Parametric Overloading in ML

by

Charles C. Krasic

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1997

©Charles C. Krasic 1997

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

A prototype implementation of Oasis, including a compiler and top-level interpreter, was constructed from the Caml-Light system. An overview of the design of the prototype will be given. The prototype helps to investigate and demonstrate the effectiveness of Oasis extensions. A key aspect of parametric overloading in Oasis is that all let bound values are overloaded; there is no new language construct for introducing overloaded values. The feasibility of this approach is affirmed by the successful bootstrapping of the Caml-Light system with Oasis. The assumption that all let bound values are overloaded required less than a dozen of the thousands of lines of Caml-Light code to change.

Acknowledgements

I thank my supervisor professor Gordon V. Cormack, for his input and great patience in helping me complete this thesis. Also, professor John Ophel and professor Dominic Duggan for providing very generous assistance. I am grateful to Dave Mason, Gordon Vreugdenhil, and the many others at Waterloo for all of the stimulating discussions and inspirational diversions. Finally, I thank my family for their love and support.

Contents

List of Figures

Chapter 1

Introduction

Type systems play an important role in programming language design; they have a direct impact on the level of reliability, performance, and expressiveness afforded by programming languages. The type system of the ML programming language is widely regarded as one of the most advanced type systems implemented in a general purpose programming language. In this thesis, I describe the design and implementation of a type system that extends the classic ML type system with parametric overloading. I call the resulting language Oasis. Parametric overloading improves the ML type system on two fronts. First, it provides a general framework for the definition and use of overloaded symbols; thus allowing more concise and intuitive programs where overloading style is natural. Second, it defines a mechanism of deferred overload selection that expands ML's support for reuse.

A full prototype compiler for Oasis was constructed from the Caml Light system. As with Caml Light, the majority of Oasis is written in ML and hence the prototype system self compiles. The self compilation of Oasis serves as empirical evidence that Oasis is a conservative extension of ML.

1.1 Thesis Outline

In the remainder of this chapter, I will introduce ML, overloading, and parametric overloading with a few short examples. I will also explain some software engineering issues that motivated various aspects of this thesis. Chapter 2 will discuss related work. In Chapter 3, I will give a more detailed tutorial on ML and Oasis. The tutorial will work through a number of examples that illustrate the mechanics of parametric overloading and demonstrate its expressive capability. Chapter 3 will also relate the major technical issues that occurred during Oasis's development. Chapter 4 presents a formal specification of the Oasis type system. Chapter 5 details the Oasis algorithms for type inference and overload resolution. Finally, chapter 6 contains concluding observations and proposes possible future work.

1.2 The ML Type System

The type system of ML[21], known as the Hindley-Milner type system, has two stand-out features. First, it supports type inference; type inference allows the programmer to write programs without explicit type annotations, while retaining full static type checking. Consider the following ML code fragment.

```
let a = 5 in
let b = "hello" in
let f = fun x -> if x > 10 then true else false in
let c = f a in
let d = f b ;;
```

The first two declarations are straightforward. We note that ML would infer the types of `a` and `b` to be `int` and `string` as is obvious from their definitions. From the declaration of `f`,the ML type checker would infer that `f` is a function whose argument `x` must be an integer and result must be `boolean`. The declaration of `c` would succeed with `c` assigned type `boolean`, but the the declaration of `d` would fail because `b` is a `string` which conflicts with the expected argument type for `f`. In most typed languages, either the programmer would have to explicitly annotate the program with types or some checking of types would not occur until runtime. ML's type inference mechanism allows the programmer the luxury of full static type checking without the burden of writing types explicitly.

The second major feature of the ML type system is parametric polymorphism. A type system supports polymorphism if it allows objects to have more than one type. The canonical example of a polymorphic function is the identity function:

```
let id = fun x -> x;;
```

The type assigned by ML to this function will be `'a -> 'a`. Here, `'a` is a type variable, meaning that it can take on many possible values. We say `id` is a polymorphic function, whose argument can be of any type. The additional constraint implicit in this type is that the result of the function has the same type as the argument. Other programming languages, for example Ada, use the term *generic* to describe functions with parametric polymorphic types. Among these languages, the ML approach is eminent because it requires no explicit effort on the part of the programmer in writing generic code. Hindley-Milner infers most generic types from programs; the concept of *principal types* formally defines this feature[4]. So ML inference not only infers information necessary for basic type correctness, but it also automatically observes generality.

1.3 Overloading

Overloading refers to the ability to have multiple values, usually functions, bound to the same symbol. Overloaded values are identified first by their symbol and then further distinguished according to their type. The latter step is referred to overload resolution.

Overloading is common in mathematics; for example, addition operations are represented by the `+` symbol:

```
  2 + 2
1.5 + 3.1
```

Adding scalars is a fundamentally different operation from adding real values, but mathematical convention prescribes using the same symbol, `+`, for both operations. We say that `+` is overloaded. Overloading allows writing more concise expressions, at the expense of some formality. If we didn't want to use overloading, we might choose to invent new names to separate scalar addition from real addition:

```
  2 + 2
1.5 # 3.1
```

Here we use `+` for scalar addition, and `#` for real addition. Inventing new names is tiresome, and further these distinct names fail to capture the notion that the operations in question are very similar. An alternative technique would be to use explicitly qualified versions of the same symbol[1]:

```
  2 int.+ 2
1.5 real.+ 3.1
```

[1] A mathematician would probably use subscripts here instead of prefixes

There are no rules for when overloading is appropriate; the issue is one of style. The longevity of overloading style in the domain of mathematics provides good evidence that the alternatives do not obviate the utility of overloading.

