P}(\texttt{V}, 7)$$

ET-UNIFY ensures that the sets of existentially quantified type variables on the two sides of a unification are disjoint.

The rest of the rules are fairly straightforward. For conjunctions, ET-CONJ requires that the set of existentially quantified variables produced in each conjunct be disjoint. For example, given the above two judgements, the following is valid:

$$(E_0, \Omega_0) \overset{\text{goal}}{\vdash} \texttt{deconstruct\_to\_int}(\texttt{X}, \texttt{V}, \texttt{P}) \wedge \texttt{P}(\texttt{V}, 7)$$

The type variable $\texttt{T}$ is produced in the first conjunct, and consumed in the second. We would, however, not be able to call `deconstruct_to_int` twice and assume that it returns the same existential type both times — the requirement that the set of existentially quantified variables produced in the two branches

$$\text{ET-HO-CALL}\ \frac{(E,\Omega_i) \overset{\text{val}}{\vdash} t_i : \tau_i \qquad (E,\{\}) \overset{\text{val}}{\vdash} t : \mathbf{pred}(\tau_1,\ldots,\tau_n)}{(E, \biguplus_{i=1}^{n} \Omega_i) \overset{\text{goal}}{\vdash} t\langle t_1,\ldots,t_n\rangle}$$

$$\text{ET-UNIFY}\ \frac{(E,\Omega_1) \overset{\text{val}}{\vdash} t_1 : \tau \qquad (E,\Omega_2) \overset{\text{val}}{\vdash} t_2 : \tau}{(E,\Omega_1 \uplus \Omega_2) \overset{\text{goal}}{\vdash} t_1 = t_2}$$

$$\text{ET-TRUE}\ \frac{}{(E,\{\}) \overset{\text{goal}}{\vdash} \texttt{true}}$$

$$\text{ET-SOME}\ \frac{V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \qquad (E \oplus V,\Omega) \overset{\text{goal}}{\vdash} G}{(E,\Omega) \overset{\text{goal}}{\vdash} \exists v_1 \ldots v_n.G}$$

$$\text{ET-NOT}\ \frac{(E,\Omega) \overset{\text{goal}}{\vdash} G}{(E,\Omega) \overset{\text{goal}}{\vdash} \neg G}$$

$$\text{ET-CONJ}\ \frac{(E,\Omega_1) \overset{\text{goal}}{\vdash} G_1 \qquad (E,\Omega_2) \overset{\text{goal}}{\vdash} G_2}{(E,\Omega_1 \uplus \Omega_2) \overset{\text{goal}}{\vdash} G_1 \wedge G_2}$$

$$\text{ET-DISJ}\ \frac{(E,\{\}) \overset{\text{goal}}{\vdash} G_1 \qquad (E,\{\}) \overset{\text{goal}}{\vdash} G_2}{(E,\{\}) \overset{\text{goal}}{\vdash} G_1 \vee G_2}$$

Figure 4.8: Typing rules for goals in existential type system

be disjoint stops us from making such a judgement.

For disjunctions, ET-DISJ requires that neither disjunct produce any existentially quantified type variables. This ensures that the set of existentially quantified type variables be the same regardless of which disjunct is followed during the execution of a program. This rule is actually overly conservative.

A less conservative version might allow the disjuncts to produce existentially quantified type variables, as long as exactly the same set is produced by each.

$$\text{ET-DISJ'}\frac{(E,\Omega) \overset{\text{goal}}{\vdash} G_1 \qquad (E,\Omega) \overset{\text{goal}}{\vdash} G_2}{(E,\Omega) \overset{\text{goal}}{\vdash} G_1 \vee G_2}$$

However, the software engineering benefits of doing so are not clear. While there are some situations in which this may be desirable, there are other situations in which this would be likely to propagate the reporting of type errors to further up the call tree. Furthermore, it is not clear how to implement this rule efficiently in a type checker for a practical programming language. This remains a topic for future work.

$$\text{ET-EXIST-DEL}\frac{(E,\Omega) \overset{\text{goal}}{\vdash} G \qquad \beta \text{ does not appear free in } E}{(E,\Omega \setminus \{\beta\}) \overset{\text{goal}}{\vdash} G}$$

$$\text{ET-EXIST-ADD}\frac{(E[\tau/\beta],\Omega) \overset{\text{goal}}{\vdash} G}{(E,\Omega \uplus \{\beta\}) \overset{\text{goal}}{\vdash} G}$$

Figure 4.9: Typing rules for existential type elimination and introduction in existential type system

The rules in figure 4.9 provide a means for adding and deleting type variables from the set of type variables produced by a goal.

ET-EXIST-DEL allows a type variable to be removed from Ω if that type variable does not appear in the types of any variable in E. This can be the case for existentially quantified type variables which are produced inside a goal, and are only consumed locally within that goal. For example, given

$$\begin{aligned}
P_1 &= \{(\texttt{deconstruct_to_int}, 3) \mapsto \exists\texttt{T}.\mathbf{pred}(\texttt{to_int}, \texttt{T}, \mathbf{pred}(\texttt{T}, \texttt{int}))\} \\
F_1 &= \{(\texttt{f}, 2) \mapsto \exists\texttt{T}.\langle\texttt{T}, \mathbf{pred}(\texttt{T}, \texttt{int})\rangle \longrightarrow \texttt{to_int}\} \\
V_1 &= \{\texttt{V} \mapsto \texttt{T}, \texttt{P} \mapsto \mathbf{pred}(\texttt{T}, \texttt{int})\} \\
V_2 &= \{\texttt{X} \mapsto \texttt{to_int}\} \\
\Omega_1 &= \{\texttt{T}\}
\end{aligned}$$

then with $E_1 = \{P_1, F_1, \{\}\}$, we can make the following judgements:

$$
\begin{array}{l}
(1)\ (E_1 \oplus V_2 \oplus V_1, \Omega_1) \overset{\text{goal}}{\vdash} \texttt{deconstruct\_to\_int(X, V, P)} \wedge \texttt{P(V, 7)} \\
(2)\ (E_1 \oplus V_2, \Omega_1) \overset{\text{goal}}{\vdash} \exists \texttt{V, P.deconstruct\_to\_int(X, V, P)} \wedge \texttt{P(V, 7)} \\
(3)\ (E_1 \oplus V_2, \{\}) \overset{\text{goal}}{\vdash} \exists \texttt{V, P.deconstruct\_to\_int(X, V, P)} \wedge \texttt{P(V, 7)}
\end{array}
$$

(1) comes from our previous example. (2) follows from (1) by ET-SOME. (3) follows from (2) by ET-EXIST-DEL.

ET-EXIST-ADD allows any type to be replaced by an existentially quantified type variable. For example, given

$$
\begin{array}{rl}
V_1 = & \{\texttt{V} \mapsto \texttt{int}\} \\
V_2 = & \{\texttt{V} \mapsto \texttt{T}\}
\end{array}
$$

the following are valid judgements (regardless of E):

$$
\begin{array}{c}
(E \oplus V_1, \{\}) \overset{\text{goal}}{\vdash} \texttt{V} = \texttt{1} \\
(E \oplus V_2, \{\texttt{T}\}) \overset{\text{goal}}{\vdash} \texttt{V} = \texttt{1}
\end{array}
$$

(The second follows from the first by ET-EXIST-ADD). The concrete type `int` assigned to `V` has been replaced by an existentially quantified type. However, our type inference/checking algorithm only introduces existential types when necessary, or when an existential type is declared by the user.

The typing rules for values are found in figure 4.10.

We now have two rules for terms. The first, ET-TERM, deals with terms without a '`new`' annotation. For terms with existentially quantified components, this corresponds to a deconstruction. In a similar manner to the `ET-CALL` rule, this rule allows the function symbol's universally quantified type variables to be instantiated with types from the caller's context, and binds the existentially quantified type variables to variables which are recorded in Ω as being produced by this term. For example, the following is a valid judgement:

$$(E_0, \Omega_0) \overset{\text{val}}{\vdash} f(V, P)$$

Because this term appears as part of a deconstruction, the type variable `T` is produced by this term, so it appears in Ω.

The second term rule, ET-NEW-TERM, deals with terms with a '`new`' annotation. These correspond to terms being used as a construction. As such, the function symbol's existentially quantified type variables are allowed to be bound

$$\text{ET-TERM}\frac{E.F(f,n) = \forall\overline{\alpha}.\exists\overline{\beta}.\langle\tau_1,\ldots,\tau_n\rangle \longrightarrow \tau \qquad (E,\Omega_i) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}]}{(E,\overline{\beta'} \uplus \biguplus_{i=1}^{n} \Omega_i) \overset{\text{val}}{\vdash} f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}]}$$

$$\text{ET-NEW-TERM}\frac{E.F(f,n) = \forall\overline{\alpha}.\exists\overline{\beta}.\langle\tau_1,\ldots,\tau_n\rangle \longrightarrow \tau \qquad (E,\{\}) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha},\overline{\tau''}/\overline{\beta}]}{(E,\{\}) \overset{\text{val}}{\vdash} \texttt{new}\ f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha},\overline{\tau''}/\overline{\beta}]}$$

$$\text{ET-VAR}\frac{E.V(v) = \tau}{(E,\{\}) \overset{\text{val}}{\vdash} v : \tau}$$

$$\text{ET-HO-CONST}\frac{V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \qquad (E,\{\}) \oplus V \overset{\text{goal}}{\vdash} G}{(E,\{\}) \overset{\text{val}}{\vdash} \lambda v_1 \ldots v_n.G : \mathbf{pred}(\tau_1,\ldots \tau_n)}$$

Figure 4.10: Typing rules for values in existential type system

to types — they are treated exactly the same as the universally quantified type variables. For example, the following is a valid judgement:

$$(E_0,\{\}) \overset{\text{val}}{\vdash} \texttt{new}\ f(V,P)$$

Because this term appears as part of a construction, the type variable `T` is consumed by this term so the set of produced type variables for this judgement is empty.

Note that ET-NEW-TERM requires that none of the arguments to the term in question produce any type variables. This is not strictly necessary to ensure soundness. However, even if we permitted the arguments to produce type variables, and recorded the disjoint union of these variables as the set of variables produced by the whole term, it would not make any difference to the set of programs accepted by our type system. Given that this rule checks a term construction, none of the arguments can be part of a deconstruction. This effectively constrains the arguments to produce no type variables.

The ET-HO-CONST rule is quite simple. We simply insist that the higher order terms as a whole produce no type variables. Note that it is possible for a higher order goal to produce and consume existentially quantified type variables locally, as long those type variables are not produced by the higher order goal itself.

$$\text{ET-PRED}\frac{\begin{array}{l} E.P(\pi, n) = \forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \\ V = \{v_1 \mapsto \tau_1, \ldots v_n \mapsto \tau_n\} \\ (E \oplus V, \overline{\beta}) \overset{\text{val}}{\vdash} \lambda v_1 \ldots v_n.G \end{array}}{E \overset{\text{pred}}{\vdash} \pi = \lambda v_1 \ldots v_n.G}$$

Figure 4.11: Typing rules for predicate definitions in existential type system

The ET-PRED rule derives a polymorphic type for a predicate. The existentially quantified type variables in the predicate's type are any that are produced by the predicate's body that appear in the predicate's type. (Any extra type variables produced by the predicate's body that do not appear in the predicate's type must be explicitly eliminated using ET-EXIST-DEL.)

For example, given E_1 as previously defined and with

$$V = \{\mathtt{V1} \mapsto \mathtt{to\_int}, \mathtt{V2} \mapsto \mathtt{T}, \mathtt{V3} \mapsto \mathbf{pred}(\mathtt{T}, \mathtt{int})\}$$

the following are judgements involved in the typing of `deconstruct_to_int/3`:

$$\begin{array}{rl} (E_1 \oplus V, \{\mathtt{T}\}) \overset{\text{goal}}{\vdash} & \mathtt{V1} = \mathtt{f}(\mathtt{V2}, \mathtt{V3}) \\ E_1 \overset{\text{pred}}{\vdash} & \mathtt{deconstruct\_to\_int} = \lambda \mathtt{V1}, \mathtt{V2}, \mathtt{V3}.\mathtt{V1} = \mathtt{f}(\mathtt{V2}, \mathtt{V3}) \end{array}$$

The user can introduce new type constructors using type declarations of the form:

$$\textsf{Type declaration} \qquad \forall\overline{\alpha}.\chi(\alpha) \longrightarrow \langle \exists\overline{\beta_1}.f_1(\overline{\tau_1}); \ldots; \exists\overline{\beta_n}.f_n(\overline{\tau_n}) \rangle$$

$$\text{ET-TYPE-DECL}\frac{\begin{array}{l} \mathbf{tvars}(\overline{\tau_i}) \subseteq \overline{\alpha} \cup \overline{\beta_i} \\ F = \{(f_i, |\overline{\tau_i}|) \mapsto \forall\overline{\alpha}.\exists\overline{\beta_i}.\overline{\tau_i} \longrightarrow \chi(\overline{\alpha})\} \end{array}}{E \overset{\text{decl}}{\vdash} \forall\overline{\alpha}.\chi(\overline{\alpha}) \longrightarrow \langle \exists\overline{\beta_1}.f_i(\overline{\tau_1}); \ldots; \exists\overline{\beta_n}.f_i(\overline{\tau_n}) \rangle \rhd E \oplus F}$$

Figure 4.12: Typing rule for type declarations in existential type system

The rule for type declarations is given in figure 4.12. The only difference between this rule and the one defined in chapter 2 is that this rule handles any existentially quantified type variables for each function symbol.

The rule for predicate declarations, ET-PRED-DECL, is the same as the MO-PRED-DECL rule given in chapter 2, and is not repeated here.

4.4 Type classes and existential types

In the examples of using existential types presented earlier in this chapter, we frequently packaged a value of unknown type along with higher order goals to manipulate this value. If we do not package any higher order goals with the value, the packaged value is essentially useless to us[2].

We now present a type system which combines the type class and existential type mechanisms, allowing the programmer to place type class constraints upon existentially quantified type variables. As a result, the programmer is able to construct heterogeneous collections of values whose type is constrained to be in some type class. Rather than needing to package the value with higher order goals to manipulate that value, the value can be directly manipulated by the type class' methods. Similarly, an existentially typed output from a predicate can be type class constrained so that the abstract value can be manipulated by type class methods.

Type class constraints on existentially quantified type variables are attached to a type using the '=>' operator.

For example, rather than using an explicit higher order goal, our `to_int` example could be written thus:

```
:- typeclass to_int(T) where [
    pred to_int(T, int)
].
:- instance to_int(int) where [
    to_int(N, N)
].
:- instance to_int(string) where [
    to_int(S, N) :- string__length(S, N)
].

:- type to_int ---> some [T] f(T) => to_int(T).
```

[2]In Mercury, such a type is not actually useless. Dynamic typing allows the programmer to attempt to cast a value of unknown type into any known type and then, if the cast succeeds, to perform any operations on that value that are appropriate for the known type. Mercury's standard library also provides a set of routines which deconstruct values of arbitrary types into a representation of their constituent function symbol and arguments. These routines are used, amongst other things, to implement Mercury's generic I/O routines, so values of unknown type can always be printed out.

Note that type classes, predicates, and types all have their own name spaces, so the use of the identifier `to_int` in the three contexts above is quite acceptable. Using these definitions, we can construct a heterogeneous collection:

```
    ...
    Xs = ['new f'(42), 'new f'("hello")]
    ...
```

We could also write a predicate which deconstructs a `to_int`:

```
:- some [T] pred deconstruct_to_int(to_int, T) => to_int(T).
deconstruct_to_int(f(V), V).
```

The corresponding construction predicate is:

```
:- pred construct_to_int(to_int, T) <= to_int(T).
construct_to_int('new f'(V), V).
```

Together, existential types and type classes allow the programmer to apply many object-oriented programming styles. The following example, adapted from [52], gives an example of this. We make use of the "`x_y_point`" type class defined in chapter 3.

First, we can define a type which holds a value whose type is a member of the `x_y_point` type class:

```
:- type x_y_point ---> some [T] x_y_point(T) => x_y_point(T).
```

We can then make this type itself an instance of `x_y_point`:

```
:- instance x_y_point(x_y_point) where [
    (get_x(x_y_point(P), X) :- get_x(P, X)),
    (get_y(x_y_point(P), Y) :- get_y(P, Y)),
    (set_x(x_y_point(P0), X, 'new x_y_point'(P)) :-
                    set_x(P0, X, P)),
    (set_y(x_y_point(P0), Y, 'new x_y_point'(P)) :-
                    set_y(P0, Y, P))
].
```

We can then define the following type, which stores an `x_y_point` along with a colour to associate with that point:

```
:- type coloured_point ---> coloured_point(x_y_point, colour).
```

This type can be made an instance of both `x_y_point` and `coloured_point` from chapter 3:

```
:- instance x_y_point(coloured_point) where [
    (get_x(coloured_point(P, _C), X) :- get_x(P, X)),
    (get_y(coloured_point(P, _C), Y) :- get_y(P, Y)),
    (set_x(coloured_point(P0), X, coloured_point(P)) :-
                    set_x(P0, X, P)),
    (set_y(coloured_point(P0), Y, coloured_point(P)) :-
                    set_y(P0, Y, P))
].
:- instance coloured_point(coloured_point) where [
    get_colour(coloured_point(_P, C), C)
].
```

For every instance of the `x_y_point` type class, we automatically have a coloured version. In terms of object oriented programming, this is a form of *implementation inheritance* through *delegation*. We discuss the relationship between existential typing, type classes and object oriented programming in depth in chapter 5.

4.4.1 Type correctness

Type correctness in a system with type classes and existential types is a fairly straightforward combination of the conditions for type correctness in both systems.

The most important concept is that constraints on existentially quantified type variables (the constraints that are attached to a type with '`=>`' rather than '`<=`') are constraints that are guaranteed to be satisfied. This is the dual of the situation with '`<=`' constraints, which are constraints that are yet to be satisfied.

When deconstructing a value with a constrained, existentially typed component, the constraints upon that existential type are guaranteed to be satisfied because those constraints must be satisfied at the construction of that value.

When calling a predicate, '`<=`' constraints must be satisfied by the caller. Correspondingly, the caller can assume that '`=>`' constraints are satisfied, as they are satisfied by the callee.

For example, in our `deconstruct_to_int` example above, the deconstruction of the `to_int` value, gives us a `to_int(T)` constraint that is guaranteed to be satisfied. Thus, the constraint "`=> to_int(T)`" appears on the predicate's type signature — it is satisfied by the deconstruction, so it is satisfied for the predicate as a whole.

Similarly, in `construct_to_int`, the construction of the `to_int` value, requires us to satisfy the `to_int(T)` constraint. That guarantee is propagated to the type of the enclosing predicate, where it appears as a '<=' constraint.

4.4.2 Transformation

Just as a '<=' constraint corresponds to a constraint satisfied by the caller and a '=>' constraint corresponds to a constraint satisfied by the callee, the transformation for a '<=' constraint results in a dictionary that is passed *in* by the caller, while the transformation for a '=>' constraint results in a dictionary that is passed *out* by the callee.

Similarly, when constructing a value with a constrained, existentially typed component, we store a dictionary corresponding to that constraint with the other arguments of the function symbol. When deconstructing such a value, the dictionary corresponding to the satisfied constraint is the one stored with the other arguments when the term was constructed.

For example, our `deconstruct_to_int` predicate is transformed into :

```
deconstruct_to_int(D, X, V) :- X = f(D, V).
```

Here, the value `X` is deconstructed into the existentially typed value `V` and the dictionary `D`. Both the value and the dictionary are passed as outputs from the predicate.

The corresponding construction predicate is transformed into:

```
construct_to_int(D, X, V) :- X = 'new f'(D, V).
```

This time the value `X` is constructed from the value `V` and the dictionary `D` which are both passed as inputs to the predicate.

4.5 Typing and transformation rules

We now present a set of typing rules which define type correctness in our system which combines type classes and existential types.

4.5.1 Type language

Figure 4.13 defines the type language of our combined system. A Qualified type may existentially or universally quantify type variables, and has two sets of type class constraints: constraints that must be satisfied by the caller, and constraints that are satisfied by the callee. A Function symbol type may also contain a set of class constraints.

Our abstract syntax is simply the extended one defined in section 4.3.

Type variable	α		
Type constructor	χ		
Class name	κ		
Simple type	τ	$::=$	α
		$\mid$	$\chi\overline{\tau}$
		$\mid$	$\mathbf{pred}(\overline{\tau})$
Context	θ	$\rightarrow$	$\langle \kappa_1\overline{\alpha}_1, \ldots, \kappa_n\overline{\alpha}_n \rangle$
Qualified type	σ	$\rightarrow$	$\forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\overline{\tau}) \Leftarrow \theta \Rightarrow \theta$
Function symbol type	φ	$::=$	$\forall\overline{\alpha}.\exists\overline{\beta}.\overline{\tau} \Rightarrow \theta \longrightarrow \chi\overline{\alpha}$

Figure 4.13: Syntax of types in the source language

Our typing rules use judgements of the form:

$$(E, \Omega, \Theta) \vdash X \rightsquigarrow X'$$

to represent that X is well-typed with respect to the environment E, which is the same as the environment we used for our type class system (as defined in chapter 3), Ω is the set of existentially quantified variables that were produced by X, and Θ is the set of type class constraints whose satisfaction is guaranteed inside X. As in chapter 3, X' denotes the transformation of X.

- $(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \rightsquigarrow G'$ denotes that G is a well typed goal with respect to E, produces the type variable set Ω and the type class constraint set Θ, and is transformed into G'.

- $(E, \Omega, \Theta) \overset{\text{val}}{\vdash} t : \tau \rightsquigarrow t'$ denotes that t is a well typed term of type τ with respect to E, produces the type variable set Ω and the type class constraint set Θ, and is transformed into t'.

- $E \overset{\text{pred}}{\vdash} \pi = \lambda\overline{v}.G \rightsquigarrow \pi' = \lambda\overline{v'}.G'$ denotes that $\pi = \lambda\overline{v}.G$ is a well typed predicate definition with respect to E, and is transformed into $\pi' = \lambda\overline{v'}.G'$.

- $E \overset{\text{decl}}{\vdash} X \triangleright E' \rightsquigarrow P$ denotes that X is a well formed declaration with respect to E, and yields the new environment E' which takes the declaration into account, and P is a set of predicates which appear in the transformed program.

Throughout our explanation of these typing rules, we will make use of the following environment:

$$
\begin{aligned}
P_0 = \;& \{ \\
& \qquad (\texttt{to\_int}, 2) \mapsto \forall \texttt{T}.\mathbf{pred}(\texttt{T}, \texttt{int}) \Leftarrow \texttt{to\_int}(\texttt{T}), \\
& \qquad (\texttt{deconstruct\_to\_int}, 2) \mapsto \exists \texttt{T}.\mathbf{pred}(\texttt{to\_int}, \texttt{T}) \Rightarrow \texttt{to\_int}(\texttt{T}), \\
& \qquad (\texttt{construct\_to\_int}, 2) \mapsto \forall \texttt{T}.\mathbf{pred}(\texttt{to\_int}, \texttt{T}) \Leftarrow \texttt{to\_int}(\texttt{T}) \\
& \} \\
F_0 = \;& \{ \\
& \qquad ('\texttt{f}', 1) \mapsto \exists \texttt{T}.\langle \texttt{T} \rangle \Rightarrow \texttt{to\_int}(\texttt{T}) \longrightarrow \texttt{to\_int} \\
& \} \\
I_0 = \;& \{ \\
& \qquad \texttt{to\_int}(\texttt{int}) \mapsto dt_{to\_int\_int}, \\
& \qquad \texttt{to\_int}(\texttt{string}) \mapsto dt_{to\_int\_string} \\
& \} \\
C_0 = \;& \{ \\
& \qquad (\texttt{to\_int}, 1) \mapsto \forall \texttt{T}.\{(\texttt{to\_int}, 2) \mapsto \mathbf{pred}(\texttt{T}, \texttt{int})\} \\
& \} \\
E_0 = \;& (P_0, F_0 \cup F_{\mathsf{builtin}}, \{\}, I_0, C_0, \{\})
\end{aligned}
$$

where $dt_{to\_int\_int}$ and $dt_{to\_int\_string}$, which represent higher order goals for producing dictionaries from the corresponding `instance` declarations, are defined as follows:

$$
\begin{aligned}
dt_{to\_int\_int} &= \lambda \texttt{DVout}.(\texttt{DVout} = \texttt{dictionary}(\lambda \texttt{A}, \texttt{B}.\texttt{A} = \texttt{B})) \\
dt_{to\_int\_string} &= \lambda \texttt{DVout}.(\texttt{DVout} = \texttt{dictionary}(\lambda \texttt{A}, \texttt{B}.\texttt{string\_\_length}(\texttt{A}, \texttt{B})))
\end{aligned}
$$

4.5.2 Goals, values and predicate definitions

The rule for predicate calls is given in figure 4.14. It is essentially a combination of the rule for type classes and the rule for existential types defined earlier. The only addition to the rule is the handling of Θ, the set of constraints produced by the call. For a call, this set is simply the constraints produced by the called predicate (appropriately renamed) plus any constraints produced by deconstructing arguments to the call. The purpose of each line of this rule is:

(1) looks up the signature of the called predicate.

(2) checks the type of each argument of the call.

(3) checks that every "`<=`" constraint from the called predicate is satisfied, and finds the variable corresponding to the dictionary for that constraint.

$$
\text{ET-TC-CALL}\;\frac{\begin{array}{l}(1)\ E.P(\pi, n) = \forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \\ \qquad \Leftarrow \langle\theta_1, \ldots, \theta_k\rangle \Rightarrow \langle\theta'_1, \ldots, \theta'_{k'}\rangle \\ (2)\ (E, \Omega_i, \Theta_i) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha}, \overline{\beta'}/\overline{\beta}] \rightsquigarrow t'_i \\ (3)\ E.LI(\theta_i[\overline{\tau'}/\overline{\alpha}, \overline{\beta'}/\overline{\beta}]) = dv_i \\ (4)\ E.LI(\theta'_i[\overline{\tau'}/\overline{\alpha}, \overline{\beta'}/\overline{\beta}]) = dv'_i \\ (5)\ \Omega = \overline{\beta'} \uplus \biguplus_{i=1}^{n} \Omega_i \\ (6)\ \Theta = \biguplus_{i=1}^{k'} (\theta'_i[\overline{\tau'}/\overline{\alpha}, \overline{\beta'}/\overline{\beta}]) \uplus \biguplus_{i=1}^{n} \Theta_i\end{array}}{\begin{array}{l}(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} \pi(t_1, \ldots, t_n) \\ \qquad \rightsquigarrow \pi_n(dv_1, \ldots, dv_k, dv'_1, \ldots, dv'_{k'}, t'_1, \ldots, t'_n)\end{array}}
$$

Figure 4.14: Typing/transformation rules for calls

(4) finds the variable corresponding to the dictionary for every "`=>`" constraint from type signature of the called predicate.

(5) constructs the set of type variables produced by this call.

(6) constructs the set of type class constraints produced by this call.

For example, given

$$
\begin{aligned}
V &= \{\mathtt{F} \mapsto \mathtt{to\_int}, \mathtt{V} \mapsto \mathtt{T}\} \\
LI &= \{\mathtt{to\_int(T)} \mapsto \mathtt{DV}\} \\
\Omega &= \{\mathtt{T}\} \\
\Theta &= \{\mathtt{to\_int(T)}\}
\end{aligned}
$$

and with E_0 defined as above, the following is a valid judgement:

$$
\begin{aligned}
(E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{goal}}{\vdash} \ & \mathtt{deconstruct\_to\_int(F, V)} \\
& \rightsquigarrow \mathtt{deconstruct\_to\_int(DV, F, V)}
\end{aligned}
$$

The rules for the rest of the goal constructs are given in figure 4.15. These rules are all essentially the same as those for our existential type system without type classes, with the additional complexity of keeping track of any constraints produced.

For example, just as the ET-CONJ rule enforces the invariant that each existentially quantified type variable produced in a conjunction is produced in exactly one of its conjuncts, the ET-TC-CONJ rule enforces the invariant that each type class constraint produced in a conjunction is produced in exactly one of its conjuncts. For example, consider the following conjunction:

$$\texttt{deconstruct\_to\_int}(\texttt{F},\texttt{V}) \wedge \texttt{to\_int}(\texttt{V},\texttt{N})$$

which involves calling `deconstruct_to_int` to obtain a value whose type is existentially quantified but constrained to be in the `to_int` type class, then passing that value to the `to_int` method. Given

$$\begin{array}{ll} V & = \{\texttt{F} \mapsto \texttt{to\_int}, \texttt{V} \mapsto \texttt{T}, \texttt{N} \mapsto \texttt{int}\} \\ LI & = \{\texttt{to\_int}(\texttt{T}) \mapsto \texttt{DV}\} \\ \Omega & = \{\texttt{T}\} \\ \Theta & = \{\texttt{to\_int}(\texttt{T})\} \end{array}$$

then the typing of the conjunction involves the following steps.

$$\begin{array}{rl} (E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{goal}}{\vdash} & \texttt{deconstruct\_to\_int}(\texttt{F},\texttt{V}) \\ & \rightsquigarrow \texttt{deconstruct\_to\_int}(\texttt{DV},\texttt{F},\texttt{V}) \\ (E_0 \oplus LI \oplus V, \{\}, \{\}) \overset{\text{goal}}{\vdash} & \texttt{to\_int}(\texttt{V},\texttt{N}) \\ & \rightsquigarrow \texttt{to\_int}(\texttt{DV},\texttt{V},\texttt{N}) \\ (E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{goal}}{\vdash} & \texttt{deconstruct\_to\_int}(\texttt{F},\texttt{V}) \wedge \texttt{to\_int}(\texttt{V},\texttt{N}) \\ & \rightsquigarrow \texttt{deconstruct\_to\_int}(\texttt{DV},\texttt{F},\texttt{V}) \wedge \texttt{to\_int}(\texttt{DV},\texttt{V},\texttt{N}) \end{array}$$

The rules in figure 4.16 provide a means for adding and deleting type variables and type class constraints from the sets of type variables and type class constraints produced by a goal.

ET-TC-EXIST-DEL and ET-TC-EXIST-ADD play a similar role to the rules ET-EXIST-DEL and ET-EXIST-ADD from section 4.3. They provide a mechanism for deleting or adding a type variable to Ω.

ET-TC-CNSTR-DEL and ET-TC-CNSTR-ADD are concerned with deleting or adding a type class constraints to Θ.

ET-TC-CNSTR-DEL allows any type class constraint to be removed from Θ. The result on the transformed program is that the variable corresponding to the dictionary for that variable becomes existentially quantified across the goal in question.

For example, given E_0, LI, V, Ω and Θ as defined above, we have

$$\begin{array}{rl} (E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{goal}}{\vdash} & \texttt{deconstruct\_to\_int}(\texttt{F},\texttt{V}) \wedge \texttt{to\_int}(\texttt{V},\texttt{N}) \\ & \rightsquigarrow \texttt{deconstruct\_to\_int}(\texttt{DV},\texttt{F},\texttt{V}) \wedge \texttt{to\_int}(\texttt{DV},\texttt{V},\texttt{N}) \end{array}$$

$$\text{ET-TC-HO-CALL}\ \frac{\begin{array}{c}(E, \Omega_i, \Theta_i) \overset{\text{val}}{\vdash} t_i : \tau_i \rightsquigarrow t'_i \\ (E, \{\}, \{\}) \overset{\text{val}}{\vdash} t : \mathbf{pred}(\tau_1, \ldots, \tau_n) \rightsquigarrow t'\end{array}}{(E, \biguplus_{i=1}^{n} \Omega_i, \biguplus_{i=1}^{n} \Theta_i) \overset{\text{goal}}{\vdash} t\langle t_1, \ldots, t_n\rangle \rightsquigarrow t'\langle t'_1, \ldots, t'_n\rangle}$$

$$\text{ET-TC-UNIFY}\ \frac{\begin{array}{c}(E, \Omega_1, \Theta_1) \overset{\text{val}}{\vdash} t_1 : \tau \rightsquigarrow t'_1 \\ (E, \Omega_2, \Theta_2) \overset{\text{val}}{\vdash} t_2 : \tau \rightsquigarrow t'_2\end{array}}{(E, \Omega_1 \uplus \Omega_2, \Theta_1 \uplus \Theta_2) \overset{\text{goal}}{\vdash} t_1 = t_2 \rightsquigarrow t'_1 = t'_2}$$

$$\text{ET-TC-TRUE}\ \frac{}{(E, \{\}, \{\}) \overset{\text{goal}}{\vdash} \texttt{true} \rightsquigarrow \texttt{true}}$$

$$\text{ET-TC-SOME}\ \frac{\begin{array}{c}V = \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \\ (E, \Omega, \Theta) \oplus V \overset{\text{goal}}{\vdash} G \rightsquigarrow G'\end{array}}{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} \exists v_1 \ldots v_n.G \rightsquigarrow \exists v_1 \ldots v_n.G'}$$

$$\text{ET-TC-NOT}\ \frac{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \rightsquigarrow G'}{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} \neg G \rightsquigarrow \neg G'}$$

$$\text{ET-TC-CONJ}\ \frac{\begin{array}{c}(E, \Omega_1, \Theta_1) \overset{\text{goal}}{\vdash} G_1 \rightsquigarrow G'_1 \\ (E, \Omega_2, \Theta_2) \overset{\text{goal}}{\vdash} G_2 \rightsquigarrow G'_2\end{array}}{(E, \Omega_1 \uplus \Omega_2, \Theta_1 \uplus \Theta_2) \overset{\text{goal}}{\vdash} G_1 \wedge G_2 \rightsquigarrow G'_1 \wedge G'_2}$$

$$\text{ET-TC-DISJ}\ \frac{\begin{array}{c}(E, \{\}, \{\}) \overset{\text{goal}}{\vdash} G_1 \rightsquigarrow G'_1 \\ (E, \{\}, \{\}) \overset{\text{goal}}{\vdash} G_2 \rightsquigarrow G'_2\end{array}}{(E, \{\}, \{\}) \overset{\text{goal}}{\vdash} G_1 \vee G_2 \rightsquigarrow G'_1 \vee G'_2}$$

Figure 4.15: Typing/transformation rules for goals

and therefore, by ET-TC-CNSTR-DEL

$$\begin{aligned}(E_0 \oplus LI \oplus V, \Omega, \{\}) \overset{\text{goal}}{\vdash} \quad & \texttt{deconstruct\_to\_int}(\texttt{F}, \texttt{V}) \wedge \texttt{to\_int}(\texttt{V}, \texttt{N}) \\ & \rightsquigarrow \exists \texttt{DV}.\texttt{deconstruct\_to\_int}(\texttt{DV}, \texttt{F}, \texttt{V}) \\ & \qquad \wedge\ \texttt{to\_int}(\texttt{DV}, \texttt{V}, \texttt{N})\end{aligned}$$

$$\text{ET-TC-EXIST-DEL}\ \frac{\begin{array}{l}(E, \Omega \uplus \{\beta\}, \Theta) \overset{\text{goal}}{\vdash} G \leadsto G' \\ \beta \text{ does not appear free in } E\end{array}}{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \leadsto G'}$$

$$\text{ET-TC-EXIST-ADD}\ \frac{(E[\tau/\beta], \Omega, \Theta[\tau/\beta]) \overset{\text{goal}}{\vdash} G \leadsto G'}{(E, \Omega \uplus \{\beta\}, \Theta) \overset{\text{goal}}{\vdash} G \leadsto G'}$$

$$\text{ET-TC-CNSTR-DEL}\ \frac{\begin{array}{l}LI = \{\kappa\overline{\tau} \mapsto dv\} \\ (E \oplus LI, \Omega, \Theta \uplus \kappa\overline{\tau}) \overset{\text{goal}}{\vdash} G \leadsto G'\end{array}}{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \leadsto \exists dv.G'}$$

$$\text{ET-TC-CNSTR-ADD}\ \frac{\begin{array}{l}(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \leadsto G' \\ E \overset{\text{dict}}{\vdash} \kappa\overline{\tau} \leadsto dt \\ E.LI(\kappa\overline{\tau}) = dv\end{array}}{(E, \Omega, \Theta \uplus \theta) \overset{\text{goal}}{\vdash} G \leadsto G' \wedge dt(dv)}$$

Figure 4.16: Typing/transformation rules for existential type and type class constraint elimination and introduction

ET-TC-CNSTR-ADD allows any type class constraint which can be proven to be added to Θ. This is typically used along with ET-TC-EXIST-ADD. For example, given

$$\begin{aligned}
V_1 &= \{\texttt{X} \mapsto \texttt{int}\} \\
V_2 &= \{\texttt{X} \mapsto \texttt{T}\} \\
LI_1 &= \{\texttt{to_int}(\texttt{int}) \mapsto \texttt{DV}\} \\
LI_2 &= \{\texttt{to_int}(\texttt{T}) \mapsto \texttt{DV}\} \\
\Theta_1 &= \{\texttt{to_int}(\texttt{int})\} \\
\Theta_2 &= \{\texttt{to_int}(\texttt{T})\}
\end{aligned}$$

we can make the following judgements:

$$\begin{array}{lrl}
(1) & (E_0 \oplus V_1 \oplus LI_1, \{\}, \{\}) \overset{\text{goal}}{\vdash} & \texttt{X} = \texttt{1} \leadsto \texttt{X} = \texttt{1} \\
(2) & E_0 \overset{\text{goal}}{\vdash} & \texttt{to_int}(\texttt{int}) \leadsto dt_{to_int_int} \\
(3) & (E_0 \oplus V_1 \oplus LI_1, \{\}, \Theta_1) \overset{\text{goal}}{\vdash} & \texttt{X} = \texttt{1} \leadsto \texttt{X} = \texttt{1} \wedge dt_{to_int_int}(\texttt{DV}) \\
(4) & (E_0 \oplus V_2 \oplus LI_2, \{\texttt{T}\}, \Theta_2) \overset{\text{goal}}{\vdash} & \texttt{X} = \texttt{1} \leadsto \texttt{X} = \texttt{1} \wedge dt_{to_int_int}(\texttt{DV})
\end{array}$$

(3) follows from (1) and (2) by ET-TC-CNSTR-ADD, and (4) follows from (3) by ET-TC-EXIST-ADD.

$$\text{ET-TC-TERM}\frac{\begin{array}{l} E.F(f,n) = \forall\overline{\alpha}.\exists\overline{\beta}.\langle\tau_1,\ldots,\tau_n\rangle \Rightarrow \langle\theta_1,\ldots\theta_k\rangle \longrightarrow \tau \\ (E,\Omega_i,\Theta_i) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}] \rightsquigarrow t'_i \\ E.LI(\theta_i[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}]) = dv_i \\ \Omega = \overline{\beta'} \uplus \biguplus_{i=1}^{n} \Omega_i \\ \Theta = \biguplus_{i=1}^{k'}(\theta'_i[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}]) \uplus \biguplus_{i=1}^{n} \Theta_i \end{array}}{\begin{array}{c} (E,\Omega,\Theta) \overset{\text{val}}{\vdash} f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha},\overline{\beta'}/\overline{\beta}] \\ \rightsquigarrow f(dv_1,\ldots,dv_k,t'_1,\ldots,t'_n) \end{array}}$$

$$\text{ET-TC-NEW-TERM}\frac{\begin{array}{l} E.F(f,n) = \forall\overline{\alpha}.\exists\overline{\beta}.\langle\tau_1,\ldots,\tau_n\rangle \Rightarrow \langle\theta_1,\ldots,\theta_k\rangle \longrightarrow \tau' \\ (E,\{\},\{\}) \overset{\text{val}}{\vdash} t_i : \tau_i[\overline{\tau'}/\overline{\alpha},\overline{\tau''}/\overline{\beta}] \rightsquigarrow t'_i \\ E.LI(\theta_i[\overline{\tau'}/\overline{\alpha},\overline{\tau''}/\overline{\beta}]) = dv_i \end{array}}{\begin{array}{c} (E,\{\},\{\}) \overset{\text{val}}{\vdash} \texttt{new}\ f(t_1,\ldots,t_n) : \tau[\overline{\tau'}/\overline{\alpha},\overline{\tau''}/\overline{\beta}] \\ \rightsquigarrow f(dv_1,\ldots,dv_k,t'_1,\ldots,t'_n) \end{array}}$$

$$\text{ET-TC-VAR}\frac{E.V(v) = \tau}{(E,\{\}) \overset{\text{val}}{\vdash} v : \tau}$$

$$\text{ET-TC-HO-CONST}\frac{\begin{array}{c} V = \{v_1 \mapsto \tau_1,\ldots,v_n \mapsto \tau_n\} \\ (E,\{\},\{\}) \oplus V \overset{\text{goal}}{\vdash} G \end{array}}{(E,\{\},\{\}) \overset{\text{val}}{\vdash} \lambda v_1 \ldots v_n.G : \mathbf{pred}(\tau_1,\ldots\tau_n)}$$

Figure 4.17: Typing/transformation rules for values

The rules for values are given in figure 4.17. These rules are essentially a combination of the rules for the type class system and the existential type system. The ET-TC-TERM rule shows that the constraints guaranteed to be satisfied by a deconstructed term is the union of the "=>" constraints from the function symbol's type and any constraints guaranteed to be satisfied by the arguments to the function symbol.