So what happens if we decide to add overloading to our favorite programming language? There are at least a few nasty surprises. Many problems that are instinctive for humans, such as resolving overloaded symbols, can be quite difficult to codify in computer algorithms. The overload resolution problem for a sufficiently general overloading facility is probably impractical. This phenomenon is pronounced in the context of ML because Hindley-Milner type checking already teeters at the edge of intractability. Aside from the inherent complexity of supporting overloading, other problems arise from interactions between overloading and other programming language facilities. Especially problematic are the interactions with advanced module systems, automatic type coercions, and inheritance facilities. Consequently, languages that do support overloading (e.g. Ada, C++, SML) do so with restrictions.

1.4 Parametric Overloading

The discussion above described overloading as a feature that allows a occasionally convenient programming style. This might not seem too exciting; but something interesting happens when overloading is combined with Hindley-Milner typing. The notions of type inference and parametric polymorphism naturally lead to the ability to do something called deferred overload selection. Consider the following Oasis declaration.

```
#let double x = x + x;;
```

```
double : [?+:'a -> 'a -> 'a].'a -> 'a = <fun>
#
```

In this example, we define a function `double` which takes its argument `x` and doubles it using the overloaded function `+`. The example demonstrates an extension to the basic idea of parametric polymorphism, or generic, described earlier. Suppose that `+` is defined both for scalar (integer) values, and for real values. Then both of following expressions would be valid in Oasis:

```
double 2
double 2.5
```

The type of double is `[?+: 'a -> 'a -> 'a].'a -> 'a`, read a function from type variable `'a` to `'a`. In addition, this type specifies that application of double requires an overload parameter `+` having type`'a -> 'a -> 'a`. The intent is that this `+` will be supplied automatically at each call-site as appropriate.

Another important feature of parametric overloading in Oasis is that there is no new syntax needed for declaring overloaded symbols. Instead, Oasis automatically decides which symbols are candidates for overloading. Briefly, Oasis usually treats variables declared with let as overloaded, while other variables such as function parameters are not overloaded. With this approach, as with ML polymorphism, the programmer often gets overloading for free in Oasis.

Deferred overload resolution expands the notion of generic in ML while it continues, in the spirit of Hindley-Milner, to relieve the programmer of the explicit annotation burden. In Chapter 3, I will present a number of examples that further illustrate the potential of deferred overload selection.

1.5 Objectives for Programming Languages

The remainder of this thesis will focus on ML, its type system, overloading, and the implementation of parametric overloading. Before proceeding with these wonderful things, it is worth noting some of the bigger picture relating to programming language design and its role in advancing the state of software engineering technology. In particular, there are three concepts that motivated much of the direction of this thesis: reliability, reuse, and rapid application development.

1.5.1 Type Systems for Reliability

Building reliable software is difficult and costly. One way of judging reliability is to count the number of known bugs. With this approach, we are assuming that reliability is largely a matter of perception. If testing reveals no bugs, and users don't complain too much, then the software is considered stable. Real world software development often operates this way. Real world software breaks all the time. The problem is largely one of scale. As software systems become larger and more complex, it becomes more expensive to try to do a good job of testing. We can try to build tools to help test, but testing all cases automatically is not possible: most programs allow an infinite number of input possibilities.

The inadequacy of testing naturally leads to the view that software should be proven correct relative to some reference point. The formal methods community advocates writing specifications in well understood semantic frameworks, and then through one or more steps of reification, the specifications are transformed into implementations. In this way, software can be proven correct relative to a specification. The specification represents the point of contention; nothing stops the specification from being wrong. However, the rational of this approach is that specifications can be written at much higher levels of abstraction than implementations, so that users can be better equipped to relate their perception of what happens in a system to what is really happening. Building large systems using formal methods is not yet practical and is very rare in the commercial world, but in the long run, formal methods represent the only hope of bug free software.

The problem with formal methods is that proving correctness is hard. In particular, it is not a task that can be automated: proving that two programs, say a specification and an implementation, are equivalent is a canonical example of an undecidable problem. While the formal methods community do have tools to assist humans in constructing proofs, they are too time and skill intensive for most software efforts.

At one end of the spectrum, you establish correctness by running programs on test cases. At the other end, you prove correctness before you ever run the program. This is where type systems come in: they provide a sort of middle ground between testing and proving. A type system allows for the construction of a partial proof of correctness, in a manner that is, for the most part, fully automated. The amount of proof provided can vary from type system to type system. For example, the type system of ANSI C, if used properly, can establish that a function is called with the right number and right type of arguments. On the other hand, the type system of ANSI C can not establish whether a program will de-reference a pointer to an invalid address. A program that does so, and C programs so often do, will have undefined behavior: it could be benign, it could corrupt data, it could be detected by the operating system causing the program to be terminated, or if the operating system lacks suitable protection, it could bring the machine down. A substantial proportion of the programs in the world today are written in C.

A type system is strong if it establishes that the program will behave as predicted by the language semantics. Any errors that occur must be within the context of the language semantics. If the semantics of the language allow errors to occur at runtime, albeit in a predictable fashion, then it is dynamically typed. For example, a Scheme program that attempts to assign a floating point value to a integer variable will trigger an appropriate scheme type error at runtime. This differs significantly from the C example above, in that the nature of the error is well defined according to the language semantics and does not depend on random factors, or the level of protection offered by the operating system.