For example, given

$$\begin{array}{ll} LI & = \{\texttt{to\_int(T)} \mapsto \texttt{DV}\} \\ V & = \{\texttt{V} \mapsto \texttt{T}\} \\ \Omega & = \{\texttt{T}\} \\ \Theta & = \{\texttt{to\_int(T)}\} \end{array}$$

then the following is a valid judgement:

$$(E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{val}}{\vdash} \mathtt{f(V)} : \mathtt{to\_int} \leadsto \mathtt{f(DV, V)}$$

Conversely, the ET-TC-NEW-TERM rule requires any constraints on the function symbol (and its arguments) to be satisfied at the construction of a term. Given V and LI as above, then the following is a valid judgement for the construction of a term:

$$(E_0 \oplus LI \oplus V, \{\}, \{\}) \overset{\text{val}}{\vdash} \mathtt{new\ f(V)} : \mathtt{to\_int} \leadsto \mathtt{f(DV, V)}$$

$$\text{ET-TC-PRED}\ \frac{\begin{array}{l} E.P(\pi, n) = \forall\overline{\alpha}.\exists\overline{\beta}.\mathbf{pred}(\tau_1, \ldots, \tau_n) \\ \qquad \Leftarrow \langle\theta_1, \ldots, \theta_k\rangle \Rightarrow \langle\theta'_1, \ldots, \theta'_{k'}\rangle \\ LI = \{\theta_1 \mapsto dv_1, \ldots, \theta_k \mapsto dv_k, \theta'_1 \mapsto dv'_1, \ldots, \theta'_{k'} \mapsto dv'_{k'}\} \\ V = \{v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n\} \\ (E \oplus V \oplus LI, \overline{\beta}, \{\theta'_1, \ldots, \theta'_{k'}\}) \overset{\text{val}}{\vdash} G \leadsto G' \end{array}}{\begin{array}{l} E \overset{\text{pred}}{\vdash} \pi = \lambda v_1 \ldots v_n.G \\ \qquad \leadsto \pi_n = \lambda dv_1, \ldots, dv_k, dv'_1, \ldots dv'_{k'}, v_1, \ldots, v_n.G' \end{array}}$$

Figure 4.18: Typing/transformation rules for predicate definitions

The rule for predicate definitions is given in figure 4.18. This rule is essentially a combination of the rules for type classes and for existential types, with the addition of handling Θ, the set of constraints guaranteed to be satisfied by the predicate body. We set up LI so that it contains an entry for every dictionary passed into the predicate, and an entry for every dictionary produced by the body of the predicate.

For example, consider the typing of `deconstruct_to_int` predicate definition:

```
:- some [T] pred deconstruct_to_int(to_int, T) => to_int(T).
deconstruct_to_int(f(V), V).
```

Now, given

$$\begin{array}{ll} LI & = \{\mathtt{to\_int(T)} \mapsto \mathtt{DV}\} \\ V & = \{\mathtt{F} \mapsto \mathtt{to\_int}, \mathtt{V} \mapsto \mathtt{T}\} \\ \Omega & = \{\mathtt{T}\} \\ \Theta & = \{\mathtt{to\_int(T)}\}) \end{array}$$

then the following are valid judgements:

$$
\begin{array}{rl}
(E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{val}}{\vdash} & \mathtt{f(V) : to\_int} \rightsquigarrow \mathtt{f(DV,V)} \\
(E_0 \oplus LI \oplus V, \{\}, \{\}) \overset{\text{val}}{\vdash} & \mathtt{F : to\_int} \rightsquigarrow \mathtt{F} \\
(E_0 \oplus LI \oplus V, \Omega, \Theta) \overset{\text{goal}}{\vdash} & \mathtt{F = f(V)} \rightsquigarrow \mathtt{F = f(DV,V)} \\
E_0 \overset{\text{pred}}{\vdash} & \mathtt{deconstruct\_to\_int} = \lambda\mathtt{F,V.F = f(V)} \\
& \rightsquigarrow \mathtt{deconstruct\_to\_int_2} = \lambda\mathtt{DV,F,V.F = f(DV,V)}
\end{array}
$$

4.5.3 Dictionaries

The typing rules concerning the satisfaction of type class constraints in our type system are the same as those given for the type class system in chapter 3 with one small exception. Figure 4.19 shows the rule TC-ET-LI-ADD, which is almost exactly the same as TC-LI-ADD from chapter 3, in figure 3.7. The only difference is that TC-ET-LI-ADD allows the goal in question to produce type variables or type class constraints through Ω and Θ.

$$
\text{TC-ET-LI-ADD}\;\frac{\begin{array}{l} LI = \{\kappa\overline{\tau} \mapsto dv\} \\ (E \oplus LI, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \rightsquigarrow G' \\ E \overset{\text{dict}}{\vdash} \kappa\overline{\tau} \rightsquigarrow dt \end{array}}{(E, \Omega, \Theta) \overset{\text{goal}}{\vdash} G \rightsquigarrow \exists dv.(dt(dv) \wedge G')}
$$

Figure 4.19: Typing/transformation rule for dictionary introduction

The rules which handle satisfaction and translation of type class constraints themselves, TC-ET-DICT-I and TC-ET-DICT-REL, are identical to the rules TC-DICT-I and TC-DICT-REL from figure 3.8. We do not repeat them here.

4.5.4 Declarations

The rule for type declarations is given in figure 4.20. It is the same as the one defined earlier in the chapter for the type system with existential types, except that we now allow constraints on a function symbol's type.

The rules to handle `pred`, `typeclass` and `instance` declarations can be taken directly from our previous type systems. ET-TC-PRED-DECL is identical to MO-PRED-DECL from figure 2.8, ET-TC-TYPECLASS is identical to TC-TYPECLASS from figure 3.9 and ET-TC-INSTANCE is identical to TC-INSTANCE from figure 3.10. We do not repeat them here.

$$
\text{ET-TC-TYPE-DECL}\ \frac{\begin{array}{l}\mathbf{tvars}(\overline{\tau_i}) \subseteq \overline{\alpha} \cup \overline{\beta_i} \\ F = \{(f, |\overline{\tau_i}|) \mapsto \forall \overline{\alpha}.\exists \overline{\beta_i}.\overline{\tau_i} \Rightarrow \theta_i \longrightarrow \chi(\overline{\alpha})\}\end{array}}{\begin{array}{l} E \overset{\text{decl}}{\vdash} \forall \overline{\alpha}.\chi(\overline{\alpha}) \longrightarrow \\ \qquad \langle \exists \overline{\beta_1}.f(\overline{\tau_1}) \Rightarrow \theta_1; \ldots; \exists \overline{\beta_n}.f(\overline{\tau_n}) \Rightarrow \theta_n \rangle \\ \quad \triangleright E \oplus F \end{array}}
$$

Figure 4.20: Rule for type declarations

4.6 Existential types in Mercury

Existential typing also has an interesting interaction with Mercury's mode system.

Mercury's mode system may reorder conjuncts based upon data flow dependencies to ensure that the consumer of a value occurs before its producer [98]. Without existentially quantified type variables, the data flow of types in a program is always top down (from caller to callee). However, with the introduction of existential quantification comes the possibility for upwards data flow of types (and thus of dictionaries). This more interesting type data flow can itself cause data dependencies which cause Mercury to reorder conjuncts.

For example, given the following example where we define a class of types for which we have a predicate which will initialise a value of that type.

```
:- typeclass initable(T) where [
    pred init(T)
    mode init(out) is det
].

:- some [T] pred p(list(T)) => initable(T).
:-          mode p(in) is det.

p(Xs) :-
    init(X),
    q(Y),
    Xs = [X,Y].

:- some [T] pred q(T) => initable(T).
:-          mode q(in) is det.
```

```
q(...) :- ...
```

In this example, the body of the predicate `p/1` has a data flow dependency from the dictionary introduced during the type class transformation which means that the body must be reordered to be mode correct. The dictionary for `initable(T)` is produced by `q/1`, but it is needed by the call to `init/1`, so the body must be reordered to put the call to `q/1` before the call to `init/1`. The resulting transformed and reordered code for `p/1` is:

```
p(D, Xs) :-
    q(D, Y),
    init(D, X),
    Xs = [X,Y].
```

Our implementation in the Mercury compiler is able to perform reordering based on types and dictionaries because the type checking and type class transformation phases of compilation happen before the mode checking phase.

4.7 Remarks

We have described two type systems. The first system extends our higher order Mycroft-O'Keefe type system with the ability to existentially quantify type variables on predicate type signatures and on the types of function symbols. The second system extends the first by including our type class system from chapter 3. This type system has been implemented in the Melbourne Mercury compiler, and has been part of the Mercury distribution since version 0.9.

Previous existential type systems for functional programming languages such as [51] were able to syntactically determine whether an occurrence of a functional symbol was being used to construct or to deconstruct a value. As such, the appropriate typing rule could be chosen. However, in logic programming there is no syntactic distinction between construction and deconstruction unifications. Under type systems which do not allow type variables to be existentially quantified, this is not a concern. If there are no existentially quantified type variables involved, the typing of a unification is the same whether it is a construction or a deconstruction. We have, however, shown that the presence of existentially quantified type variables requires the type system to distinguish between constructions and deconstructions.

Our solution to this has been to require the programmer to explicitly annotate construction unifications which involve existentially quantified type variables with the '`new`' keyword. This does not seem to be a great burden in practice, as most constructions do not involve existentially quantified arguments.

Nonetheless, this work leaves open the question of whether the task of distinguishing between construction and deconstruction unifications for the purposes of typing can be automated and integrated into the type checker. One potential solution to this problem would be to use Mercury's overloading mechanism to treat the appearance of each function symbol as both a construction and deconstruction, and then to eliminate those possibilities which are not possible. Finally, mode information could be used to decide any cases which could not be resolved by the overloading mechanism. However, this approach is somewhat undesirable as it would require much of the information built up during type checking to be retained during the mode checking phase. It is not clear that this could be implemented in a practical system. This remains a matter for further investigation.

Chapter 5

Object Oriented Architectures and Logic Programming

Designing a software system is a difficult task. Regardless of the design methodology being used, the designer must somehow decompose the system into appropriate structures (be they objects, modules, procedures or predicates) and define the interfaces of and the relationships between these structures. This task typically involves weighing up many conflicting factors, with the ultimate aim being a design which is robust, and yet is simple both to implement and to maintain. Almost all software systems are subject to change, through either corrective or perfective maintenance [84, 97]. As such the value of a good design lies as much in how easily it allows the system to be adapted to new requirements as it does in how it reduces implementation effort. In addition, it is most desirable that a design be amenable to reuse in the implementation of other software systems. Ultimately, designs should be generic enough to be reusable, and yet specific enough to solve the particular problem being tackled.

The key tool for the software architect is *abstraction*. Abstraction is important for three key reasons:

- It allows the implementor to consider small parts of the software system in isolation, ignoring the details of the rest of the system. In doing so, it reduces the *conceptual load*.

- By forcing access to a subsystem only through a particular, well-defined interface, the programmer is not allowed to use the subsystem in inappropriate ways. With the appropriate use of abstraction, the user of a subsystem cannot break invariants, such as the elements in a list being in sorted order, which the implementation of the subsystem may rely upon. The abstraction provides *fault containment*.

- It provides a degree of *independence amongst subsystems*, meaning that
    - The implementation of a subsystem can be assigned to an individual or group of individuals who don't have detailed knowledge of the rest of the system.
    - The implementation of a subsystem can be changed without affecting users of that subsystem.
    - Completed subsystems can be installed as libraries for use by other software systems.

Object oriented design has been a very successful tool in combating the complexity of the software design process [92]. Object oriented design and object oriented programming languages came into widespread use in software development during the mid 1980s, amidst claims that the technology would solve the software crisis — that through its use both implementation and maintenance effort would be greatly reduced.

In many respects, it has delivered on its promises. Object technology is now firmly entrenched in modern software development practice, and is commonly held to be *the* simplifying approach to design and implementation [5].

But at the same time, object oriented development is not a silver bullet. Indiscriminately following an object oriented development methodology can easily lead to the construction of systems which are prohibitively complex and difficult to maintain. Object oriented design is a technique that can help conquer the complexities of software architecture, but as with any technique, it must be carefully applied.

In this chapter, we examine object oriented design, showing the mechanisms that are available to the designer to define program structures, and to extend and compose these structures. We also attempt to characterise what makes an oriented design good or bad. We examine some of the most popular object oriented programming languages with respect to how they allow the programmer to express object oriented designs. Our motivation in doing so is to evaluate which programming language mechanisms encourage good object oriented architectures.

We then introduce the concept of object oriented programming in a typed higher order logic programming language with type classes and existential types, such as Mercury. We show that type classes correspond to object interfaces, and that existential types allow us to use type classes in many ways that take advantage of object oriented design techniques. We show that, although Mercury is not an object oriented programming language per se, the language does allow the programmer to express many useful object oriented design idioms. We show that many object oriented designs can be quite mechanically translated into Mercury. Furthermore, we show that the object oriented design techniques that Mercury encourages are those that adhere to the principles of design for change.

We believe that this represents a significant advance in modelling techniques for the logic programmer.

5.1 Object oriented architectures

Object oriented programs are composed of *objects*. Objects consist of both data and procedures that operate on that data. The procedures are typically referred to as *methods*[1].

Objects perform a particular operation when they receive a particular request or *message* from a *client*. Such requests are the only way to get an object to perform an action, and are the only way to change the object's internal data.

The definition of an object is usually separated into its *interface*, which specifies the signatures of its methods and attributes, and its *implementation*, which specifies the data and code used to implement its interface. In the object oriented development literature such as [25], a particular interface that an object can be accessed as is referred to as a *type*, whereas the implementation of an object is referred to as its *class*. Note that this is exactly the opposite to the nomenclature we have introduced for typed logic programming in the earlier chapters! We have followed the path taken by the functional programming literature, which has borrowed heavily from the field of type theory, in referring to the form of an object's data (its implementation) as a *type* and the interface through which an object can be manipulated as a *class*, or *type class*. We continue to follow that path in this chapter.

The fundamental concept of object oriented systems is that of interfaces. Objects can only be manipulated through their interfaces; there is no way of getting an object to do anything or to change an object in any way other than going through its interface. Objects provide a means of *encapsulating* the implementation of an object. An object's interface is quite independent of its implementation — objects with the same interface are free to be implemented in different ways.

Object oriented designs are typically depicted diagrammatically, using a notation such as the Unified Modelling Language (UML) [91], now an industry

[1] It is no coincidence that we have also used the name "method" in previous chapters to refer to the predicates and functions in a type class interface.

standard. Figure 5.1 shows an example UML diagram of a simple object with a set of methods and some data implementing the object.

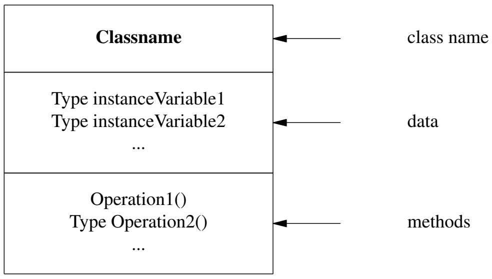

Figure 5.1: UML representation of a simple object

The definition of a class can be given in terms of another class using a mechanism known as inheritance. We say that a *sub-class* inherits from a *super-class* or *parent class.* A sub-class includes all the definitions of data and methods from its parent. An object that is an instance of a sub-class has all of the methods and data from its own definition and from that of its parent. A sub-class definition is free to *override* the definition of the methods of its parent class. Inheritance is represented diagrammatically in UML as shown in figure 5.2.

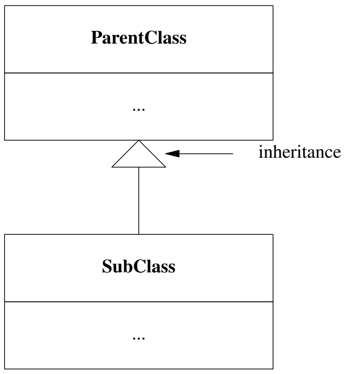

Figure 5.2: UML representation of inheritance

A parent class definition may omit the definitions of some or all of its methods, merely providing their signatures. The definitions of these methods is deferred to the definition of a sub-class. A method without a definition is called an *abstract method.* A class with one or more abstract methods is called an *abstract class.* Classes with no abstract methods are called *concrete classes.* Abstract classes cannot be instantiated into objects, and as a consequence are only useful as parent classes.

Classes whose methods are all abstract and which contain no data are called *pure abstract classes*. Such classes typically serve to simply define an interface for sub-classes. The ability to define object interfaces, in the form of abstract base classes, as entities completely separate from any implementation is an extremely important mechanism in designing systems for change, and is referred to as *interface inheritance*. Similarly, inheritance where the parent class is not a pure abstract class is referred to as *implementation inheritance*.

Abstract classes appear in UML diagrams with their names in an italic font. Similarly, abstract methods are listed in italic font. Figure 5.3 shows a class with an abstract method that is inherited by a sub-class which provides an implementation for that method.

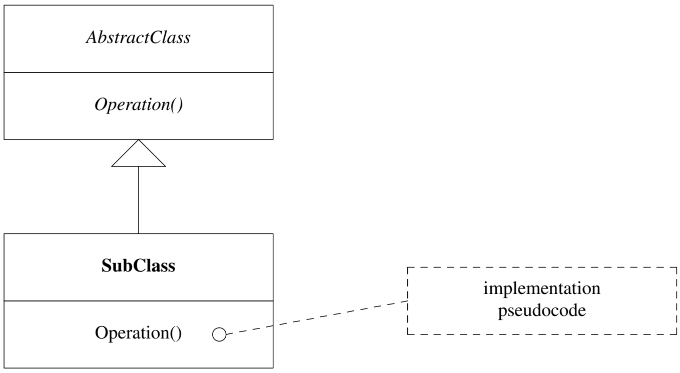

Figure 5.3: UML representation of an abstract class inherited by a concrete class

A client invoking a method upon an object has no way of knowing what the implementation of the object is. It only knows that the object can be accessed through the well defined interface it has been given. In particular, the client does not know whether the object is an instance of the parent class or one of its sub-classes. When a method is invoked upon an object, the actual code that is executed is chosen at run time depending on which particular sub-class an object is an instance of. This mechanism is known as *dynamic binding*, and is an essential part of object oriented development. Dynamic binding means that objects with different implementations but with the same interface may be substituted for one another at run time. This is referred to as *polymorphism* by the object oriented literature. While this is different to the "polymorphism" that functional and logic programmers speak of (by which they generally mean "parametric polymorphism"), there is still a strong connection: both mechanisms allow the programmer to write code which will work for objects with many different kinds of representations. As we will see below, the object oriented concept of polymorphism corresponds more closely to parametric polymorphism with type class constraints, or just "constrained polymorphism."

5.2 Object oriented programming

There are a plethora of programming languages which claim to be "object oriented programming languages". Each of these varies in the way that they support the mechanisms of object oriented design.

C++ [102] and Eiffel [64] use a "class" construct for the definition of class interfaces and implementations. A class declaration can inherit from another class, define new methods and override inherited methods. A C++ class generally has access to all the data of its parent classes. However, data and methods can be declared "private" to hide these details from sub-classes.

The following is a simple example of inheritance in C++. The `person` class is a parent class of `student` — every student is also a person, but students are also assigned a student number.

```
class person {
    public:
        string name;
    ...
}

class student : public person {
    public:
         int student_number;
    ...
}
```

In contrast, Smalltalk [36] programs are untyped. The programmer does not declare the types of variables — instead, a run time check is typically performed to make that sure an object is capable of receiving a particular message (i.e. that a particular method is valid for that object). Smalltalk's inheritance mechanism is purely one of inheriting implementations — there is no way of defining the signature of a method, and thus no way of defining an interface in any real sense of the word. Nonetheless, many would say that Smalltalk is the programming language most faithful to the object oriented paradigm. It is untyped, but everything is an object, even numbers.

5.2.1 Dynamic binding

Dynamic method dispatch or binding is an essential part of any object oriented programming language. Methods for the programmer to achieve this vary from language to language.

In Smalltalk, all dispatch is dynamic. In C++, however, static binding is the default. This is presumably a design decision based on efficiency; if the compiler can statically determine which method is being called, it is bound to be more efficient than looking up the method at run time.