A language is statically typed if all type errors are detected at compile time. ML is a statically typed language. The strength of a static type system is that it effectively performs a partial proof of correctness on the program, eliminating the need for a whole class of testing. In contrast, a dynamically typed system still depends on testing to determine whether type errors can occur. Static typing may have performance advantages too, since the proof removes the need for many runtime type checks. Static typing does not come without a price: static type checking requires that the programming language be restricted in order for the type checking problem to be practical[2]. For example, ML does not allow mutually recursive functions to be defined across module boundaries, precisely because it would disallow static type checking. Advocates of ML, such as myself, would assert that ML is in fact very expressive, and that its type system is one of the most advanced in terms of striking a balance between expressiveness of the language and power of the type system.

[2]This point is applies equally well to any form of strong typing, as dynamic typing does not come for free either.

1.5.2 Software Reuse

Re-inventing the wheel is expensive. We do it all the time in software development.

So far the most substantial movement to try to address the reuse problem, has been centered around object-oriented programming (OOP) technologies. There are now many OOP languages, and most languages that started out before OOP arrived, have adopted new features to provide for OOP. A key aspect of OOP enabled programming languages is support for type systems with polymorphism[3]. An object has polymorphic type if it may have more than one type. The sort of polymorphism provided by OOP is usually referred to as inheritance or inclusion polymorphism. The types allowed for an object is defined by the inheritance hierarchy.

Parametric overloading is cross of two other well known forms of polymorphism: parametric polymorphism and overloading polymorphism. Why do we need a new kind of polymorphism? The OOP community have focussed a great deal of attention to reuse through inheritance. The demand for a solution to reuse is widespread, and so the OOP community has been extremely successful in attracting both academic and industrial interest. Functional programming languages, such as ML, have attracted considerable academic interest, but have failed to generate any significant industrial interest. Functional language advocates have to take a cue from the industrial success of OOP: emphasize reuse. Parametric overloading provides an alternative mechanism to inheritance for reuse.

1.5.3 Rapid Application Development

While progress on software reliability and re-use has been discouraging, one area where there has been noticeable and tangible progress is the realm of rapid application development (RAD). The advent of systems like PowerBuilder, Visual Basic, various 4GL's (so called fourth generation languages), all leverage the bundling of various tools into the programming language environment to enable very quick turnaround times. In contrast to systems based on the traditional edit-compile-link cycle, RAD systems usually integrate tools such as graphical user interface (GUI) builders into some form of interpreter-based development environment. In many contexts, interpreter based systems have been rejected out of hand as being to slow compared to compilation, but RADs have demonstrated that interpreters are a valid alternative in many scenarios–especially RADs for GUIs. Most RADs have the capability to allow embedding of externally compiled code (usually C) in much the same way that many C compilers allow strategic embedding of assembly code. Thus one could advocate the use of RADs, instead of C, as vehicles for constructing entire systems, just as C is used instead of assembly language. I think it is worth noting that RAD systems derive most of their benefit from some fairly straightforward software engineering decisions, rather than from any sort of deep thinking that seems necessary to solve problems of reliability and reuse.

A problem with most RADs is the weak programming languages that form their foundation. It's easy to get something working with a typical RAD, but once the prototype grows, the weaknesses of the language show. Few strong languages have implementations which provide the rich environment of a RAD though. As with the OOP and reuse, I think advocates of functional programming should leverage the success of RADs. The implementation described in this thesis is based on the Caml-Light ML system. The Caml-Light environment includes good RAD tools within the context of an ML implementation.

Chapter 2

History and Related Work

In this chapter I will describe the origins and history of Oasis parametric overloading, along with some discussion of other approaches to parametric overloading.

2.1 Overloading at Waterloo

There has been a great deal of work done concerning parametric overloading by various members of the programming language group at the University of Waterloo.

Cormack and Wright described a system with implicit parameters; these parameters have explicit typing at the definition site and inference of types at the call site [10]. This system was the motivation for watML, a combination of the implicit parameter idea with ML. Ophel constructed the prototype of watML based on Anu ML and introduced the notion of liberal resolution [25]. Other work at Waterloo combined the similar mechanisms with languages such as C [5, 39] and Ada[28].

In 1992, I implemented a second prototype version of watML, called WatML-NJ, based on the SML of New Jersey compiler [2]. Later work on parametric overloading was concerned with kind based type systems [7]. The WatML-NJ prototype was intended to be preliminary step towards implementing a true implementation of kind based parametric overloading. As it turned out, the WatML-NJ prototype was fairly effective without kinds, and the experience raised several issues that inspired me to re-develop watML yet again as Oasis. Another key factor motivating Oasis was the advent of the Caml-Light system [18], which I believed to be a much better platform for experimentation than SML of New Jersey.

2.2 Extensional Polymorphism

Another interesting approach to overloading in Caml-Light is the recent work on extensional polymorphism [27]. Mechanically this system is distinguished from Oasis in that it works by passing parameters which encode type rather than the overloaded operations themselves. An explicit mechanism for declaring generics allows users to introduce new overloaded definitions. One advantage of this approach is that it allows for easy accommodation of overlap and default instances. A disadvantage is that is more difficult to adapt to separate compilation.

2.3 Haskell

There is a large body of material concerning overloading in Haskell. The seminal reference on Haskell overloading is Wadler and Blott [35]. As described in [6], there are significant differences, because of the choice of resolution strategies, between Oasis and various strategies for overloading in Haskell. In general, the Haskell approaches could be characterized by a very formal approach to language semantics, in contrast to my approach which tended more towards experimentation. Mark

Jones has given very in-depth treatment to overloading [15]. Recent work by Oderskey, Wadler and Wehr has proposed an alternative [22] for overloading in Haskell. Parametric overloading in Oasis has far fewer semantic restrictions than these proposals, but as such Oasis must admit the possibility of non-termination in the type checker.