The C++ programmer is able to force dynamic binding using the `virtual` keyword. A method that is declared virtual is dispatched to the appropriate class at run time based on the implementation of the object rather than on the interface of the reference. For example, given the following C++ code:

```
class c {
    public:
        virtual int f() { return 1; }
...
}

class c1 : public c {
    public:
        int f() { return 42; }
...
}

int call_f(c x)
{
    return x->f();
}
```

then if `y` is a object which is an instance of the base class `c`, then the call `call_f(y)` will return `1`, whereas if `y` is an instance of the sub-class `c1`, then the call would return `42`. The particular definition of the `f` method called is chosen at run time, depending on the class of the value passed to `call_f()`. Note that if the definition of `f()` in the class `c` was not annotated with the `virtual` keyword, then every call to `call_f` would return `1`.

Virtual functions can also be made abstract in C++ by assigning them the constant `0`. This is referred to as a *pure virtual* function.

```
class c {
```

```
    public:
        virtual f() = 0;
...
}
```

If a class has any pure virtual methods then it cannot be instantiated into an object. Only sub-classes for which a definition has been provided for each pure virtual method can be instantiated.

An alternative approach is taken in Ada 95 [38], where the programmer declares the type of an object to be "class-wide", using the `'Class` annotation, if dynamic dispatch is to be used for that object. This associates dynamic dispatch with particular references to objects rather than with particular methods or classes. In the following example (taken from [93]), the formal parameter `r` of the `print_appropriate_address` function is declared to be of the *class wide type* `person'Class`.

```
type person is tagged record ...
type student is new person with ...
type teacher is new person with ...

procedure print_address(r:person) is ...
procedure print_address(s:student) is ...
procedure print_address(t:teacher) is ...

procedure print_appropriate_address(r:person'Class) is
begin
    print_address(r);
    -- the appropriate overloaded version is called
    -- at run time depending on the type of r
end

procedure print_appropriate_address_2(r:person) is
begin
    print_address(r);
    -- the version of print_address from the base class
    -- ``person'' is always called
end
```

The rationale behind this mechanism is that it gives the programmer greater control over whether static or dynamic binding is used. In C++, all references are essentially "class-wide", since a `virtual` method always uses dynamic dispatch.

In Java [1, 22], dynamic dispatch is the default. However, methods annotated with the `final` keyword in the parent class definition cannot be overriden by the sub-class, so these methods may be implemented using static dispatch. The use of interface inheritance is seen as being so important that pure abstract base classes are defined using the `interface` keyword.

In C♯ [19, 109], methods to use dynamic dispatch are specified using the `virtual` prefix, and classes overriding implementation of such methods must use the keyword `overriding` at their definition. A pure virtual method is prefixed with the `abstract` keyword, and the same keyword is used to designate that a class is a pure abstract one.

5.2.2 Generics

Generics provide object oriented languages with expressiveness similar to parametric polymorphism in functional and logic programming languages. Ada 95 and Eiffel both provide a generics mechanism. C++'s templates also provide a form of generics. These mechanisms allow class (or package in Ada 95), function and method types to be parametrised by particular types. For example, the following outlines a parameterised class of stacks in C++:

```
template <class T>
class stack {
    private:
        T *stack_elements;
        int cur;
    public:
        void push(T item) { stack_elements[cur++] = item; }
        T pop() { return stack_elements[cur--]; }
        ...
}
```

Using the template mechanism, we are able to declare the argument passed to the `push()` method and the return value of the `pop()` method to be both of type `T`, where `T` is the element type used by this instance of our `stack` class. For example, we could build stacks of type `stack<int>` and `stack<char>`, in which case the methods would handle values of type `int` and `char` respectively. Without using a generics mechanism such as templates, the argument of the `push()` method and the return value of the `pop()` would have to be declared as being the same type for all instances of the `stack` class. As a result, the user would be forced to use more run-time type casts and the compiler would catch fewer errors.

Two extensions of Java, known as Generic Java [4] and Pizza [77], both provide support for generics. Both languages are strict supersets of Java; every valid Java program is a valid Generic Java or Pizza program.

5.2.3 Multiple inheritance

A class that inherits from more than one other class is said to be using *multiple inheritance*.

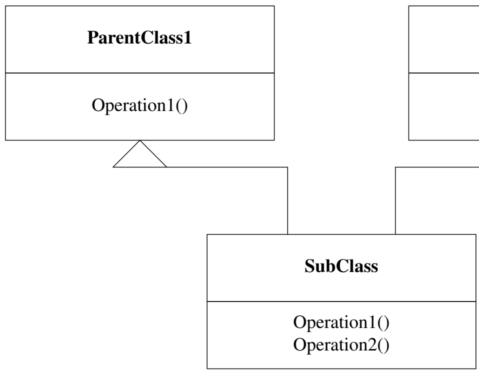

Figure 5.4: UML representation of multiple inheritance

Multiple inheritance is a contentious issue in programming language design. There are two reasons for this. First, multiple inheritance severely complicates the implementation of inheritance. Second (and perhaps more importantly), the semantics of multiple inheritance is not simple. The difficulty lies in the meaning of a class which has more than one parent class with a method with the same signature. For example, given the following C++ classes:

```
class a {
    public:
        void print();
    ...
}

class b {
    public:
        void print();
    ...
```

```
}

class c : public a, public b {
    ...
}
```

If we have a variable `x` of type `c`, what should happen if we execute `x.print()`? The answer is not clear.

In C++, the above class definitions are quite legal. However, a semantic error will be reported at the *use* of the ambiguous method, in this case at the call `x.print()`.

Eiffel, on the other hand, will report a static semantic error when defining a derived class with such a semantic ambiguity.

Yet another approach is taken by CLOS (the Common Lisp Object System); the class definition is accepted, and executing the ambiguous method results in the execution of the version of the method of whichever parent class that appeared first in the sub-class' header. This goes against the "principle of least astonishment" — the user should be able to easily predict what a program will do, and not be surprised by the result.

While multiple inheritance is supported in C++, Eiffel and CLOS, it is disallowed in Java, Simula, Smalltalk, Modula-3, Ada 95 and Oberon.

The C++ programmer is able to explicitly qualify a method call with the name of the parent class from which to select the method being called. Using this mechanism, the programmer can redefine the method for the sub-class in terms of one or more parent classes:

```
void c::print() {
    a::print();
    b::print();
}
```

Shared vs. replicated multiple inheritance

The situation becomes much more complicated when both parents inherit from the same base class themselves. For example, in the example below, the class `c` inherits from both `b1` and `b2`. Both `b1` and `b2` inherit from `a`.

```
class a {
    int data1;
    int data2;
    ...
```

```
}

class b1 : public a { ... }

class b2 : public a { ... }

class c : public b1, public b2 { ... }
```

In this case, whether it would be right for `c` to inherit two copies of the data members of `a`, or only one shared copy depends very much upon the programmer's intentions.

In C++, the default is *replicated inheritance*. In our example above, two copies of `a` would be created. The mechanism for creating a single shared copy of multiple inherited classes is called *shared inheritance*. The C++ programmer can force shared inheritance through the use of the `virtual` keyword. Note that this has a different meaning for the same keyword when applied to method of a class, where it forces dynamic binding to be used for that method.

```
class b1 : public virtual a { ... }

class b2 : public virtual a { ... }
```

Consider the situation where we have a class `list_node` which represents linked-list nodes, as well as classes `student` and `teacher` which both inherit from a base class `person`. The system also maintains separate linked lists of students and teachers, so `student` and `teacher` both inherit from `list_node`.

```
class list_node{ ... }

class person {
    public:
        void change_address(string address) { ... }
        void print_details() { ... }
    ...
}

class student : public person, public list_node { ... }

class teacher : public person, public list_node { ... }
```

Now, imagine that we want to create a new class `student_teacher`, which represents people who are both students and teachers.

```
class student_teacher : public student, public teacher { ... }
```

In this case, we really want to keep only one copy of the `person` data, as changes to this data is likely to be pertinent whether the person is being viewed as a student or a teacher. However, we still want to be able to store our `student_teacher` in *both* the student and teacher linked lists. Hence, we should use replicated inheritance for the `list_node` class, but shared inheritance for `person`. The desired structure is shown in figure 5.5

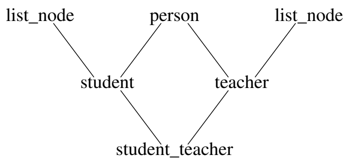

Figure 5.5: Replicated `list_node`, shared `person`

This can be implemented with the following C++ code:

```
class student : public virtual person, public list_node { ... }

class teacher : public virtual person, public list_node { ... }

class student_teacher : public student, public teacher { ... }
```

Eiffel's default is shared inheritance. Replicated inheritance is simulated using a renaming mechanism. This is illustrated in the following example from [93]:

```
class student inherit person ; list_node ...
class teacher inherit person ; list_node ...

class student_teacher
inherit
    student
        rename
            prev as prev_student
            next as next_student
        end;
    teacher
        rename
            prev as prev_teacher
            next as next_teacher
```

```
        end
feature
    ...
end
```

Mixin inheritance

Java supports a limited form of multiple inheritance known as *mixin inheritance*, which is multiple inheritance where at most one parent class is a concrete class, and all others are pure abstract classes.

In Java, a class without any data or implementations of any of its methods can be denoted with the keyword `interface`. A syntactic distinction is made between inheriting an interface and inheriting a class: a class can `implement` any number of interfaces and `extend` zero or one classes. The following example, from [22], shows a class `CenteredRectangle` which inherits the implementation of the `Rectangle` class, and implements an abstract interface called `Centered`.

```
public class CenteredRectangle
             extends Rectangle implements Centered {
    // We inherit all the methods of Rectangle, but must
    // provide implementations of all the Centered methods.
...
}
```

Figure 5.6 shows an example of mixin inheritance.

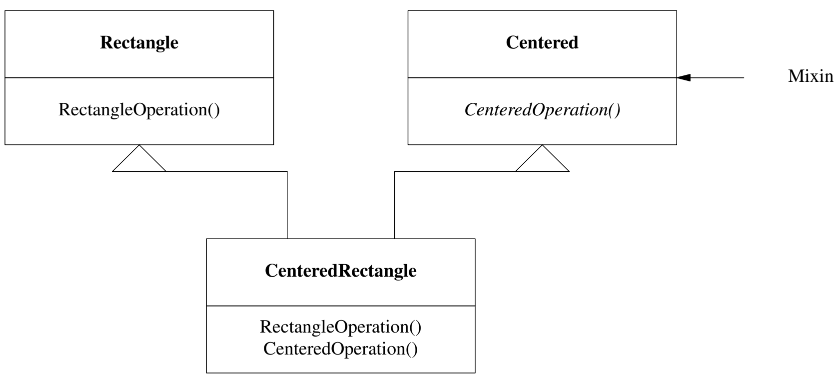

Figure 5.6: UML representation of mixin inheritance

5.3 Designing for change

Despite the best intentions of software designers, software systems typically do not stand up well to the rigours of maintenance. Bug fixes, requirements changes, performance enhancements and new features take their toll; even the best thought out design faces change. Changes are often made quickly, and often by programmers who do not understand the reasoning behind the original design. Even when this is not the case, when changes are made in plenty of time by those with a thorough understanding of the design, designs still have to change in ways that cannot reasonably be foreseen. As such, it is not reasonable for the software architect to blame the failure of a design on change: change is a fact of life in the world of software, and if a design cannot adapt to change, it is the design that is at fault.

When making changes to a system, there is typically more than one way of doing so. Some of these ways maintain the integrity of the design. Others, the "hacks," do not. A design is a poor one when the integrity maintaining methods are all harder than the hacks.

Changes to a design produce new dependencies in subtle ways. These dependencies are the source of problems in software architecture. As designs change and dependencies become more complicated, making more changes becomes ever more difficult. The challenge for the software designer, then, is to produce a design which takes into consideration the kinds of changes that are likely to occur, and allows the software to be changed in these ways without reducing maintainability, and without compromising the original design.

5.3.1 Implementation vs. interface inheritance

Inheritance is a powerful software reuse mechanism. It allows functionality to be extended rapidly by defining a new class in terms of an old one. By re-using the implementation of the parent class, parts of the sub-class' implementation come seemingly for free.

There is, however, a hidden cost in terms of software dependency. Every time the implementation of one class is inherited by another, a dependency is created between the classes. Because the derived class accesses the implementation of the parent class directly rather than using the parent class' interface, the implementations of the two classes become inextricably coupled. Sometimes this dependency is unavoidable, but often it is not.

The ability to define a single interface (abstract class) that is implemented by a number of different classes using interface inheritance is arguably more important than the ability to reuse implementations through implementation inheritance. Interface inheritance allows clients to remain unaware of the implementation of objects, and unaware of other interfaces that an object may satisfy.

5.3.2 Principles of good object oriented design

In this section we examine some well established principles of good object oriented design. Each of these principles encourages designs in which undesirable dependencies do not develop. An excellent summary of these principles can be found in [62]. Each principle is explained in full in the references given in the following sections.

Open closed principle (OCP)

A module should be open for extension but closed for modification. [61]

This principle states that modules should be written so that their functionality can be extended without itself needing to be modified. When the behaviour of the system needs to change, it should be possible to extend modules by adding new code, rather than modifying code in existing, tested modules.

The mechanism for satisfying this principle is abstraction. Consider the example (based on [61]) whereby a routine is responsible for logging on to a server using a modem. There are several different kinds of modems, and dialling a number is done in a different way on each kind of modem, as shown in the following C++ example:

```
struct Modem
{
    enum Type {hayes, courrier, ernie} type;
    // ...
    // other data concerning the modem, including
    // information for each specific kind of modem
    // ...
};

void LogOn(Modem& m, string& phonenum, string& user, string& pw)
{
    if (m.type == Modem::hayes) {
        DialHayes(m, phonenum);
    }
```

```
    else if (m.type == Modem::courrier) {
        DialCourrier(m, phonenum);
    }
    else if (m.type == Modem::ernie) {
        DialErnie(m, phonenum);
    }
    SendUser(m, user);
    SendPassword(m, pw);
}
```

The problem with this solution is that the `LogOn()` function must be changed every time a new kind of modem is added.

Rather than depending on a particular implementation, the `LogOn()` should manipulate the modem structure through an abstract interface. An example of such an architecture is shown in figure 5.7.

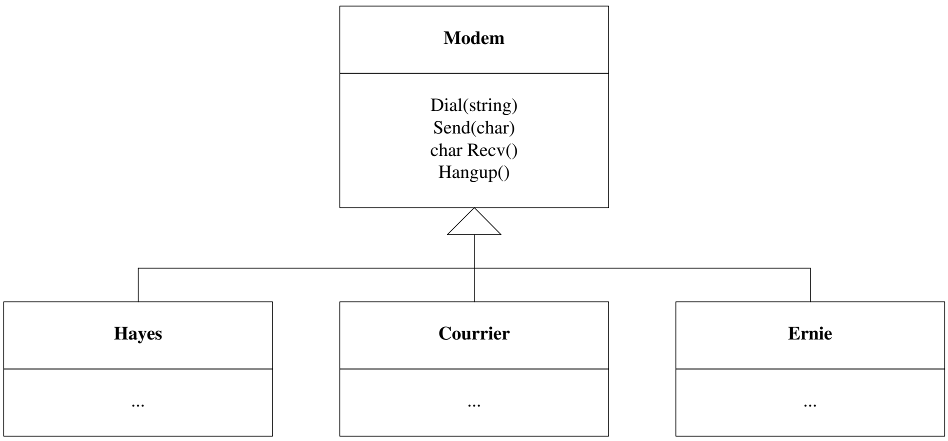

Figure 5.7: Example of applying the Open Closed Principle

The `Logon()` function can then be rewritten as:

```
void LogOn(Modem& m, string& phonenum, string& user, string &pw)
{
    m.Dial(phonenum);
    SendUser(m, user);
    SendPassword(m, pw);
}
```

This definition does not need to change when we add a new modem type.

Liskov substitution principle (LSP)

Subclasses should be substitutable for their base classes. [60]

A sub-class should only inherit from a parent class if every instance of the sub-class is also an instance of the parent class. That is, it should be possible to substitute an instance of the sub-class whenever an instance of the parent class is expected. More specifically, every operation that is valid upon objects from the parent class should be valid upon objects of the sub-class.

For example, a square is a rectangle, but if the class representing rectangles has methods which allow the width and height to be set independently, then allowing a class representing squares to be a sub-class of the rectangle class would be a violation of the LSP.

In terms of preconditions and postconditions, for every method of a parent class, the implementation of that method in the sub-class must never require the strengthening of the precondition or weakening of the postcondition. Making the square class a subclass of the rectangle class violates this because, for example, a method which allows a rectangle's height to be changed would have a postcondition that the width is not changed, but this is not true for squares.

Dependency inversion principle (DIP)

Depend upon abstractions. Do not depend upon concrete entities. [58]

This principle takes its name from the fact that traditional imperative programs have a structure such that the high level modules depend upon the details of the lower level modules. This is inherently flawed. An object oriented design that follows the DIP exhibits very different dependencies. All dependencies are on abstractions. In particular, the low level modules that would have been depended upon by high level modules instead themselves depend upon abstractions. *The dependency is inverted.*

Figure 5.8 shows the kind of undesirable dependency structure that is typical of traditional imperative programs.

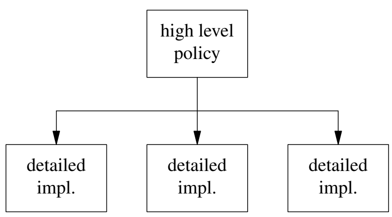

Figure 5.8: Dependency structure of a poorly designed architecture

Figure 5.9 shows the kind of dependency structure that is desirable in an object oriented architecture.

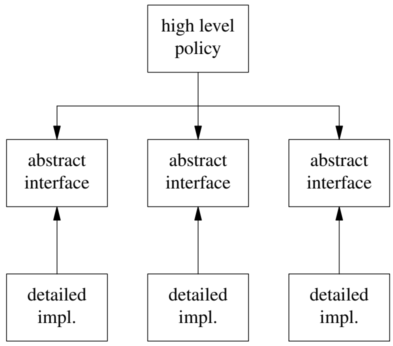

Figure 5.9: Dependency structure of a good object oriented architecture

The importance of this principle is clear: the OCP gives us a clear goal for object oriented architectures; the DIP gives us a mechanism for achieving that goal. Every dependency should be upon an interface (an abstract class). There should be no dependencies upon concrete classes.

By making interfaces explicit, and creating dependencies only on these interfaces rather than on implementations directly, the software designer is able to greatly reduce accidental dependencies.

Interface segregation principle (ISP)

Many client specific interfaces are better than one general purpose interface.[59]

If a class has several different clients which access its functionality in different ways, it is better to create an interface for each different way that the object is accessed than to have all clients use one gargantuan interface. (Of course, this principle is to be taken with a grain of salt — a class with hundreds of different interfaces would be a maintenance nightmare in itself.) This principle is illustrated in figures 5.10 and 5.11. The first shows three clients using a service through a single, bloated interface. The second shows a better design: each client accesses the same service through a cut down, client-specific interface. As such, each client only depends on the part of the object's interface that is important to that client.

The rationale for this principle is also simple: it is best not to force a client to depend upon details of an interface that it does not use.

5.3.3 Inheritance vs. object composition

Implementation inheritance is *white-box* reuse; the internals of the parent class are visible when one inherits the implementation of the parent class to the

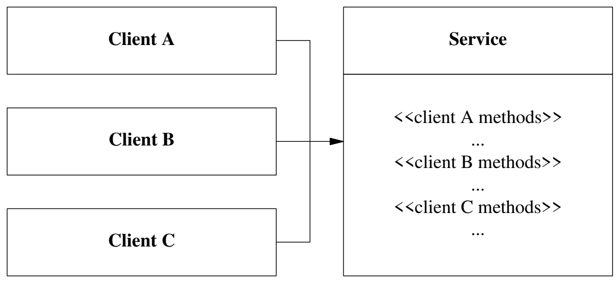

Figure 5.10: Clients interacting with an object through a single monolithic interface

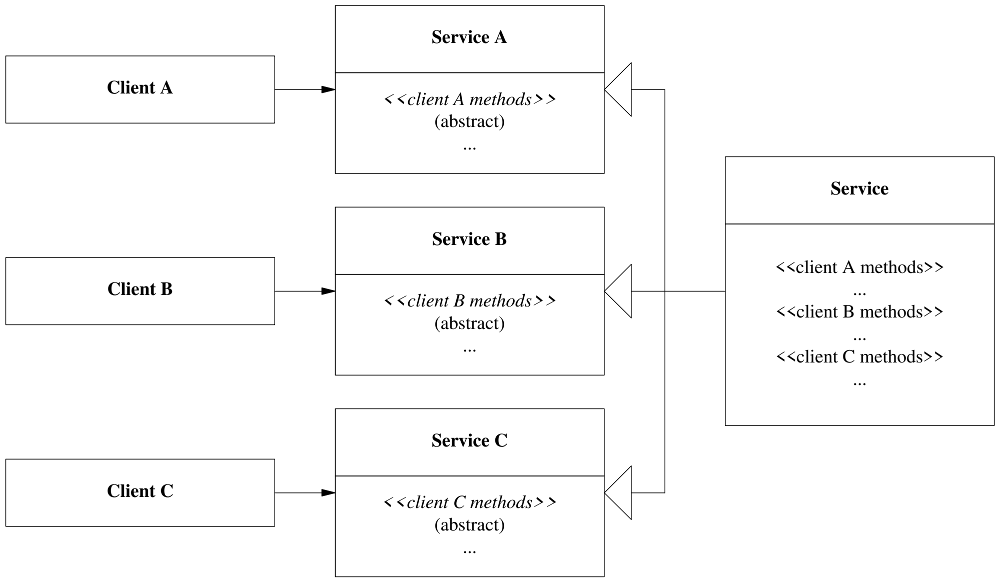

Figure 5.11: Segregated interfaces

sub-class.