2.4 Non-overloading Systems

There are a number of other works, which do not directly relate to overloading, but nonetheless relate to Oasis.

2.4.1 Ohori

Ohori has done work on polymorphic record calculi for ML [23]. This work allows records in ML to be extensible, in a manner that resembles aspects of Object-Oriented programming. The compilation technique for this system greatly resembles parametric overloading, in that implicit parameters are inferred and solved in a manner similar to overload parameters. Jones thesis [15] has material on extensible records also. More recently Ohori has extended his work to cover possible ways to remove boxing from data representations based on a implicit parameter style mechanism [24]. Both of Ohori's proposals seem like natural future extensions for Oasis.

2.4.2 Effect Systems

There has been various work on *Effect Systems*, at first concerning issues of proper treatment of references in an ML language [29, 30], and more recently the replacement of garbage collection by static memory management [32, 1]. The semantics for these systems appear to work in a manner that resembles the treatment of overloading in Oasis, as well as Mark Jones system for qualified types.

Chapter 3

Language Overview

This chapter will provide an informal overview of Oasis. I assume the reader is familiar with programming, not necessarily in ML, and has had some exposure to functional programming and typed languages.

The following section will provide a short overview of the Caml-Light dialect of ML, which should be useful for readers unfamiliar with ML, or familiar with the Standard ML (SML) dialect but not Caml-Light. The overview will cover only as much of Caml-Light as necessary for understanding the rest of this thesis. While some aspects of Caml-Light differ significantly from core SML, the subset of Caml-Light presented here differs from the corresponding parts of SML only in syntax.

Readers familiar with Caml-Light can safely skip the following section.

3.1 Introduction to Caml-Light

Caml-Light provides both an interpreter, and batch compiler. The interpreter accepts declarations and expressions one at a time in standard read-eval-print loop style. Alternatively, the batch compiler supports an edit-compile-link style development cycle. In either case, programs are processed as a series of implementation phrases, either expressions and declarations:

```
impl_phrase ::= expr;;
              | decl;;
```

Top level phrases are always terminated by double semi-colons.

3.1.1 Expressions

The basic grammar of Caml-Light expressions are as follows:

```
expr ::= const
      |  id
      |  constr expr
      |  let pat = expr in expr
      |  let rec pat = expr in expr
      |  fun matchlist
      |  match matchlist
      |  expr expr
      |  expr; expr
      |  id <- expr
      |  try expr with match
      |  raise id
      |  ( expr [, expr ...] )
      |  listexpr
      |  ...
```

Constant

Constant expressions include integers, floating point numbers, and strings, in the usual formats. There is also the unit value `()`, and the nil list `[]`.

Variable

Identifiers can be of arbitrary length, and may consist of characters, digits and some punctuation characters (underscore, single quote, dash, etc.).

Constructor

ML has the notion of data constructors. A data constructor without an argument is like an enumerated type. A data constructor is like a variant record, where the constructor represents the discrimination tag.

Let

The let construct allows expressions to be bound to variables within other expressions. The second form of let adds the `rec` keyword to allow recursive definitions. The binding can also decompose the bound expression into several variables using pattern matching. Here is a partial syntax for Caml-Light patterns:

```
pat ::= id
     | ( pat [, pat] )
     | ( pat as id )
     | ( constr [ pat ] )
     | _
     | ...
```

The most basic pattern is the simple variable pattern.

Tuple patterns match the subexpressions of a tuple against a series of sub patterns. The `as` keyword allows multiple bindings of the same expression. For example, you can write:

```
let ((x,y) as point) = (5,6) in ...
```

Here, `x` is bound to `5`, `y` is bound to `6`, and point is bound to tuple `(5,6)`.

A constructor pattern allows decomposition of constructor expressions; it is used mostly within function definitions and match expressions.

A pattern consisting of a single `_` represents the *wildcard* pattern; it matches any expression without binding any variables. The wildcard is often useful for ignoring parts of a data structure when not all components are required.

Function

Functions in ML allow pattern matching according to the following syntax:

```
matchlist ::= pat -> expr [ | matchlist ... ]
```

Note, the `|` here is a token and not a grammar symbol. When the match list for a function contains multiple matches, it behaves like a case statement. The patterns are sequentially checked against the function argument; if a pattern matches the corresponding expression is evaluated, otherwise the next pattern is checked.

A function application is written `expr expr`; for example, ML evaluates

```
(fun x -> x + 1) 1
```

to 2.

Imperative Features

ML is not purely functional, variables can be declared to be mutable and assigned to. An expression sequence `expr; expr` evaluates the first expression discarding the result, and then returns the value of the second expression. A Caml-Light assignment expression has the form `id <- expr`. Caml-Light also supports the SML notation for `ref` values.

Exceptions

ML has an exception mechanism. Exceptions are declared like data constructors. The `try expr with match` construct wraps `expr` with an exception handler defined by the `match`. If an exception escapes `expr`, then `match` is attempted on the exception. If none of the patterns in `match` succeed in matching the exception, it continues to propagate. Exceptions are initiated by the `raise exn` expression.

Tuples

A tuple is an ordered composite value. Caml-Light also has records, but they are not discussed in this thesis.