Object composition, however, is a *black-box* reuse technique. Object composition is an alternative to implementation inheritance where a new object is produced simply by putting together or composing other objects.

Composition is an inherently dynamic mechanism. Whereas the structures involved in inheritance are all defined at compile time, composition does not occur until objects are actually constructed and gain references to other objects at run time. As a consequence, composition is a more flexible approach than inheritance. Using composition, one can change an object's implementation at run time. Inheritance forces implementations to be fixed at compile time.

Given that a sub-class becomes dependent on the implementation of its parent class, it is often said that implementation inheritance "breaks encapsulation" [96]. A class which is derived using implementation inheritance is able to rely upon the implementation details of its parent class rather than accessing the parent through its interface.

Using composition, the objects in question are forced to respect each other's interfaces, and rely on nothing more. This forces the architect to use carefully designed interfaces and interchangeable implementations. As such, it does not break encapsulation and results in far fewer implementation dependencies.

Composition generally results in smaller classes and smaller class hierarchies that are less likely to become unmanageable. The consequence of this is that the system typically has more objects; the behaviour or the system depends more on the relationships between these objects than it does on classes.

"Inheritance is over used as a reuse technique, and designs are often made more reusable (and simpler) by doing more composition and less inheritance" [25].

Delegation is a technique for achieving the same results as implementation inheritance through the use of composition and forwarding. Rather than inheriting the implementation of the methods of the parent class, the "parent" object is composed into the new object. Calls to the new object's methods are simply forwarded to the "parent" class.

Implementation inheritance can always be replaced by delegation.

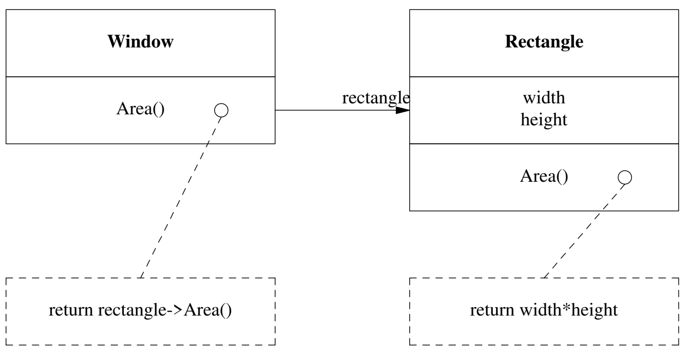

Figure 5.12: Reusing a class through composition and delegation rather than inheritance

5.4 Object oriented programming in Mercury

We have explored the mechanisms behind object oriented design, the support for these mechanisms in various object oriented programming languages, and the principles that lead to reusable and maintainable object oriented designs. We next examine the modelling of objects in a purely declarative language.

Previous schemes to allow object oriented programming in logic programming [12, 111] have been based around extending the language with specific "object" or "class" constructs, including an implementation inheritance mechanism. Object systems of modern logic programming languages such as SICStus Prolog [103] and Visual Prolog [87] are based around these ideas. In both of these languages, objects are introduced by declarations which specify a set of facts and rules whose scope is local to that object. The internal state of an object is maintained by asserting and retracting different values for the arguments of the facts local to that object. For example, the following Visual Prolog code implements a "`counter`" class, with methods for incrementing and decrementing, as well as getting the current value of the counter. The state of the counter is maintained by asserting a new clause for the `count` predicate. This predicate is declared as "`single`", meaning that only one clause is ever stored for that predicate. As a consequence, whenever a new clause is asserted, the old one is implicitly retracted.

```
CLASS counter
PREDICATES
       inc()
       dec()
       INTEGER getval()
ENDCLASS

IMPLEMENT counter
FACTS
       single count(INTEGER)
CLAUSES
       count(0).
       inc:-
           count(X),
           X1=X+1,
           assert(count(X1)).
       dec:-
           count(X),
           X1=X-1,
           assert(count(X1)).
       getval(Val):-
           count(Val).
ENDCLASS
```

The SICStus Prolog scheme is similar, although because the underlying language is untyped, the types of methods are not declared, and as such there is no way of specifying an interface without a corresponding implementation.

Schemes such as these attempt to directly mimic the "class" constructs of traditional object-oriented programming languages. The main drawback of these schemes is that they inherently rely upon objects being implicitly stateful, and thus violate referential transparency.

We present a different approach, based around the use of type classes and existential types which

- Allows the language to stay pure; we retain a simple declarative semantics.
- Can be used in any higher order logic programming language with type classes and existential types.
- Allows the programmer to express object oriented designs. We give several examples of translating a design from UML to Mercury.
- Leads to more robust and maintainable systems than existing approaches for logic programming languages.

5.4.1 Representing object state

In section 5.1, we introduced the concept of objects being entities composed of both data and methods to operate upon that data. Object oriented programming languages typically provide some explicit language constructs to support the creation of such entities. In C++, for example, the `class` construct allows object interfaces and implementations to be defined directly. Objects can be instantiated from classes by simply declaring a variable of that type. (C++ also features a system of *constructors* which support the automatic initialisation of objects). A pure logic programming language, however, has no such construct. The entities in a logic programming language are all logical terms (including higher order terms). We can take some inspiration here from work such as [21, 33, 78, 85] which investigate type-theoretic models of the constructs found in object oriented languages.

First of all, it is necessary to consider the issue of state. Objects in traditional, imperative object oriented languages are inherently stateful. When we invoke a method on an object that somehow changes its internal data, we are changing its state. An object has one *identity*, but its *state* changes throughout its lifetime. Values in a pure logic programming language, however, are not allowed to change. Once a variable's value is bound, it is bound to that value for the lifetime of that variable.

The logic programmer uses a different notion of state. Rather than each object having one identity and taking on many different states, the logic programmer has a notion of state whereby each value in a program represents a single state, but each identity is represented by many different values throughout the program. The logic programmer "threads" the state of objects through the program.

In the following Mercury predicate, the database object has three different states. Its initial state is referred to as `D0`. After the call to the predicate `delete_dropouts_from_database`, it is referred to as `D1`. The final state of the object, after the call to `insert_new_enrolments_into_database`, is called `D`. This final state is output from the predicate, through the output argument that appears in the predicate's head.

```
:- pred update_student_database(list(student), list(student),
                student_database, student_database).
:- mode update_student_database(in, in, in, out) is det.

update_student_database(Dropouts, NewEnrolments, D0, D) :-
    delete_dropouts_from_database(Dropouts, D0, D1),
    insert_new_enrolments_into_database(NewEnrolments, D1, D).
```

Note that there is nothing explicit in the program to denote that the three different variables denote the same object; it is a concept that exists entirely in the programmer's mind. (One danger of this approach is that the programmer may accidentally use a variable that refers to an old state of an object after the state of the object has been updated. This would have the effect of ignoring all changes to the object's state since that state variable was used. For this reason, the Mercury programmer will often use *unique modes* to ensure that old state values are never reused. The unique mode system allows variables to be marked as dead once the state of the object they denote has changed).

An alternative approach is to separate the entities that refer to identity and state completely. One value can refer to the object's identity, while another set of objects refers to the states of that object (and potentially of many others). In Haskell, this is done using a *state monad* [107]. The Mercury programmer will use the `store` module, which defines:

- the `store(S)` type class which represents an explicit state for objects referred to by `mutvars`.

- the `mutvar(T,S)` type which represents a mutable reference to a value of type `T` in a store of type `S`.
- an instance delaration for `store(io__state)`, which allows us to create and manipulate references with respect to the program's `io__state`.
- a predicate

  ```
  :- some [S] store__new(S) => store(S).
  ```

  which allows us to create a new store for use in situations where the `io__state` is not available.

Rather than threading the object which corresponds to our student database through the code, the example above can instead be written as:

```
:- pred update_student_database(list(student), list(student),
                mutvar(student_database, S), S, S)
                       <= store(S).
:- mode update_student_database(in, in, in, di, uo) is det.

update_student_database(Dropouts, NewEnrolments, D, S0, S) :-
    delete_dropouts_from_database(Dropouts, D, S0, S1),
    insert_new_enrolments_into_database(NewEnrolments, D, S1, S).
```

or, using DCG notation:

```
update_student_database(Dropouts, NewEnrolments, D) -->
    delete_dropouts_from_database(Dropouts, D),
    insert_new_enrolments_into_database(NewEnrolments, D).
```

While still requiring us to thread our `store` argument (albeit through a hidden argument in our DCG version), we are now able to refer to the student database object by a single name, `D`, throughout the body of this predicate.

At some point, however, we need to access an actual `student_database`, and not just a reference to one. The predicate `new_mutvar` initialises a `mutvar` from a given value, while the predicates `get_mutvar` and `set_mutvar` can be used to dereference and update a reference of a `mutvar`.

```
:- pred new_mutvar(T, mutvar(T, S), S, S) <= store(S).
:- mode new_mutvar(in, out, di, uo) is det.
```

```
:- pred get_mutvar(mutvar(T, S), T, S, S) <= store(S).
:- mode get_mutvar(in, out, di, uo) is det.

:- pred set_mutvar(mutvar(T, S), T, S, S) <= store(S).
:- mode set_mutvar(in, in, di, uo) is det.
```

For example, within the implementation of `delete_dropouts_from_database`, we will need to dereference our `mutvar` to access the actual `student_database` object. We can expect the code to look something like this:

```
:- pred delete_dropouts_from_database(list(student),
            mutvar(student_database, S), S, S) <= store(S).
:- mode delete_dropouts_from_database(in, in, di, uo) is det.
delete_dropouts_from_database(Dropouts, D, S0, S) :-
    get_mutvar(D, Database0, S0, S1),
    list__foldl(delete_student, Dropouts, Database0, Database),
    set_mutvar(D, Database, S1, S).

:- pred delete_student(student,
                        student_database, student_database).
:- mode delete_student(in, in, out) is det.
delete_student(...) :- ...
```

The ability to separate an object's state from its identity using the approach we have just described using `mutvar` and `store` is very convenient, and can lead to considerably more legible code. As we have described it, however, it is little more than a convenience. The real value of the approach using explicit references such as `mutvar` is that:

- It allows us to have more than one reference to an object, and when the object is changed through one reference, any other users of that object will have access to the updated object. This relieves the programmer the task of manually updating a value in many places in a program once that value changes. Instead, users of an object can just keep track of a reference, and any use of the object pointed to by the reference always go through the current value of the `store`.

- When necessary, it allows us to construct programs in which objects can refer to each other in a cyclic manner.

5.4.2 Representing object interfaces and implementations

A notion of state is only the first step towards being able to implement an object oriented architecture; we are yet to tackle the issue of associating methods with an object. One simple approach lies in the use of explicit higher order constants. We can construct objects dynamically by building a structure which contains both the data representing the object and the methods that operate upon that data.

Consider the UML diagram in figure 5.13. We could represent this in Mer-

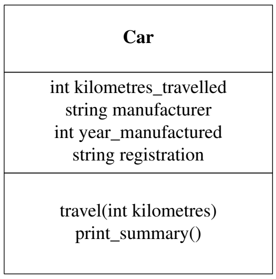

Figure 5.13: UML representation of a "Car" class

cury as:

```
:- type car_data
            ---> car_data(
                   int,    % kilometres travelled
                   string, % manufacturer
                   int,    % year manufactured
                   string  % registration info
            ).

:- type car
            ---> car(
                   car_data,                 % current state
                   pred(car_data,            % travel
                        car_data, int),
                   pred(car_data,            % print_summary
                        io__state, io__state)
            ).

:- pred travel(car_data, car_data, int).
...
```

```
:- pred print_summary(car_data, io__state, io__state).
...

:- func my_car = car.
my_car = car(
                car_data(195462, "Mitsubishi", 1992, "ABC-123"),
                travel,
                print_summary
            ).
```

Note that the travel method is modelled as a predicate with two arguments of type car_data (representing that state of the object before and after the call) while the print_summary method only has a single argument of that type. This is because the printing method presumably does not change the state of the car object in any way. (The programmer could also make the stylistic choice that it is more consistent to pass two arguments of this type to every method, regardless of whether the method changes the object's state.)

This representation, however, does not allow us to reuse the car structure for different implementations of the class.

The UML diagram in figure 5.14 shows the situation where we have abstracted away some of the details of the Car class.

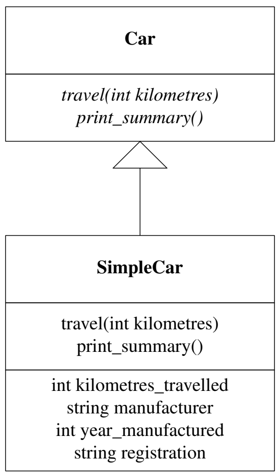

Figure 5.14: A better representation of a "Car" class

This gives us a hint towards a more reusable implementation:

```
:- type car(T)
        ---> car(
                T,
```

```
                  pred(T, T, int),              % travel
                  pred(T, io__state, io__state) % print_summary
         ).
```

We can now reuse this interface for different kinds of representations.

```
:- type simple_car_data
         ---> simple_car_data(
                 int,       % kilometres travelled
                 string,    % manufacturer
                 int,       % year manufactured
                 string     % registration info
          ).
:- type simple_car
         ---> simple_car(
                  car(simple_car_data)
         ).
:- func my_car = simple_car.
my_car = simple_car(
              car(
                  simple_car_data(
                      195462,
                      "Mitsubishi",
                      1992,
                      "ABC-123"
                  ),
                  car_travel,
                  print_car
              )
           ).

:- type hire_car_data
         ---> hire_car_data(
                   list(int), % list of journey lengths
                   string,    % manufacturer
                   string,    % model
                   int,       % year manufactured
                   string,    % hire company
                   string     % registration info
         ).
```

```
:- type hire_car
        ---> hire_car(
                car(hire_car_data).
        ).
:- func my_hire_car = hire_car_data.
my_hire_car = hire_car(
            car(
                hire_car_data(
                    [23, 129, 4, 412, ...]
                    "Mazda",
                    "626",
                    1998,
                    "Hertz"
                    "XYZ-987"
                ),
                hire_car_travel,
                hire_print_car
            )
        ).
```

5.4.3 Type classes as class interfaces

The technique that we sketched in the previous section for representing objects in logic programming has several drawbacks.

- It is a dynamic technique — methods are not assigned to objects until run time. As such, the structure of the software is not as clear as it could be from simply examining the source code. While there is certainly some flexibility gained by being able to constructs objects with new methods at run time, this flexibility comes at the cost of making it much more difficult to statically describe the interfaces to software structures.

- It is tedious and error prone. The programmer is responsible for assigning methods to each object as it is constructed. As class hierarchies become more and more complex, the task of constructing an object becomes even more arduous, and calling a method of a class from higher up the hierarchy (presumably involving retrieving the method explicitly from some nested structure) becomes even more so.

- Because Mercury's type system only allows first order quantification of type variables, we cannot construct objects with truly polymorphic methods.

A better solution lies in the use of type classes. A type class represents a set of operations that can be performed upon a type. Rather than explicitly constructing a set of methods using higher order constants, we can use the type class mechanism to implicitly construct the set of methods for us. Methods are assigned to objects automatically by the compiler, even in the presence of extremely complicated class hierarchies. By simply constraining the data in our object to be a member of a particular type class, we implicitly pass along a set of methods with the object.

Continuing our example from the previous section, we can translate the Car class from our UML diagram in figure 5.14 into:

```
:- typeclass car(T) where [
        pred travel(T, T, int),
        mode travel(in, out, in) is det,
        pred print_summary(T, io__state, io__state),
        mode print_summary(in, di, uo) is det,
].
```

Implementations are now provided by:

```
:- instance car(simple_car_data) where [
        pred(travel/3) is car_travel,
        pred(print_summary/3) is print_car
].
:- instance car(hire_car_data) where [
        pred(travel/3) is hire_car_travel,
        pred(print_summary/3) is hire_print_car
].
```

5.4.4 Implementing interfaces

We have sketched a simple program which uses a type class to model a UML class, and provided two implementations using `instance` declarations.

It is worth being explicit about the language mechanism that have allowed us to do this:

- Definitions of abstract classes can be translated directly into `typeclass` declarations. Methods of the abstract class translate directly to methods of the corresponding type class.

- `type` declarations can provide definitions for object data formats.
- `instance` declarations are the interface inheritance mechanism. We can provide different implementations of an abstract class by providing different instances of a type class for different types.

5.4.5 Existential typing to encapsulate object implementations

Using constrained polymorphic types to represent objects has the disadvantage that the type of a value, or data representation of an object, must be known at the point at which a method is called in the call tree. That type may be an abstract type, but the particular type must still be known.

Existential types provide a solution to this. We can encapsulate the data representation of an object while constraining that representation to support the methods of the interface that we intend to use it as:

```
:- type car ---> some [T] car(T) => car(T).
```

By using this technique to encapsulate the representation of an object, we are able to treat objects in a manner similar to their treatment in traditional object oriented languages. For example, this technique allows us to build heterogeneous collections of objects which all satisfy the same type class interface.

This allows us to model the association, aggregation and composition mechanisms of object oriented design. For example, when one class has an association with another class, the first class only accesses the second through its interface. As such, any class which is derived from the second class can be used, regardless of its implementation. Existential quantification of the implementation of a class models this mechanism directly.

We will show several applications of this technique in the following sections.

5.4.6 Implementation inheritance and delegation

The methodology we have described here for implementing object oriented designs in Mercury has not included a way of expressing implementation inheritance. We have separated the notions of interface (`typeclass`) and implementation (`type`), and provided a construct for inheriting interfaces (`instance`), but we have not created a construct for inheriting implementations.

The type class mechanism of Haskell [29] allows the programmer to declare *default implementations* for type class methods. Any such implementation is declared along with the signature of the method in the type class declaration, and that implementation is used by any instance which does not provide its own implementation for the method. This may give the impression of providing an implementation inheritance mechanism, but in reality this is not the case. Default methods do not provide an implementation hierarchy; instead, only a single default implementation of a method is associated with the abstract base class. As such, the default implementation cannot make use of any particular representation for the object. The default implementation must be usable for *any* instance of the abstract base class. Default methods merely provide a syntactic shorthand which is useful in the case where most instances of a type class method are expected to use one implementation of a method, and that implementation is given in terms of calls to other methods of the type class.

Instead, the mechanism we advocate for implementation inheritance is the use of delegation. For example, consider the situation in figure 5.15, where "SubClass" is an implementation of "AbstractClass", and "SubSubClass" inherits the implementation of "SubClass", overriding the definition of method "Operation1".

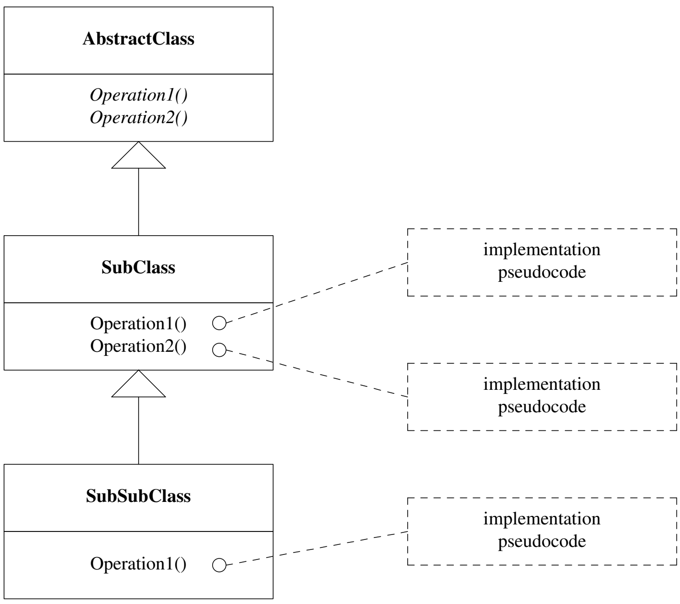

Figure 5.15: An inherited implementation

We can implement this in a straightforward manner in Mercury, using delegation to model the implementation inheritance of "SubClass" by "SubSubClass". The abstract base class corresponds to a type class, while the two concrete classes correspond to types which are both instances of this type class.

```
:- typeclass abstractclass(T) where [
    pred operation1(T, T, ...),
    pred operation2(T, T, ...)
```

```
].

:- type subclass ---> ...
:- instance abstractclass(subclass) where [
    (operation1(...) :- ...),
    (operation2(...) :- ...)
].

:- type subsubclass
          ---> some [T] i(
                  T,
                  ...
          ) => subclass(T).

:- instance abstractclass(subsubclass) where [
    (operation1(i(S0, ...), 'new i'(S, ...)) :-
         operation1(S0, S, ...)),
    (operation2(i(S0, ...), 'new i'(S, ...)) :-
         operation2(S0, S, ...))
].
```

`subsubclass` inherits the implementation of `operation2` from `subclass` by delegation. The `coloured_point` example from section 4.4 is an example of this scheme.