Lists

As with most functional languages, the list type plays an important role; the list type in ML is polymorphic. There are two data constructors for lists: `::` and `[]`. For example, `(2::3::4::[])` is a list containing three integers. The constructors for lists are not typically used in this way, but rather are used in pattern matches to Deconstruct lists (like car and cdr in LISP). The alternative notation for constructing lists uses braces and semi-colons as in `[1;2;3;4]`.

3.1.2 Declarations

```
decl ::= let pat = expr
       | let rec pat = expr
       | type tydecl
       | exception exndecl
       | ...
```

Let

Let declarations are analogous to let expressions, except that instead of declaring a variable *in* another expression, a let declaration declares top level values. Top level Values are available in all subsequent declarations and expressions.

Types

The basic types in Caml are `int`, `float`, `string`, and `unit`. A tuple type is written with the * character separating the tuple component types; for example, `int * int` denotes a pair of integers. Function types are denoted by an arrow `->` separating argument and result type. Multiple arrows denote higher order function types; for example `int -> int -> int`.

Types can also include type variables, to represent polymorphic types. For example, the list constructor `::` has type `'a -> 'a list -> 'a list`.

A user can declare data types, for example:

```
type 'a Colours = RED of 'a list
                | BLUE of int
;;
```

The example declares a new type Colours and its data constructors RED and BLUE.

Type synonyms are also possible:

```
type IntSet == int list;;
```

Here a new type IntSet is declared to be a synonym for integer lists.

3.2 Introduction to Oasis

Oasis is Caml-Light with parametric overloading. A variable is overloaded if there can be several versions of it available at the same time, where the individual versions are distinguished by their respective types. Overloading is a special case of type polymorphism, usually classified as *ad-hoc* polymorphism[3]. Of course, polymorphism already plays a central role in ML as parametric polymorphism is a primary feature of the ML type system[21, 4]. Parametric overloading is a combination of overloading polymorphism and parametric polymorphism[6].

Most programming languages have overloading of some sort, but overloading has had a mixed reception in the ML community. Standard ML (SML) has overloading only for a fixed set of built-in operations, and it has a distinct mechanism, known as eq-types, for dealing with equality. Haskell has type classes, which allow symbols to be overloaded. While Haskell is like ML in many respects, there are major differences too; most notably, Haskell is a lazy language, while ML is strict. The terms lazy and strict refer to the evaluation strategies of the language: what order are expressions evaluated in? Loosely speaking, lazy means evaluation is delayed until it becomes necessary. Strict means expressions are evaluated as they occur. In particular, strict evaluation enforces function arguments to be fully evaluated before a function is invoked[1]. Examination of the relative merits of various evaluation strategies remains a research issue. In practice though, virtually all mainstream programming languages belong to the strict category.

As yet, there has been no commonly accepted general purpose overloading mechanism for ML. The current version of Caml-Light has no support for overloading whatsoever.

3.3 Examples of Parametric Overloading

Here is a simple first example of overloading in Oasis [2]:

```
#let add_two = fun x -> x + 2;;
add_two : int -> int = <fun>
#let add_two = fun x -> x + 2.0;;
add_two : float -> float = <fun>
#add_two 2;;
- : int = 4
#add_two 2.0;;
- : float = 4.0
```

[1] These definitions are deliberately informal. The precise technical details of evaluation strategies are not relevant here.

[2] This is an excerpt from an interactive, or top-level, Oasis session. The lines beginning with #, the prompt symbol, are the user input and the other lines are the Oasis's responses.

The two declarations of `add_two`, the first for integers and the second for floats, are overloaded; each of these declarations produces an instance of `add_two`. As is typical of languages with overloading, Oasis uses the types of the actual parameters to determine which instance to execute for each application of `add_two`. The process of choosing appropriate instances is called resolution.

When appropriate, ML can defer the resolution decision:

```
#let triple = fun x -> x + x + x;;
triple : [?+: 'a -> 'a -> 'a].'a -> 'a = <fun>
```

The type of triple can be read as polymorphic type `'a -> 'a` with a proviso: when triple is called, an appropriate instance of `+` must be available; this appropriate instance will have type `'a -> 'a -> 'a`. In this example, Oasis did not resolve the `+`'s occurring in the body of triple because it did not see any reason to choose either of the available instances; that is, neither the integer instance nor the float instance seemed more appropriate than the other. Instead of arbitrarily resolving to one of them, or indicating a user error, Oasis deferred the resolution decision by parameterizing triple [3].

The `+` parameter is called an implicit overload parameter; it is implicit in the sense that Oasis deals automatically both with inferring the formal parameter when triple is defined, and with inferring the correct actual parameter at all calls to triple; for example, we can now write the following:

```
#triple 5;;
- : int = 15
#triple 4.5;;
- : float = 13.5
```

[3] Some overloading systems (e.g. SML) would declare user error indicating that + is ambiguous

For each application of triple, Oasis implicitly supplies the correct instance of + based on the actual argument type. Note what happens if we try to apply triple to an inappropriate argument:

```
#triple true;;
Top-level input:
>triple true;;
>^^^^^^
Implicit overload parameter + has no solution
for type bool -> bool -> bool.
```

In Oasis, the *let* construct denotes overloaded binding; there is no extra syntax for declaring overloaded values. Values bound by other mechanisms, such as function parameters or match expressions, are not overloaded. Caml provides a special syntax for curried function definitions:

```
let id pat [pat ...] = expr
```

This syntax is syntactic sugar for the following form:

```
let id = fun pat [pat ...] -> expr
```

Only the id part of such a construct is considered let-bound by Oasis.