Note that the implementation of inheritance through delegation is really a very simple source-to-source transformation that can be automated. This approach has been taken in Haskell++ [37], a Haskell preprocessor which provides a more traditional object oriented programming language style class construct (in which data and methods are declared in the same construct) and a corresponding implementation inheritance mechanism. Its implementation uses a well defined source-to-source transformation to implement inheritance using delegation. The program resulting from the transformation is pure Haskell code.

5.5 Logic programming and design for change

Type classes and existential types provide an elegant and simple way of modelling objects.

The principles of good object oriented design all indicate that the use of interface inheritance is the key to designing for change. Implementation inheritance, while powerful, in general has a detrimental effect on software quality through the unnecessary introduction of dependencies. The language mechanisms we have shown (`type`, `typeclass` and `instance` declarations) provide no mechanism for implementation inheritance. This is quite deliberate: implementation inheritance is a poor architectural tool that results in designs with dependency structures which make maintenance and reuse more difficult. We believe that the net effect of implementation inheritance on software architecture tends to be a negative one.

Object oriented architectures implemented in our type system are more likely to follow the principles of good object oriented design that we outlined in section 5.3.2. By syntactically separating the notions of interface and implementation, we make it far easier for the programmer to produce programs whose dependency structures look like those in figure 5.9. In addition, interface inheritance is a powerful tool for constructing programs that obey the LSP: the programmer can very simply provide different implementations of an abstract interface. It is also a very simple matter to allow an object to be accessed through a number of different abstract interfaces. The following code fragment implements the class diagram shown in figure 5.11.

```
:- typeclass service_a(A) where [
        <<client A methods>>
        ...
].
:- typeclass service_b(B) where [
        <<client B methods>>
        ...
].
:- typeclass service_c(C) where [
        <<client C methods>>
        ...
].

:- type service.
:- instance service_a(service) where [ ... ].
:- instance service_b(service) where [ ... ].
:- instance service_c(service) where [ ... ].
```

For every programming language, there are things which are simply expressed in terms of the language's constructs, and there are things that are not. For those things which are not, a set of *paradigmatic idioms* [3] develop. The C programmer, for instance, is forced to follow very stringent guidelines to separate a program into modules. As time has gone by, programming language designers have realised that modules are typically worthy of being a programming language construct in their own right. To the declarative language programmer, "state" is a common programming idiom: the programmer either threads an object through the program and has some implicit notion that a sequence of variables denote the sequence of states of an object, or uses some other mechanism such as the `store` module or a state monad. Designers of declarative languages typically make the decision that these idioms are unobtrusive enough that the addition of stateful objects to the language is not necessary: the loss of referential transparency is not perceived as worthwhile. It is also possible to add direct support for the state-threading idiom, using constructs such as definite clause grammars (DCGs) or extended DCGs [90], while maintaining a simple declarative semantics for the language. Persistent or mobile objects, coroutining and distribution are other examples of idioms programmed explicitly in some languages, while in others where these mechanisms are deemed important enough there are explicit language constructs to implement them.

The same is true of object oriented programming. The C programmer who wishes to program in an object oriented style uses idioms such as "interfaces" and "polymorphism". The C++ programmer, however, uses explicit language constructs to implement these things, but no language can ever hope to provide language constructs to implement everything that a programmer may want to express. The common idioms used by programmers are often referred to as *design patterns*. Many common design patterns for object oriented languages are described in [25, 86].

We now examine several common object oriented design patterns, and show that these can be expressed simply using type classes and existential types.

5.5.1 Abstract factory pattern

The DIP states that all dependencies should be upon abstractions. The abstract factory pattern [25] is a way of abstracting the process of object creation so that all parts of the program that need to create an object can do so through an abstraction. Figure 5.16 depicts two abstract classes of widgets: Window and Scrollbar. Each of these widgets is implemented in both the Presentation Manager and Motif toolkits.

In an application that creates and uses windows and scrollbars, it is desirable

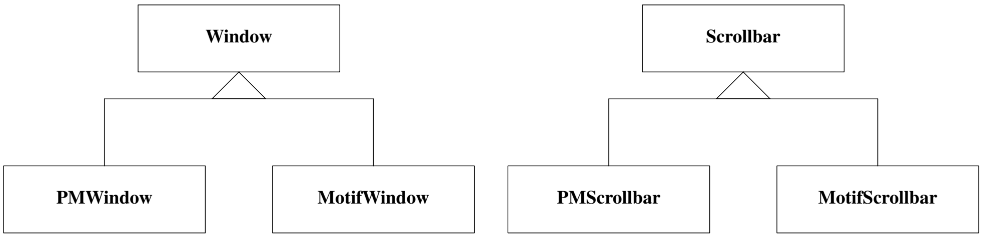

Figure 5.16: Window and Scrollbar classes

to be able to easily change between windowing toolkits. However, the actual toolkit being used must be specified whenever a widget is created. As such, it is undesirable to create widgets explicitly from either toolkit. Instead, it is best to define an abstract factory, shown in figure 5.17 as WidgetFactory.

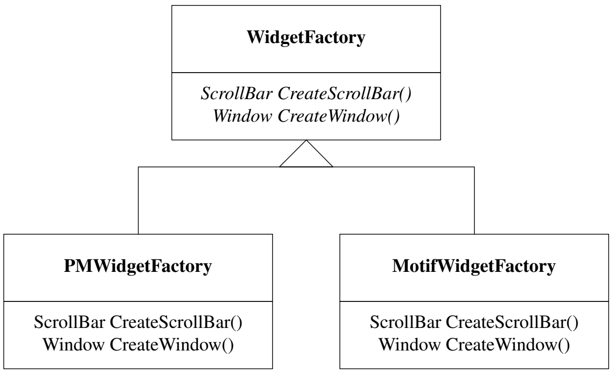

Figure 5.17: WidgetFactory hierarchy

We can represent this very simply with type classes and existential types. Note that the interface types returned by the WidgetFactory methods are represented as existentially typed values, constrained to be in the appropriate type class.

```
    % Type classes to represent the two abstract interfaces.
:- typeclass window(W) where [ ... ].
:- typeclass scrollbar(S) where [ ... ].

:- type pmwindow.
:- type motifwindow.
:- type pmscrollbar.
:- type motifscrollbar.

:- instance window(pmwindow) where [ ... ].
```

```
:- instance window(motifwindow) where [ ... ].
:- instance scrollbar(pmscrollbar) where [ ... ].
:- instance scrollbar(motifscrollbar) where [ ... ].

   % The abstract factory
:- typeclass widgetfactory(F) where [
    some [S] pred createscrollbar(F, S, io__state, io__state)
                        => scrollbar(S),
             mode createscrollbar(in, out, di, uo) is det,
    some [W] pred createwindow(F, W, io__state, io__state)
                        => window(W),
             mode createwindow(in, out, di, uo) is det
].

:- type pmwidgetfactory.
:- type motifwidgetfactory.
:- instance widgetfactory(pmwidgetfactory) where [
    (createscrollbar(_F, S) --> create_pmscrollbar(S)),
    (createwindow(_F, W) --> create_pmwindow(W))
].
:- instance widgetfactory(motifwidgetfactory) where [
    (createscrollbar(_F, S) --> create_motifscrollbar(S)),
    (createwindow(_F, W) --> create_motifwindow(W))
].
```

5.5.2 Composite pattern

The Composite pattern [25] allows a client to treat objects and compositions of objects in a uniform manner. For example, applications such as graphical editors allow the user to build complex diagrams from graphical components such as geometrical objects, images and text. In addition, graphical components can be grouped together into a single component. These groupings can be used as elements of a diagram, or can be used as elements of another group.

A UML class diagram for part of a simple graphical editor is given in figure 5.18[2]. Rather than defining a class corresponding to graphical primitives and another class corresponding to composite elements, primitives and composites

[2]The diamond shaped figure at the beginning of the arrow connecting the "Picture" and "Graphic" classes denotes the "aggregation" relationship between the "Picture" and "Graphic" classes. The filled circle at the end of the arrow denotes that instances of the "Picture" class contain one or more "Graphic" elements.

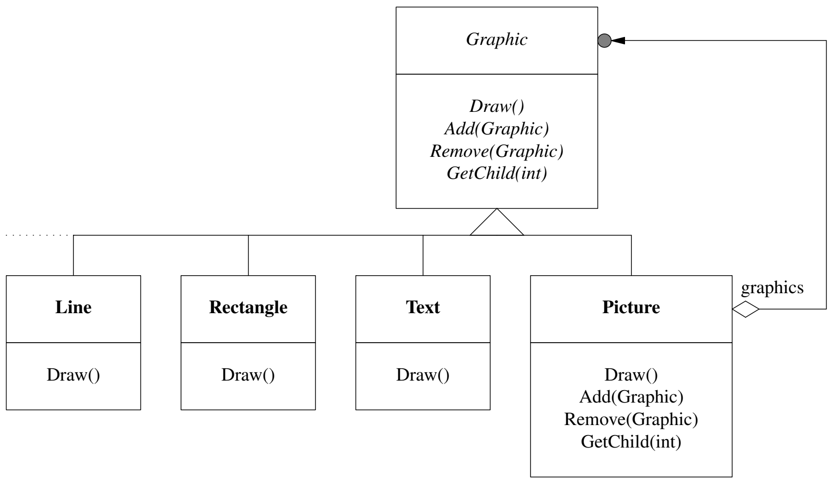

Figure 5.18: Class diagram for a graphical editor

are modelled as sub-classes of the same super-class “Graphic”. This relieves the client of the burden of distinguishing between primitive and composite elements.

(Note that the only method implemented by the classes corresponding to primitive elements is “Draw()”. All the other methods are only implemented by the class corresponding to composite elements. A more safety-conscious, but less flexible, design might not include those methods in the interface of “Graphic”.)

We can implement this design quite directly in Mercury using type classes and existential types. The abstract “Graphic” class corresponds to the following type class

```
:- typeclass graphic(G) where [
    pred draw(mutvar(G, io__state), io__state, io__state),
    mode draw(in, di, uo) is det,
    pred add(mutvar(G, io__state), mutvar(Child, io__state),
        io__state, io__state) <= graphic(Child),
    mode add(in, in, di, uo) is det,
    pred remove(mutvar(G, io__state), mutvar(Child, io__state),
        io__state, io__state) <= graphic(Child),
    mode remove(in, in, di, uo) is det,
    some [Child] pred getchild(mutvar(G, io__state),
        mutvar(Child, io__state), io__state, io__state)
        => graphic(Child),
    mode getchild(in, out, di, uo) is det
].
```

Note that we have used `mutvars` and the `io__state`, as described in section 5.4.1, to encapsulate the state of the picture elements in our design. A picture editor is an example of an application where objects are likely to be referred to by many different parts of a system. By using a level of indirection through a `mutvar`, the programmer is relieved of the error-prone task of keeping all references to a given object consistent. In addition, this approach avoids the run time overhead that would be likely to occur if for example every composite object had to be reconstructed each time one of its components changed.

We can implement the classes corresponding to the primitive elements quite simply:

```
:- type line.
:- type text.
:- type rectangle.

:- instance graphic(line) where [
    (draw(L) --> draw_line(L)),
    (add(_, _) --> { error("cannot add child to line) }),
    (remove(_, _) --> { error("cannot remove child from line) }),
    (getchild(_, _, _) :- { error("cannot getchild from line) }),
].
:- instance graphic(text) where [
    (draw(T) --> draw_text(T)),
    (add(_, _) --> { error("cannot add child to text) }),
    ...
].
:- instance graphic(rectangle) where [
    (draw(R) --> draw_rectangle(R)),
    ...
].
```

The composite "Picture" class can be implemented quite directly by a type which stores a list of child elements.

```
:- type graphic ---> some [G] g(G) => graphic(G).
:- type picture ---> picture(list(graphic)).

:- instance graphic(picture) where [
    (draw(P) -->
        get_mutvar(P, picture(Cs)),
```

```
        foldl(draw_graphic, Cs)
    ),
    (add(P, C) -->
        get_mutvar(P, picture(Cs)),
        set_mutvar(P, picture([C|Cs])),
    ),
    (remove(P, C) --> ...),
    (getchild(P, I, C) --> ...)
].

:- pred draw_graphic(graphic, io__state, io__state).
:- mode draw_graphic(in, di, uo) is det.

draw_graphic(graphic(G)) -->
    draw(G).
```

5.5.3 Visitor pattern

The visitor pattern [25] is used when there are a number of operations to be performed at all the nodes in a structure. The purpose of this pattern is to be able to add new operations without changing the classes that define the nodes in the structure.

The prototypical example of this pattern is that of a compiler which manipulates an abstract syntax tree [25]. The syntax tree has various kinds of nodes, representing such things as function calls, assignments and variable references. The compiler will typically perform many different kinds of operations on these nodes, such as type checking, code generation and pretty printing. However, the naive class structure shown in figure 5.19 is less than ideal: for each operation, the code specific to each node type is associated with the node itself.

For example, this means that the type checking code would need to be spread out amongst the definitions of each of the node types. It would be much easier to comprehend and maintain if the code for each operation could be kept separately. Instead, using the visitor pattern, we keep two separate class hierarchies: one with a sub-class for each kind of node and one with a sub-class for each operation.

The operation class hierarchy is shown is figure 5.20. There is a sub-class for each operation. Each operation has a corresponding implementation for each node type.

The node class hierarchy is shown is figure 5.21. There is a sub-class for

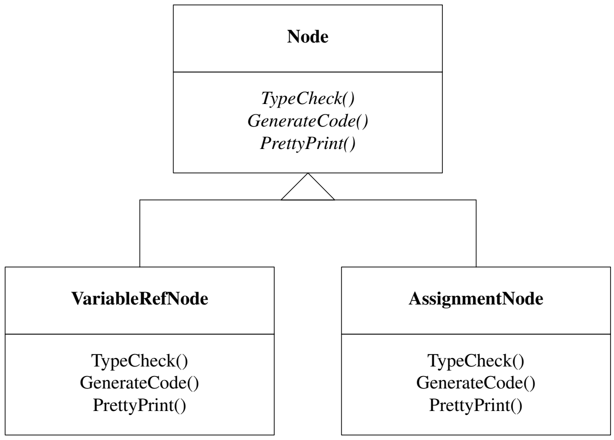

Figure 5.19: A naive structure for syntax tree nodes

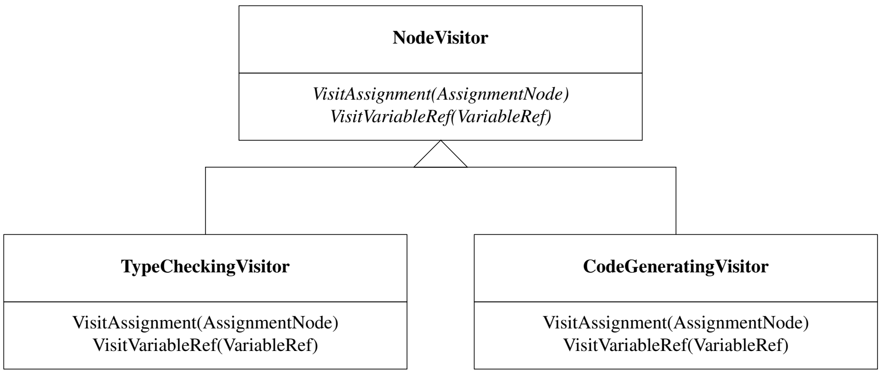

Figure 5.20: Visitor structure using the visitor pattern

each node type. Each class has a single method which merely forwards the analysis to the correct code for that node type.

This structure enables us to add operations by simply inheriting from the NodeVisitor class and providing an implementation for each node type. Adding a new node type requires adding a method to NodeVisitor, and consequently providing an implementation of that method to the sub-class of NodeVisitor corresponding to each analysis.

We can implement this design directly in Mercury:

```
:- type assignment_node.
:- type variable_ref_node.

:- typeclass node_visitor(NV) where [
    pred visit_assignment(assignment_node, NV, NV),
    mode visit_assignment(in, in, out) is det,
```

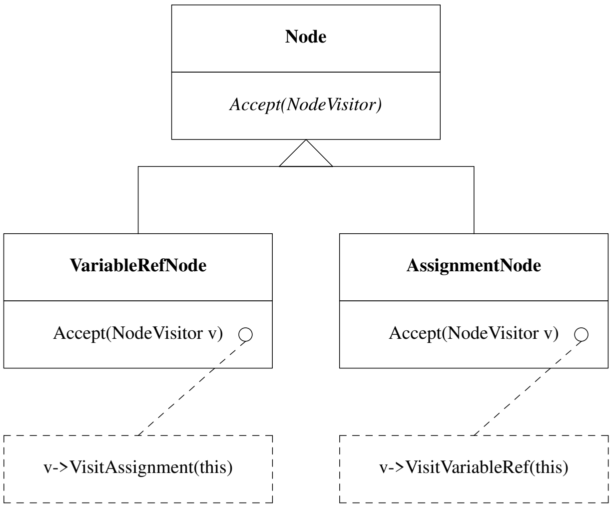

Figure 5.21: Node structure using the visitor pattern

```
    pred visit_variable_ref(variable_ref_node, NV, NV),
    mode visit_variable_ref(in, in, out) is det
].

:- type typechecking_visitor.
:- type codegenerating_visitor.

:- instance node_visitor(typechecking_visitor) where [
    pred(visit_assignment/3) is ...
    pred(visit_variable_ref/3) is ...
].
:- instance node_visitor(codegenerating_visitor) where [
    pred(visit_assignment/3) is ...
    pred(visit_variable_ref/3) is ...
].

:- typeclass node(N) where [
    pred accept(N, NV, NV) <= node_visitor(NV),
    mode accept(in, in, out) is det
].

:- instance node(assignment_node) where [
    (accept(A, NV0, NV) :- visit_assignment(NV0, NV, A))
].
```

```
:- instance node(variable_ref_node) where [
    (accept(V, NV0, NV) :- visit_variable_ref(NV0, NV, V))
].
```

While it is comforting to see that we can directly translate the UML class diagram into Mercury code, this is not how the Mercury programmer would typically solve this problem.

We will now examine two alternative approaches.

Using a multi parameter type class

As noted in [3], the purpose of the visitor pattern is really to simulate multi-way method dispatch. The `accept` method effectively encodes the implementation of the node and passes this information to the visitor object which executes the correct method implementation for that node and visitor operation. We have a mechanism for doing this directly: multi-parameter type classes.

```
:- type typechecking_info.
:- type codegenerating_info.

:- type assignment_node.
:- type variable_ref_node.

:- typeclass node(Node, Analysis) where [
    pred apply_operation(Node, Analysis, Analysis),
    mode apply_operation(in, in, out) is det
].

:- instance node(assignment_node, typechecking_info) where [
    pred(apply_operation/3) is ...
].
:- instance node(variable_ref_node, typechecking_info) where [
    pred(apply_operation/3) is ...
].
:- instance node(assignment_node, codegenerating_info) where [
    pred(apply_operation/3) is ...
].
:- instance node(variable_ref_node, typechecking_info) where [
    pred(apply_operation/3) is ...
].
```

Using a discriminated union type

There is, however, an even simpler solution still. Given that the problem we are trying to solve is to have all the code for the operation we are applying to the node in one place, and we have a node structure which is unlikely to change, the solution chosen by most declarative language programmers would be to simply declare a discriminated union type with an alternative for each node type. The implementation of each operation can then simply switch on the function symbols of this type, and thus execute the correct code for each node type.

```
:- type node
        ---> variable_ref_node(...)
        ;    assignment_node(...)
        ;    ...
        .

:- pred typecheck(node, ...).

typecheck(N, ...) :-
   (
       N = variable_ref_node(...),
           % handle the variable_ref_node case
       ...
   ;
       N = assignment_node(...),
           % handle the assignment_node case
       ...
   ;
       ...
   ).
```

New operations can be added as we please. If we add a new node type, the switches in the implementations of each operation will need to have a case added to handle the new node.