Also, function parameters can in fact become overloaded in certain cases, for example:

```
fun x -> let x = e1 in e2
```

Here, x is overloaded in `e2`. In contrast, if you write:

```
let x = e1 in fun x -> e2
```

Then only the function parameter `x` is available in `e2`.

Before we describe the mechanisms of parametric overloading directly, we will present more examples of the new possibilities it enables.

3.3.1 Arithmetic Operators

Defining overloaded arithmetic in Oasis is easy:

```
#let prefix + = int__add_int;;
+ : int -> int -> int = <fun>
#let prefix + = float__add_float;;
+ : float -> float -> float = <fun>
#1 + 2;;
- : int = 3
#1.0 + 2.0;;
- : float = 3.0
#1 + 2.0;;
Toplevel input:
>1 + 2.0;;
>    ^^^
This expression has type float,
but is used with type int.
```

The expression `int__add_int` indicates a qualified reference to global value. Here `int` is the name of a Caml-Light module, and `add__int` is a value declared in that module. In this example, `int__add_int` and `float__add_float` are the primitive addition operations built in to Oasis. The last example fails because we have yet to define addition operations for mixed-mode arithmetic; unlike some languages, Oasis does not do type *coercions*.

3.3.2 Mixed Mode Arithmetic

We can add mixed-mode arithmetic as follows:

```
#let prefix + (x:int) (y:float) = (float_of_int x) + y;;
+ : int -> float -> float = <fun>
#let prefix + (x:float) (y:int) = x + (float_of_int y);;
+ : float -> int -> float = <fun>
#let double x = x + x;;
double : [?+:'a -> 'a -> 'b].'a -> 'b = <fun>
#let triple x = x + x + x;;
triple : [?+:'a -> 'a -> 'b;+:'b -> 'a -> 'c].'a -> 'c = <fun>
```

Note how the type of triple changes in the presence of mixed-mode arithmetic.

3.3.3 Recursion

Recursion allows some interesting examples. Consider arithmetic on lists.

```
#let rec prefix + = fun
  (x::xs) (y::ys) -> (x+y)::(xs+ys)
| xs      []      -> xs
| []      ys      -> ys;;
+ : [?+:'a -> 'a -> 'a].'a list -> 'a list -> 'a list = <fun>
```

Note how the two occurrences of `+` in the first match refer to different instances; one for `'a` and the other for `'a list`. An alternative way of writing the same function would be:

```
#let prefix + = fun xs ys -> map2 (prefix +) xs ys;;
+ : [?+:'a -> 'a -> 'a].'a list -> 'a list -> 'a list = <fun>
```

The function map2 is part of the standard Oasis library. It is defined as follows:

```
let map2 f =
  mapl where rec mapl = fun
    [] [] -> []
  | (h1::t1) (h2::t2) -> f h1 h2 :: mapl t1 t2
  | _ _ -> invalid_arg "map2"
;;
```

3.3.4 Common Abstractions

Since operations on lists are central in ML programs, the library contains a rich set of operations, like map and map2 above, for common tasks. Many of these operations carry over to other data structures as well. Parametric overloading can allow sharing of the common abstract operations between data structures:

```
#type 'a tree = NODE of 'a tree * 'a tree | LEAF of 'a;;
Type tree defined.
#let rec map = fun
  f (NODE (l,r)) -> NODE (map f l, map f r)
| f (LEAF v) -> LEAF (f v);;
map : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
#map add_two [3;4;5];;
- : int list = [5; 6; 7]
#map add_two (NODE (NODE (LEAF 1, LEAF 2), LEAF 3));;
- : int tree = NODE (NODE (LEAF 3, LEAF 4), LEAF 5)
```

This mechanism can be used to produce more abstract code, in some respects similar to object-oriented programming. Here we define map_add_two in a very abstract way:

```
#let map_add_two = fun x -> map add_two x;;
map_add_two : [?add_two:'a -> 'a;
               map:('a -> 'a) -> 'b -> 'c].'b -> 'c = <fun>
#map_add_two [1;2;3];;
- : int list = [3; 4; 5]
#map_add_two (NODE (NODE (LEAF 1.0, LEAF 2.0), LEAF 3.0));;
- : int tree = NODE (NODE (LEAF 3.0, LEAF 4.0), LEAF 5.0)
```

The type of `map_add_two` is fully polymorphic, but it has implicit parameters for both `add_two` and `map`. This example includes a subtlety worth noticing. The type of `map_add_two` does not include the type variable `'a`, except for in its implicit parameters. So the question arises: how can the Oasis type checker determine the correct type for `'a` when `map_add_two` is called? The answer lies in the type of the instances of `map`. The type of `map_add_two` does not make any connection between `'a`, `'b` and `'c`, but the instance types for map do. Consider the application of `map_add_two` to an integer list. Matching the argument of `map` to an integer list causes an inference that the `map` we want has type `(int -> int) -> int list -> int list`[4], which in turn tells us that `'a` must be an integer and that `'b` and `'c` are integer lists; finally, we know that the required `add_two` implicit parameter has type `int -> int`.

3.3.5 Input/Output and Variable Argument Lists

Another example of where overloading is commonly desired is the input/output function print. Caml-light has a printf style module in its standard library. We can use it to define an overloaded print operation:

[4]This inference is based on the notion of an overload template. Overload templates will be introduced in section 3.5.

```
##open "printf";;
#let print x = printf "%d" x;;
print : int -> unit = <fun>
#let print x = printf "%.2f" x;;
print : float -> unit = <fun>
#let print x = printf "%s" x;;
print : string -> unit = <fun>
```

In the presentation so far, all overloaded values have been functions; this is not strictly required by Oasis. An interesting example of overloading a value follows. The example shows how variable length argument lists can be effected through a *trick* which combines parametric overloading and higher-order functions:

```
#let varprint = ();;
varprint : unit = ()
#let rec varprint x = print x; varprint;;
varprint : [?print:'a -> unit;varprint:'b].'a -> 'b = <fun>
#(varprint "hello" 1 2.0 "bye\n" : unit);;
hello12.00bye
- : unit = ()
```

Note the type constraint to type `unit` is necessary for this example to work.