This approach is certainly simpler and probably more efficient and less error prone than the "object-oriented" solution. Note that our original "object-oriented" solution is no more complicated than the corresponding C++ or Java solution; it is merely more complicated than the solution which uses the language features already built in to most declarative languages.

5.6 Remarks

Object oriented design is a powerful tool in the design of software, but there are pitfalls. Indiscriminate use of inheritance can lead to software that is difficult to reuse and maintain. However, by applying the principles discussed in this chapter, many of these pitfalls can be avoided.

The primary challenge for the designer of a programming language for large-scale software development is to produce a language which encourages the software engineer to write software which is both robust and maintainable. We believe that by providing a mechanism for interface inheritance, we do exactly that.

We have described how to implement object oriented designs in Mercury, making particular use of type classes and existential types. This is not to say that Mercury is an object oriented programming language. Rather, type classes and existential types are significant architectural tools that can be used to take advantage of object oriented design techniques.

We believe that the system we have described is an improvement over existing object mechanisms for logic programming because we do not try to directly replicate the mechanisms of traditional object oriented programming languages at the cost of referential transparency. Rather, we have shown how to apply language constructs with simple semantics which fit consistently with the purely declarative logic programming paradigm. By providing a type class mechanism, we allow the programmer to focus on the important abstractions, namely the static interfaces of the various software structures.

We also believe that our system is likely to result in significantly more maintainable architectures than the architectures that result from programming in traditional object oriented programming languages. We have provided constructs which encourage architectures which obey the principles of design for change. While we do not provide a general (implementation) inheritance mechanism, we have shown that composition and delegation are viable alternatives to inheritance in our system.

Chapter 6

Components for Logic Programming

Purely declarative programming languages have many benefits over imperative languages. Modern declarative programming languages are expressive, efficient, have clear semantics and are amenable to aggressive optimisation, which makes them very attractive tools for large classes of problems.

For declarative languages to be used extensively in real-world programs, they cannot simply offer a solution (no matter how elegant) to a particular problem in isolation, nor can they demand exclusive use of one programming language. Declarative languages need to be able to be used as a part of a larger system, or their benefits to a particular subsystem will be ignored in face of the larger concerns of the system. Declarative languages must therefore include comprehensive interfaces with existing languages and systems if they are to see widespread adoption. Put simply, interoperability needs to have high priority in the language design.

We believe the design of the external interfaces of declarative languages should be guided by the following principles:

- The purity and expressiveness of the language should be maintained.
- The natural style of the language should not be unduly restricted (but can certainly be augmented) by an external interface.
- The interface should be as seamless as possible, unless visible seams have some practical benefit.
- The interface should be bi-directional, allowing code written in the declarative language to act both as provider and as consumer of services.

Many other languages and systems have had to address similar interoperability issues in recent years. This has lead to the development and refinement of new software development architectures, including a strong movement towards the development of systems based around components. Each component has a well defined interface but the implementation is relatively unconstrained – the tools, techniques and languages used for each component can be chosen based upon their merits, rather than the demands of the rest of the system.

Component based systems are a scaled up version of programs based on abstract module interfaces, and like their smaller cousins, they come in procedural and object oriented flavours. Procedural systems are based on procedure calls from one component to another, whereas object oriented systems view each component as an object and pass messages between objects.

While at first these systems mainly put components together to form systems on a single machine, the growth of networking and particularly the Internet has lead to the widespread use of distributed component based development, with systems being comprised of components connected via local or wide area networks. This involves either remote procedure calls, or distributed object systems invoking methods over a network.

Component based systems are an excellent opportunity to use declarative languages – their strengths can be put to work in isolation from the concerns of other components. This allows declarative languages to be adopted in a piecemeal fashion, which is an important consideration when programmers must be trained and familiarised with new techniques.

One of the most popular component based systems is CORBA, the Common Object Request Broker Architecture [26]. CORBA is a distributed object framework created and maintained by the Object Management Group (OMG) which represents the input of a large consortium of companies. CORBA is based around a standard Interface Definition Language (IDL). It uses Object Request Brokers (ORBs) to act as an object "bus", which allows easy local or remote communication between objects.

This chapter describes *MCORBA*, a CORBA binding for Mercury. This binding uses type classes and existential types to provide a natural, expressive, and yet still purely declarative interface to CORBA. We have used it both to operate upon existing CORBA components from Mercury and to implement CORBA components in Mercury.

H/Direct [20], an interface from Haskell to Microsoft's component framework, the Component Object Model (COM) [88], uses some similar techniques to our system. The lack of support for existential types in Haskell requires

H/Direct to model component interfaces as types, The MCORBA approach, which we introduce in this chapter, is to model component interfaces as type classes, which we believe to be significantly more natural.

Section 6.1 introduces CORBA and explains how we interface between CORBA and Mercury. The low-level details of the interface are explored in section 6.2.

6.1 Mercury binding for CORBA

The Common Object Request Broker Architecture (CORBA) is a standard designed to tackle the problem of interoperability between software systems. It allows applications and components to communicate, independent of whether they are executing in the same address space, independent of whether they are executing on the same machine, and independent of whether they are written in the same language.

Communication between software components in a system using CORBA is handled by an Object Request Broker (ORB). The ORB is responsible for establishing and maintaining communication between objects. The ORB is "middleware" through which a client can transparently invoke operations on an object, regardless of the location or implementation of the object. Different ORB implementations may communicate in different ways, so the CORBA 2.0 specification [26] includes a protocol for communication between ORBs, the Internet Inter-ORB Protocol (IIOP).

The interfaces of CORBA components are described in OMG's Interface Definition Language (IDL). An IDL specification defines a set of *interfaces*, each of which is the abstract "signature" of an object. Each interface includes a set of operations and attributes. (Attributes are equivalent to a pair of operations, one that sets the attribute to a value and one that retrieves the value, so we will not mention them further.) For each operation, the interface defines the types of the parameters, and for each parameter, whether the parameter is an input (`in`), an output (`out`) or both (`inout`). (This level of detail in the description is necessary for strongly typed languages such as Ada, Eiffel and Mercury, and even for weakly typed languages such as C when different components reside in different address spaces or on different machines.) For example, the following IDL specification:

```
interface CallBack {
    void NewMessage(in string msg);
};
```

defines an interface which has one operation, a one-argument procedure taking a string as input.

A CORBA binding for a particular language provides a way for code written in that language to operate on objects whose interface is described in IDL, and to implement objects whose interface is described in IDL. This binding takes the form of a tool that takes an IDL specification and automatically generates *stubs* and *skeletons* in the target language. When a client written in the target language wants to invoke one of the operations of a CORBA object, it invokes the corresponding stub routine. The stub routine acts as a proxy; it asks the ORB to invoke the same operation on the real object, regardless of its location and implementation language. The ORB communicates with the real object through the skeleton code, which invokes a method of the object when it receives a corresponding request from the ORB.

Standard mappings from IDL are defined for many programming languages including Java, C, C++, and Cobol. For C++, the stub will be a class that passes messages to its real implementation via the ORB. A user who requests an object from the ORB will be given an instance of the stub class, and will be able use it as if the class were implemented locally. The skeleton is an abstract class from which the user must derive a concrete class. Instances of the derived class can then be registered with the ORB so clients can use them, or be passed as parameters to other objects.

This chapter may be seen as a definition of a Mercury binding for CORBA. This binding consists of a run-time library (called `corba`), and a tool (called `mcorba`) which generates Mercury stubs and skeletons from IDL specifications.

`mcorba` generates a Mercury type class for each IDL interface. For each operation of the interface, the type class has a method. The method will have two pairs of arguments with modes `di` and `uo`, to denote the state of the object and the state of the external world before and after the invocation of the method. (An alternative design would replace the pair of arguments representing the state of the object with a single `mutvar` argument to explicitly represent the changing state of the object using the scheme outlined in section 5.4.1. This would ease the burden of *using* an MCORBA component, but would make the task of *implementing* a CORBA component in Mercury more difficult.) The types and modes of the remaining arguments will correspond to the types and modes of the arguments of the IDL operation, with the exception that for `inout` parameters, we generate two arguments, with modes `in` and `out`. (In this case the argument of type `string` has been mapped into a Mercury argument with type `corba__string`. Some of CORBA's built in types, including `string`, correspond directly to Mercury types and as such values of Mercury native types can be passed directly to CORBA. `string` is one such type, and as such the `corba` module exports a type definition for `corba__string` as an equivalence to Mercury's built in `string` type. We explain this in more detail in section 6.1.3.)

From the IDL specification above, `mcorba` would generate

```
:- typeclass callback(T0) where [
    pred newmessage(T0, T0, corba__string, io__state, io__state),
    mode newmessage(di, uo, in, di, uo) is det
].
```

(Note that the identifier names from the IDL specification are mapped to lower case identifiers as required by Mercury. This does not pose a problem as case makes no difference in an IDL specification. The rationale for this is that IDL specifications should map naturally into many different programming languages, and it is quite common for programming languages to impose restrictions on the case of identifiers.)

This type class serves two purposes:

- The ORB can give a client an object which belongs to the type class so that the client can invoke the appropriate operations on it (and no others).
- If a Mercury value is of a type which is an instance of the type class, it may be registered with the ORB as an object that satisfies the relevant interface.

We will now examine these two purposes in greater detail.

6.1.1 The forwards interface: Mercury calls CORBA

The generation of type classes for IDL interfaces, outlined above, is sufficient for a Mercury program to invoke operations on a CORBA object. However, we also need a way of getting a handle on such an object from the ORB. CORBA provides several methods for doing this. Each method is implemented by a predicate in the `corba` module. The predicate

```
:- some [T] pred corba__resolve_object_name(orb,
        corba__string, corba__string, T, io__state, io__state)
        => corba__object(T).
:- mode corba__resolve_object_name(in, in, in, uo, di, uo)
        is det.
```

uses the COS (Common Object Services) naming service, through which a client may request an object by name. Note that the client does not request an object of a particular interface type, but merely a generic CORBA object. Since the argument is existentially typed, the implementation of this object is hidden from the client, but it is known to be connected to the ORB since the predicates of the `corba` module guarantee that only such objects satisfy the `corba__object` type class constraint. It is then the client's responsibility to ask the ORB to "narrow" this object into the required interface. It can do so by calling the `narrow` predicate that `mcorba` generates for the desired IDL interface along with the type class. As this operation has the same name for each IDL interface, the code generated for each interface is placed inside a separate Mercury module.

We will show how this is done using as an example a "chat" server with three operations: sending a message to the server for distribution to all registered clients, registering a client, and removing a client. The IDL specification is:

```
interface Chat {
    void SendMessage(in string msg);
    void RegisterClient(in CallBack obj, in string name);
    void RemoveClient(in CallBack obj, in string name);
};
```

From this, `mcorba` generates the following type class:

```
:- typeclass chat(T0) where [
    pred sendmessage(T0, T0, corba__string,
            io__state, io__state),
    mode sendmessage(di, uo, in, di, uo) is det,
    pred registerclient(T0, T0, T1, corba__string,
            io__state, io__state)
                <= (callback(T1), corba__object(T1)),
    mode registerclient(di, uo, in, in, di, uo) is det,
    pred removeclient(T0, T0, T1, corba__string,
            io__state, io__state)
                <= (callback(T1), corba__object(T1)),
    mode removeclient(di, uo, in, in, di, uo) is det
].
```

Once again, there is a type class method corresponding to each IDL interface operation. This time, the `RegisterClient` and `RemoveClient` methods both have arguments which are objects satisfying IDL interfaces. The corresponding arguments of the type class methods are constrained to belong to both the `callback` and `corba__object` type classes, since the method may invoke both specific `CallBack` object or generic CORBA operations on that argument.

The automatically generated `narrow` operation for the `Chat` interface has the signature:

```
:- type corba__narrow_result(U, N)
                    ---> narrowed(N)
                    ;    unnarrowed(U)
                    .

:- some [C] pred chat__narrow(O, corba__narrow_result(O, C))
         => (chat(C), corba__object(C)) <= corba__object(O).
:- mode chat__narrow(di, uo) is det.
```

The `narrow` operation may or may not be successful; the CORBA object may or may not actually satisfy the required IDL interface. The ORB finds out whether it does or not. If it does, `narrow` will bind the second argument to `narrowed(Chat)`, where `Chat` is a value which belongs to the `chat` type class as well as the `corba__object` type class. If not, it binds the second argument to `unnarrowed(Obj)`, where `Obj` is the original non-narrowed value.

The following code fragment shows how a Mercury program can use the library predicate `corba__resolve_object_name` and the automatically generated predicate `chat__narrow` to connect to a CORBA chat server. The program can then send the lines of text that it reads from the user to the chat server.

```
:- pred sender(corba__orb, io__state, io__state).
:- mode sender(in, di, uo) is det.
sender(Orb) -->
    corba__resolve_object_name(Orb, "Chat", "Server", Obj),
    { chat__narrow(Obj, Narrowed) },
    (
        { Narrowed = narrowed(Chat) },
        sender_loop(Chat)
    ;
        { Narrowed = unnarrowed(_OriginalObj) },
        io__write_string("Couldn't narrow object\n")
    ).

:- pred sender_loop(T, io__state, io__state) <= chat(T).
```

```
:- mode sender_loop(di, di, uo) is det.
sender_loop(Chat0) -->
    io__read_line_as_string(Res),
    (
        { Res = ok(String) },
        sendmessage(Chat0, Chat, String),
        sender_loop(Chat)
    ;
        { Res = error(Error) },
{ io__error_message(Error, ErrorString) },
        io__write_string(ErrorString)
    ;
        { Res = eof }
    ).
```

6.1.2 The backwards interface: CORBA calls Mercury

A Mercury object can be used to implement a CORBA object by providing an instance of the corresponding type class. For example, we can implement a `Chat` server as follows:

```
:- type our_server ---> clients(list(callback_object))
:- type callback_object ---> some [T] callback_object(T)
                             => ((callback(T), corba__object(T)).
:- instance chat(our_server) where [
    pred(sendmessage/5) is our_sendmessage,
    pred(registerclient/6) is our_registerclient,
    pred(removeclient/6) is our_removeclient
].

:- pred our_sendmessage(our_server, our_server, string,
    io__state, io__state).
:- mode our_sendmessage(di, uo, in, di, uo) is det.
our_sendmessage(clients([]), clients([]), _Msg) --> [].
our_sendmessage(clients([CBObj0 | Rest0],
        clients([CBObj | Rest]), Msg) -->
    { CBObj0 = callback_object(Obj0) },
    newmessage(Obj0, Obj, Msg),
    { CBObj = 'new callback_object'(Obj) },
    our_sendmessage(clients(Rest0), clients(Rest), Msg).
```

If a type has been made an instance of a type class which corresponds to an IDL interface, we need to be able to tell the ORB that a particular object of that type is ready to service requests. We achieve this by generating another predicate for each generated type class. The `is_ready` predicate notifies the ORB that a Mercury value is ready to service requests. For the `Chat` example, we would get:

```
:- some [T2] pred chat__is_ready(corba__object_adaptor, T1, T2,
               io__state, io__state)
        => (chat(T2), corba__object(T2)) <= chat(T1).
:- mode chat__is_ready(in, di, uo, di, uo) is det.
```

A CORBA construct called an *object adaptor* keeps track of which objects are ready to service requests. A given ORB may have more than one object adaptor, so the caller of `chat__is_ready` must choose which object adaptor the object connects to.

With this machinery, our chat server above can make its services available as simply as this:

```
:- pred start_server(corba__orb, corba__object_adaptor,
    io__state, io__state).
:- mode start_server(in, in, di, uo) is det.
start_server(Orb, ObjectAdaptor) -->
    { Server = clients([]) },
    chat__is_ready(ObjectAdaptor, Server, ReadyServer),
    corba__bind_object_name(Orb, "Chat", "Server", ReadyServer),
    corba__implementation_is_ready(ObjectAdaptor).
```

The call to `bind_object_name` registers the object with the name server. The call to `implementation_is_ready` hands control over to the event loop of the object adaptor, which listens for operations on any of the objects that have been registered with the adaptor, and invokes the corresponding methods. Each method invocation will generate a new version of the state of the object, which becomes the state of the object for the next method invocation. These states are not visible from outside the event loop; the only *visible* effect of the methods is on the I/O state. The I/O state arguments of `implementation_is_ready` (here hidden using DCG notation) capture these effects, just as the I/O state arguments of `io__write_string` do. Therefore method invocations are purely declarative for the same reasons that I/O operations are.

6.1.3 Other features of CORBA

The CORBA binding we have described above has touched upon only a few of CORBA's features. We will now briefly describe how we handle some other CORBA features.

- One IDL interface may inherit from another. For example, with:

  ```
  interface sub : super {
      ...
  };
  ```

  all of the operations available on `super` are also available on `sub`. We map IDL interface inheritance directly onto type class inheritance. In Mercury, type class inheritance is denoted by a type class declaration itself having type class constraints. In this example:

  ```
  :- typeclass sub(T) <= super(T) where [
      ...
  ].
  ```

  any member of type class `sub` must also be a member of `super`. (Neither IDL nor Mercury allows any kind of implementation inheritance.)

- The CORBA IDL includes several basic types. Many of these (e.g. `char`, `double`, `int`, `string` and `boolean`) can have their values mapped one-to-one to the values of a Mercury type, which allows us to map the IDL type directly to the Mercury type. Some other IDL types (e.g. `float` and `short`), whose values are a subset of the values of a Mercury type, can be handled by mapping them to the Mercury type and having code automatically generated by `mcorba` check their values before giving them to CORBA. The remaining types such as wide characters are harder to handle without adding new native types to the Mercury language definition. We can map them to new, abstract and non-native Mercury types that can then have the appropriate operations defined on them, but whether users will find such an arrangement convenient enough remains a matter for further experimentation.

- The CORBA IDL also includes several ways to build more complex types. Some of these (e.g. enumerations, structures, unbounded length sequences and discriminated unions in which the discriminant is an enumeration) can be mapped to Mercury in a natural manner, though some of these mappings may imply significant conversion costs. The rest (e.g. bounded length sequences and discriminated unions with integer discriminants) do not have obvious direct mappings to Mercury (or to Haskell either [20]), and finding a convenient way to represent them in Mercury is also future work.

- An IDL specification may declare sub-modules which introduce a new (nested) namespace. For each such module we create a Mercury submodule.

- Constants can also be defined in an IDL specification. For each such constant, we define a function with no input arguments but with a result corresponding to the constant.

- IDL also includes exceptions. We could handle these by mapping CORBA methods onto Mercury predicates that return either an exception indication or their actual result, (the way we handled the `narrow` operation in section 6.1.1), but we expect that most users would find these too cumbersome. An alternative solution would involve mapping CORBA exceptions to Mercury exceptions. This seems to be a promising direction to follow, although it is not clear how to implement exceptions across a programming language barrier in an efficient manner.

- CORBA allows dynamic lookup of available operations through its Dynamic Invocation Interface (DII), and dynamic creation of operations through the Dynamic Skeleton Interface (DSI). Incorporating DII and DSI into MCORBA is a topic for future work.

6.2 Implementation

The primary goals of the initial implementation were simplicity and expediency. We wanted to allow easy experimentation with the mapping from IDL to Mercury, and permit handwritten code to be substituted for machine generated code when necessary. We therefore decided that instead of writing a binding which talks to an ORB directly (using IIOP for example), we would simply build on top of an existing binding. The binding we chose to build upon is omniORB 2.0 [56], whose `omniidl2` tool generates stubs and skeletons in C++. Besides generating Mercury stubs and skeletons, `mcorba` also generates the glue that joins the Mercury stub to the C++ stub, and the Mercury skeleton to the C++ skeleton. Since omniORB 2.0 conforms to the CORBA 2.0 standard [26], which specifies a standard C++ binding, our implementation can easily be ported to other ORBs with C++ bindings.

Building a Mercury layer on top of a C++ binding is somewhat complicated by the fact that the current Mercury implementation has an interface only to C, not to C++ [1]. Therefore `mcorba` generates C functions that provide a simple interface to C++ code. Since these C functions complicate the explanation of the implementation, in the following discussion we will assume that Mercury can directly call C++. (Building a Mercury layer on top of a C binding would avoid these complications, so when a satisfactory C binding becomes available we will examine it.) We will also simplify the example code by skipping the required error checking.