3.3.6 Modules and Parametric Overloading

The final example in this section will present the Oasis approach to specification matching for modules.

Caml-Light has a simple but very effective module system for separate compilation[5]. Modules consist of two files: an interface specification and an implementation. The interface specification contains declarations for values and types to be exported by the implementation. Extending the Caml-Light modules to work for Oasis revealed an interesting problem in the process of matching specification types for values against the implementation.

Caml-Light requires that an implementation value have a type at least as general as its specification type. The implementation may in fact be more general than its specification. Under Oasis, the ability to have a loose fit between the specification and the implementation leads to a problem with name matching. In addition to type inference and overload resolution, Oasis performs a program transformation so that all variable names are changed to include type information. This transformation allows the front-end of the compiler to communicate to the back-end of the compiler the outcome of parametric overloading decisions, in a manner completely transparent to the back-end. Consider the following scenario for modules. Suppose we define a module `IntSet` to implement sets of integers; it has the following interface:

```
type 'a set;;
value singleton : int -> int set;;
value union : int set -> int set ->int set;;
...
```

The implementation could be defined as follows:

```
type 'a set = SET of 'a list;;
```

[5] Caml-Light modules should not be confused with SML modules. SML modules are much more advanced, and their interaction with Oasis style parametric overloading will not be considered here.

```
type intset == int set;;

let singletion x = SET [x];;
let union (SET xs) (SET ys) = SET (xs @ ys);;
...
```

So the specified type for singleton is `int set -> int set -> int set`, but the implementation type is `'a set -> 'a set -> 'a set`. Now suppose some module imports from IntSet and calls `union`. When Oasis compiles the implementation of `union`, it will transform the name of union to reflect the `'a set` type. When oasis compiles a module containing a call to `union`, it will transform the name of union to reflect the `int set` type specified in the module interface. A link error will result when the final program is linked.

There is a simple solution to this name matching problem. When matching an interface against an implementation, Oasis generates a set of implicit declarations corresponding to the exported values. These declarations allow a forced match between the external name required by the specification and the inferred name of the implementations[19]. For `union`, the implicit declaration looks like this:

```
let (union : int set -> int set -> int set) = union;;
```

Oasis processes this implicit declaration as if it appeared at the end of the implementation; it performs type inference and overload resolution to the expression, so that final result of the transformation is a declaration which reads something like:

```
let "union:int set -> int set -> int set" =
        "union:'a set -> 'a set -> 'a set";;
```

The discussion so far has provided a somewhat cursory overview of parametric overloading. The following sections will provide more details of parametric overloading in a manner suitable for the average programmer. The next chapter will present a specification for Oasis in a more formal manner.

3.4 Scoping

Parametric overloading leads to new rules for scoping of values. In ML, when a identifier is bound as a newly declared value, any previous binding is hidden within the scope of the new declaration; ML allows at most one instance of a given value to be visible at a time. In ML, if you write:

```
#let x = 2;;
x : int = 2
#let x = 2.0;;
x : float = 2.0
```

and then

```
#x + 3;;
Toplevel input:
>x + 3;;
>^
This expression has type float,
but is used with type int.
#
```

The error results because the second declared value for `x` hides the first.

Oasis relaxes the ML scoping rules for let bound values. In Oasis, a newly declared let bound value hides only those previously visible values with which it *overlaps*; Oasis allows multiple instances to be visible, so long as they do not have overlapping types. Oasis defines 'overlap' to mean that two types can unify–i.e., the type variables can be manipulated so that the types are made the same. So Oasis gives different results for the prior example:

```
#x + 3;;
- : int = 5
#
```

Overlap comes into play when instances have polymorphic types:

```
#type foo = FOO of int;;
Type foo defined.
#let print = fun x -> printf "%s" "<?>";;
print : 'a -> unit = <fun>
#let print = fun x -> printf "%d" x;;
print : int -> unit = <fun>
#print (FOO 5);;
Toplevel input:
>print (FOO 5);;
>       ^^^^^
This expression has type foo,
but is used with type int.
```

In the above example, a polymorphic print is defined with the intention that it behaves as a default print in case a more specific print does not exist. Then a specific print for integers is declared. However, the type of the integer print overlaps with the type of the default print, so that the application `print (FOO 5)` fails.

Note that in Oasis, non-let declared values hide all previous instances regardless of overlap; for example, if we re-write the previous example slightly so that the second declaration of `x` is a function parameter instead of a let bound value, then we get different results for the `x + 3` expression:

```
#let x = 2;;
x : int = 2
#let f (x:float) = x + 3;;
Toplevel input:
>let f (x:float) = x + 3;;
>                      ^
This expression has type int,
but is used with type float.
#
```

3.5 Templates

A key concept in the semantics of Oasis parametric overloading is the overload template. Overload templates provide a starting point for type inference when an overloaded value appears in an expression. Recall the triple example:

```
#let triple = fun x -> x + x + x;;
triple : [?+: 'a -> 'a -> 'a].'a -> 'a = <fun>
```

For a set of visible instances of a given value, Oasis defines the overload template as the *least general* type which describes all of the instance types. In the case where only a single instance is visible, the least general type is exactly the same as the single instance type; type inference for this case proceeds as in core ML.