6.2.1 The forwards interface: Mercury calls CORBA

The stub predicate generated by `mcorba` for a given IDL operation will invoke the corresponding method on the C++ stub object. This requires Mercury code to remember pointers to these objects. When the program calls a predicate such as `is_ready`, which returns a handle on the underlying C++ object, the actual return value will be a pointer to the C++ object. This pointer is a value in the type `corba__cppobject`, whose implementation is known only to the MCORBA system. Since values of type `corba__cppobject` are only ever returned as existentially typed arguments belonging to the `corba__object` type class, user code can never break the abstraction.

For each operation in an IDL specification, `mcorba` generates a Mercury predicate with an implementation in C++. This predicate casts the `cppobject` to a pointer to the right C++ class, invokes the requested method, and then creates the new versions of the object and the I/O state.

```
:- pred sendmessage_impl(corba__cppobject, corba__cppobject,
    corba__string, io__state, io__ state).
:- mode sendmessage_impl(di, uo, in, di, uo) is det.
:- pragma c_code(sendmessage_impl(Obj0::di, Obj::uo, Msg::in,
        IO0::di, IO::uo), [may_call_mercury], "
    Chat_ptr chat = cast_to_chat(Obj0);
    chat->SendMessage(Msg);
    Obj = cast_from_chat(chat);
    update_io(IO0, IO);
```

[1] A recent development version of the Melbourne Mercury compiler provides an interface to Microsoft's Managed C++ through the .NET Common Language Runtime (CLR) [17], but that does not help us here

```
").
```

Given one of these predicates for each method, mcorba creates an instance of the chat type class:

```
:- instance chat(corba__cppobject) where [
    pred(sendmessage/5) is sendmessage_impl,
    pred(registerclient/6) is registerclient_impl,
    pred(removeclient/6) is removeclient_impl
].
```

6.2.2 The backwards interface: CORBA calls Mercury

Any Mercury type that is an instance of the generated type class for an interface can be used as an implementation of that interface. For the ORB to use the Mercury implementation there needs to be a C++ implementation, and a layer that allows the C++ implementation to call Mercury. To do this, the C++ implementation needs to know how to invoke a given method on a given Mercury object. In C++ this would be done using a virtual function table. In Mercury, we have the type class dictionary. A Mercury predicate that manipulates an object of a given type class will have access to a type class dictionary that specifies which predicates implement the methods of the type class on that object. (A type class declaration defines the layout of dictionaries for that type class; an instance declaration specifies its contents.)

omniidl2 generates an abstract base class for every IDL interface. mcorba derives a sub-class from each such base class. The derived class simply stores the Mercury object and its type class dictionary as attributes. For our chat example, omniidl2 generates the base class _sk_Chat (providing a skeleton which can be filled in by an implementation), while mcorba generates the derived class Chat_i (which provides an implementation of that skeleton):

```
class Chat_i : public virtual _sk_Chat {
public:
    Chat_i(void *mcorba_tcdict, void *mcorba_mobj) {
        mcorba_typeclass_dict = mcorba_tcdict;
        mcorba_mercury_object = mcorba_mobj;
    }
    virtual ~Chat_i() {}
    void SendMessage(const char *);
    void RegisterClient(class CallBack *, const char *);
    void RemoveClient(class CallBack *, const char *);
```

```
private:
    void *mcorba_typeclass_dict;
    void *mcorba_mercury_object;
};
```

The methods of this C++ class call the methods of the corresponding Mercury type class. This involves looking up the relevant method in the type class dictionary, and calling it with the appropriate arguments. The Mercury type class implementation already creates code to do the lookup and parameter passing, so we simply export this code using the `pragma export` directive to allow it to be called from C (or, in our case, C++):

```
:- pragma export(sendmessage(di, uo, in, di, uo),
    "chat__sendmessage_method").
```

The implementation of the C++ method can simply call the exported C function with the stored type class dictionary, the original value of the object, and the input arguments of the method. The function will return the updated value of the object and the output arguments of the method through reference parameters.

```
void Chat_i::SendMessage(const char *msg)
{
    chat__sendmessage_method(mcorba_typeclass_dict,
        mcorba_mercury_object, &mcorba_mercury_object, msg);
}
```

Note that the C++ code does not know anything about I/O states, and that the two arguments of type `io__state` are missing. This is possible because Mercury uses values of this type only to enforce referential transparency; they do not actually have any meaningful run time representation.

Each invocation of the `mcorba`-generated `chat__is_ready` predicate creates an instance of the `Chat_i` class and calls the C++ `_sk_Chat::_obj_is_ready` method to register it with the ORB.

6.3 Remarks and related work

The main contribution of this chapter is showing how IDL interfaces can be mapped onto type classes. Since existential types are an integral part of this approach but are not (yet) part of the standard language specification for Haskell, H/Direct [83] (the Haskell binding for COM) could not take this approach, and mapped IDL interfaces onto types instead. The advantages of our approach are that:

- Type classes provide a significantly more natural means of modelling CORBA interfaces. In particular, it is very easy for a program to provide more than one implementation of an interface by providing more than one `instance` declaration for it.
- Inheritance of one interface by another comes "for free". There is no need to explicitly convert from an inherited-by to an inherited-to interface; the type class constraint mechanism handles this automatically.

Our approach should be easily adaptable to other logic and/or functional languages that have both type classes and existential types, as we expect Haskell 2 will when it arrives.

Our approach also has an advantage over C++ bindings. When an argument of a CORBA operation is another CORBA object, C++ bindings check that the actual parameter satisfies the right interface, but do not check that it has been registered with an ORB. The type class method generated by `mcorba` for the operation requires the corresponding argument to be a member of the `corba__object` type class, which guarantees that any attempt to pass an unregistered object as the actual parameter will be caught at compile time.

We have found our approach to the implementation, which is the use of a thin layer on top of an existing C++ binding, to be very effective. It can be implemented relatively quickly using a simple foreign language interface, and the resulting binding should be easily portable among different ORB implementations for C++.

Chapter 7

Conclusion

We have described an expressive type system for the higher order logic programming language Mercury. Throughout this presentation, our goal has been to precisely describe the type system that we have implemented in the Melbourne Mercury implementation. We have used typing judgements and typing rules as a means of formally specifying the type system. While our presentation has been practical in nature, we believe that the description we have given is precise enough to be amenable to a more theoretical analysis.

The aim of the design of our type system has been to support the construction of large system systems by allowing the programmer to:

- take advantage of the object oriented design techniques which are likely to lead to robust and maintainable programs
- interoperate with component software frameworks (allowing both the construction of components and the use of existing components) in a manner that is natural and not overly complex

By providing a type class mechanism which allows the programmer to directly express the concept of an abstract class or component interface, and by providing existential typing as a means of encapsulating object implementations, we allow the programmer to do both these things.

We conclude this thesis by evaluating the type system we have presented with respect to the five (informal) metrics for the engineering properties of object oriented languages proposed by Cardelli [6].

- *Economy of execution.*
  The Melbourne Mercury implementation uses a highly efficient execution algorithm [100]. Using type classes as an abstraction mechanism does introduce some small run time overhead. However, techniques such as

partial evaluation have successfully been applied in Haskell implementations to remove such of this overhead [45]. Similarly, the Melbourne Mercury implementation applies type specialisation techniques [105] to replace dynamic dispatch with static dispatch where possible. Existential typing also introduces execution overhead by requiring type information and type class dictionaries to be stored explicitly inside data structures. However, the price of this overhead is only paid in situations where the flexibility it affords is explicitly required. This is quite a different situation to traditional object oriented languages, where all data is packaged at run time inside objects, effectively requiring run time type information and virtual method tables to be stored more often than is necessary.

In addition, Mercury's static correctness properties help the programmer to pinpoint errors during development. By enforcing these static correctness properties, whole classes of errors simply cannot occur. The type system we have described in this thesis provides significant expressiveness while allowing the rest of Mercury's static correctness properties to be enforced and while itself remaining statically verifiable.

- *Economy of compilation.*
  Type classes and existential types do not compromise separate compilation in any way. While sophisticated inter-module optimisation techniques are able to significantly improve the execution time and space usage of Mercury programs, Mercury modules can always be compiled separately during development. This is not always possible with poorly designed object oriented programming languages, where recompilation of superclasses can be necessary during compilation of sub-classes.

- *Economy of small-scale development.*
  Logic programming languages have traditionally held advantages over their procedural counterparts due to their high level nature and simple semantics, along with features such as higher order programming and the built in search mechanism. Type classes and existential types do not compromise these features as at all.

- *Economy of large-scale development.*
  The true value of type classes and existential types is in the mechanisms they provide to facilitate programming in the large. Traditional logic programming languages have not allowed the programmer to implement the kinds of object oriented designs that increase reusability and maintainability of large scale software systems. The type system we have described

in this thesis does precisely this. Type classes and existential types allow the programmer to implement object oriented designs in Mercury in a straightforward manner. Moreover, our type system encourages maintainable and reusable object oriented designs.

Similarly, type classes and existential types provide a direct model of the kind of component interfaces used by modern component frameworks such as CORBA [26].

- *Economy of language features.*
  The design of declarative languages such as Mercury has traditionally been based around the application of a small set of orthogonal language mechanisms. In this thesis, we have introduced five syntactic entities to Mercury:

  - `typeclass` declarations, which correspond to the definition of abstract class interfaces.
  - `instance` declarations, which provide an interface inheritance mechanism.
  - Type class constraints, which allow predicates to depend on the abstract interfaces described by `typeclass` declarations rather than on particular implementations.
  - Existentially quantified type variables in the definition of function symbols, which allow for the encapsulation of an object's implementation inside a data structure.
  - Existentially quantified type variables in predicate signatures, which allow for the encapsulation of an object's implementation when returned by a predicate.

  These five mechanisms are quite orthogonal, and we believe that they represent the minimal set of features required to provide the expressiveness that they add to the language.

Our performance on these metrics leads us to believe that the type system we have designed and implemented for Mercury provides excellent support for the development and maintenance of large scale software systems.

Bibliography

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition edition, 2000.

[2] Hendrik Barendregt. *The lambda calculus: its syntax and semantics*. Elsevier Science Publishers, 1984.

[3] Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, February 1996.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[5] Nigel P. Capper, Roger J. Colgate, Jim C. Hunter, and Martin F. James. The impact of object-oriented technology on software quality: Three case histories. *IBM Systems Journal*, 33(1):131–157, 1994.

[6] Luca Cardelli. Bad engineering properties of object-oriented languages (position paper). *ACM Computing Surveys*, 28(4es), 1996. Article 150.

[7] Luca Cardelli. Type systems. In Allen Tucker, editor, *Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[9] Weidong Chen and Michael Kifer. Polymorphic types in higher-order logic programming. Technical Report 93/20, Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY, USA, December 1993.

[10] Weidong Chen and Michael Kifer. Sorted HiLog: Sorts in higher-order logic data languages. In *Proceedings of ICDT 1995*, pages 252–265, 1995.

[11] Philip Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 5, pages 157–188. MIT Press, Cambridge, Massachusetts, 1992.

[12] Andrew Davison. A survey of logic programming-based object-oriented languages. Technical Report 92/3, Department of Computer Science, University of Melbourne, Melbourne, Australia, January 1992.

[13] Bart Demoen, María García de la Banda, and Peter J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228. Springer Verlag, January 1999.

[14] Peter J. Denning. Sacrificing the calf of flexibility on the altar of reliability. In *Proceedings of the Second International Conference on Software Engineering*, pages 384–386, October 1976.

[15] Pierre Deransart and Jan Maluszynski. Towards soft typing for CLP. In F. Fages, editor, *Proceedings of JICSLP '98 post conference workshop on Types for Constraint Logic Programming*, June 1998.

[16] Pierre Deransart and Jan-Georg Smaus. Well-typed programs are not wrong. In Herbert Kuchen and Kazunori Ueda, editors, *Proceedings of FLOPS '2001, 5th International Symposium of Functional and Logic Programming*, March 2001.

[17] Tyson Dowd, Fergus Henderson, and Peter Ross. Compiling Mercury to the .NET Common Language Runtime. In *Proceedings of BABEL '01, the first workshop on multi-language infrastructure and interoperability*, Firenze, Italy, September 2001. Electronic Notes in Theoretical Computer Science. To appear.

[18] Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the International Conference on the Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 224–243. Springer Verlag, September 1999.

[19] ECMA. *C♯ Language Specification Draft ECMA Standard*, October 2000.

[20] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *Proceedings of the 1998 International Conference on Functional Programming*, pages 153–162, September 1998.

[21] Kathleen Fisher and John C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1:189–220, 1996. Preliminary version appeared in *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, 844–885.

[22] David Flanagan. *Java in a nutshell.* O'Reilly, 1999.

[23] Thom W. Früwirth. Towards a fully polymorphic type system for Prolog. In *Proceedings of the Workshop on Types in Logic Programming at the North American Conference on Logic Programming (NACLP '89)*, October 1989.

[24] J. H. Gallier. *Logic for Computer Science.* John Wiley and Sons, 1987.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[26] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., July 1996.

[27] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256. Springer Verlag, 1994.

[28] Michael Hanus. Polymorphic higher-order programming in Prolog. In G. Levi and M. Martelli, editors, *Proceedings of 6th International Conference on Logic Programming*, pages 382–397, 1989.

[29] The Haskell committee. *The Haskell 98 Report.* Available from `http://www.haskell.org/`.

[30] Nevin Heintze and Joxan Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 4, pages 141–156. MIT Press, Cambridge, Massachusetts, 1992.

[31] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996. A more recent version of this is available via `http://www.cs.mu.oz.au/mercury/`.

[32] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the mercury compiler. In *Proceedings of the Australian Computer Science Conference*, volume 1, pages 337–346, January 1996.

[33] Fritz Henglein and Konstantin Läufer. Programming with structures, functions, and objects. In *Proceedings of the XVII Latin American Informatics Conference (PANEL '91)*, pages 333–352, July 1991.

[34] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 1, pages 1–62. MIT Press, Cambridge, Massachusetts, 1992.

[35] Pat M. Hill and John W. Lloyd. The Gödel report. Technical Report 91-02, Department of Computer Science, University of Bristol, Bristol, England, 1991.

[36] Trevor Hopkins and Bernard Horan. *Smalltalk - An introduction to application development using Visual Works.* Prentice Hall, 1995.

[37] John Hughes and Jan Sparud. Haskell++: an object-oriented extension of Haskell. In *Electronic Proceedings of the 1995 Haskell Workshop*, Orlando, Florida, USA, 1995.

[38] ISO. Ada reference manual. Technical Report ISO/IEC 8652:1995(E), International Organisation for Standardisation, Geneva, Switzerland, 1995.

[39] ISO. ISO Prolog Standard. Technical Report Technical Report ISO/IEC DIS 13211-1:1995(E), International Organisation for Standardisation, Geneva, Switzerland, 1995.

[40] David Jeffery, Tyson Dowd, and Zoltan Somogyi. MCORBA: A CORBA binding for Mercury. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 211–227. Springer Verlag, January 1999.

[41] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical Report 98/13, Department of Computer Science and

Software Engineering, University of Melbourne, Melbourne, Australia, September 1998.

[42] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. In J. Edwards, editor, *Proc. Twenty-Third Australasian Computer Science Conf.*, volume 22(1) of *Australian Computer Science Communications*, pages 128–135. IEEE Computer Society Press, January 2000.

[43] Mark P. Jones. A theory of qualified types. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer Verlag, February 1992.

[44] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Conference on Functional Programming Languages and Computer Architecture*, pages 52–61, June 1993.

[45] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* ACM SIGPLAN, June 1994.

[46] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, May 1994.

[47] Mark P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, Nov 1994.

[48] Mark P. Jones. Type classes and functional dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science.* Springer Verlag, March 2000.

[49] R. Kowalski. Predicate logic as a programming language. In *Proceedings of the IFIP 1974 Congress, European Symposium on Programming*, pages 569–574, 1974.

[50] T.K. Lakshman and Uday S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In Kazunori Ueda, editor, *Proceedings of 1991 International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.

[51] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, New York University, July 1992. Available as Technical Re-

port 622, December 1992, from New York University, Department of Computer Science.

[52] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[53] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proc. ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[54] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994.

[55] John Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[56] Sai-Lai Lo. *The omniORB2 User's Guide.* Olivetti and Oracle Research Laboratory, March 1997.

[57] Lunjin Lu. Polymorphic type analysis of logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[58] Robert C. Martin. The dependency inversion principle. *C++ Report*, May 1996.

[59] Robert C. Martin. The interface segregation principle. *C++ Report*, August 1996.

[60] Robert C. Martin. The Liskov substitution principle. *C++ Report*, March 1996.

[61] Robert C. Martin. The open closed principle. *C++ Report*, January 1996.

[62] Robert C. Martin. Design principles and design patterns. Technical report, www.objectmentor.com, 2000.

[63] Michel Mauny and Fançois Pottier. An implementation of CAML-Light with existential types. Technical Report 2183, Institut National de Recherche en Informatique et en Automatique, France, October 1993.

[64] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[65] Gregor Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, FernUniversität Hagen, June 1996.

[66] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[67] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[68] Prakeesh Mishra. Towards a theory of types in Prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, 1984.

[69] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[70] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[71] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 1, pages 810–827. MIT Press, August 1988.

[72] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, pages 499–590. Oxford University Press, 1998.

[73] Gopalan Nadathur and Frank Pfenning. Types in higher-order logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 9, pages 245–284. MIT Press, Cambridge, Massachusetts, 1992.

[74] Lee Naish. Types and the intended meaning of logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 6, pages 189–216. MIT Press, Cambridge, Massachusetts, 1992.

[75] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, January 1993.

[76] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 202–219, Eindhoven, The Netherlands, June 1991.

[77] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[78] Martin Odersky. Objects and subtyping in a functional perspective. Technical Report RC 16423, IBM Research, Thomas J. Watson Research Center, January 1991.

[79] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[80] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[81] S. Peyton Jones, C.V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, 1993.

[82] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag, June 1997.

[83] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of the Fifth International Conference on Software Reuse*, June 1998.

[84] Shari Lawrence Pfleeger. *Software Engineering, Theory and Practice*. Prentice-Hall, 2001.

[85] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Preliminary version in Principles of Programming Languages (POPL), 1993.

[86] Wolfgang Pree. *Design patterns for object-oriented software development*. Addison-Wesley, 1995.

[87] Prolog Development Center. *Visual Prolog*. Available from `http://vip.pdc.dk/`.

[88] Dale Rogerson. *Inside COM*. Microsoft Press, 1998.

[89] Y. Rouzaud and L. Nguyen-Phuong. Integrating modes and subtypes into a Prolog type-checker. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 85–97. The MIT Press, 1992.

[90] P. Van Roy. A useful extension to Prolog's definite clause grammar notation. *SIGPLAN Notices*, 24(11):132–134, 1989.

[91] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, December 1998.

[92] Stephen R. Schach. *Classical and Object-Oriented Software Engineering.* McGraw-Hill International Editions, 1999.

[93] Michael L. Scott. *Programming language pragmatics.* Morgan Kaufman Publishers, 2000.

[94] Jan-Georg Smaus, Francois Fages, and Pierre Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In S. Kapoor and S. Prasad, editors, *Proceedings of FST and TCS '00*, pages 214–226. Springer-Verlag, December 2000.

[95] Gert Smolka. *Logic programming over polymorphically order-sorted types.* PhD thesis, Universität Kaiserslautern, May 1989.

[96] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications OOPSLA '86*, pages 38–45, Portland, OR, 1986. ACM Press.

[97] Ian Sommerville. *Software Engineering.* Addison-Wesley, 6th edition, 2001.

[98] Zoltan Somogyi. A system of precise modes for logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 769–787, Melbourne,Australia, May 1987.

[99] Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, pages 35–49, November 1999.

[100] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[101] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 2nd edition edition, 1994.

[102] Bjarne Stroustrup. *The C++ Programming Language (Special Edition).* Addison-Wesley, 2000.

[103] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, June 1995.

[104] Clemens Szyperski. *Component software: beyond object-oriented programming.* ACM Press, 1997.

[105] Simon Taylor. Optimisation of Mercury programs. Honours report, Department of Computer Science, The University of Melbourne, available from `http://www.cs.mu.oz.au/mercury/`, November 1998.

[106] Pascal van Hentenryck, Agostino Cortesi, and Baudouin le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, March 1995.

[107] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[108] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, January 1989.

[109] Christoph Wille. *Presenting C♯*. Sams Publishing, 2000.

[110] Eyal Yardeni, Thom Früwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 2, pages 63–90. MIT Press, Cambridge, Massachusetts, 1992.

[111] C. Zaniolo. Object-oriented programming in Prolog. In *Proceedings of the First IEEE Symposium on Logic Programming*, pages 265–270, 1984.

[112] Justin Zobel. Derivation of polymorphic types for Prolog programs. In *Proceedings of the 4th Internation Conference on Logic Programming*, pages 817–838, May 1987.