The initial environment in Oasis includes two instances of `+` with types `int -> int -> int` and `float -> float-> float` respectively.

Oasis determines the least general type that describes these two types is:

```
a -> 'a ->'a
```

This is the overload template for `+`. When Oasis evaluates the body triple, it uses this overload template for `+` to help infer that the type of triple should be `'a -> 'a`. Oasis also determines, as a result of the inference, that it should defer resolution of the `+` in triple and generate the appropriate implicit parameter.

A special case occurs for recursively defined values. Consider the example:

```
#let add_one x = x + 1;;
add_one : int -> int = <fun>
#let rec add_one = fun (x::xs) -> (add_one x)::(add_one xs) | [] -> [];;
add_one : [?add_one:'a -> 'b].'a list -> 'b list = <fun>
```

When Oasis evaluates the body of the second declaration of `add_one`, it does not know yet the type of newly declared `add_one`—it is in the process of determining exactly that—and so it calculates a very general template for `add_one`. This general template causes Oasis assign a fresh type, i.e. a new type variable is created, for each application of `add_one` appearing in the body.

3.6 New Syntax

There is only one syntactic addition to ML in Oasis. Consider a function declaration:

```
#let double x = x + x;;
double : [?+:'a -> 'a -> 'a].'a -> 'a = <fun>
#
```

If we decide to include `double` in the module interface, the specification is written as follows:

```
value double : [?+:'a -> 'a -> 'a].'a -> 'a;;
```

The expanded style of type expression is called a restricted type. The only place restricted types are allowed is in value specifications of module interfaces. All other type expressions, for example in patterns or in expressions, are not allowed to have the 'restricted' form; i.e. they can not include the implicit parameters.

3.7 Some Design Issues

3.7.1 Instance Scoping and Closures

Consider the following example:

```
let double z =
  let prefix + x y = x ^ y in
  z + x
;;

double "hello"
```

Assuming `+` is defined only for int and float by default, should the application expression `double "hello"` succeed? It depends on what instances of `+` are deemed available for an application of double. There are three possible choices:

1. The instances of `+` available in the scope of the body of double. This is called closed scoping.

2. The instances of `+` available in the scope of the application of double. This is called open scoping.

3. The instances of `+` available both from the scope of the body and the scope of the application. In this case there must also be a decision regarding precedence: if distinct instances having the same type occur in both scopes, then one of the instances must be chosen. We call this hybrid scoping.

It is fairly easy to eliminate pure closed scoping as a reasonable choice because it completely precludes incremental extensions.

The hybrid scoping method seems to offer the most expressive language. Under hybrid scoping, one could interpret the programmer's intent from the previous example to provide a definition of $+$, in this case defined on strings, which can be used inside double but no-where else. This could not be expressed with only open scoping. A deficiency of both closed and hybrid scoping is that the meaning of the overload constraints becomes more complicated.

Under open scoping, we could have a function double with the type:

```
[?+: 'a -> 'a -> 'a].'a -> 'a
```

This type doesn't say anything about the set of instances available for $+$, which is satisfactory if the instances are taken directly from the calling context. This approach mixes well with separate compilation.

Under closed or hybrid scoping, this type doesn't provide enough information. We need to know not only the shape of the type, but the exact set of instances in double's closure. This set of instances must become explicit in the interface specification of double if separate compilation is to work.

Closed (hybrid) scoping is also in some sense counter to the intuition associated with the parameter paradigm. A closure is meant to include free variable references. In our semantics, when a constraint abstracts, it is treated as an implicit parameter–i.e. it is a bound reference as opposed to a free reference. Closed (hybrid) scoping would thus imply that we change the notion of closure to include certain bound values in addition to free values.

Oasis uses open scoping only.

3.7.2 Ambiguity

A design choice I have made in Oasis is to restrict instance sets to be strictly non-overlapping. There are clearly cases, such as a default print routine and the required `unit` annotation in the variable argument list example, that indicate allowing overlaps would be beneficial. However, the algorithm for overload resolution (chapter 5) relies on a process of iterative speculative solution of type variables through identification of unique matches. The unique part breaks if I allow ambiguity into the instance sets through overlapping.

3.7.3 Termination

It is possible to write programs in Oasis that generate ill-formed constraints in the sense that the unrolling will not terminate. Consider the following example:

```
#let f x = x + 1;;
f : int -> int = <fun>
```

```
#let f x = x + 1.0;;
f : float -> float = <fun>
#let f x = f x;;
f : [?f:'a -> 'a].'a -> 'a = <fun>
#f 2;;
Toplevel input:
>f 2;;
>^^^^^
Probable nontermination detected while unrolling overload parameters.
```

The third declaration of `f` is not recursive, which leads to the surprising type of `[?f:'a -> 'a].'a -> 'a`. Also notice that the third declaration hides the first two because of overlap, so that it becomes the only visible instance of `f`. The application `f 2` causes the implicit overload parameter to resolve to

```
f : [?f:int->int].int->int
```

This causes a new `f` to unroll, and so the infinite loop begins. Chapter 4 will present the type rules for Oasis. It will turn out that the type derivation for this example would be infinite. We would also note that had this particular example somehow typed, it still would be non-terminating at run time.

There are several approaches to take to non-termination in Oasis:

1. Design the scenario out of the type system. Make it impossible to form such a program.

2. Devise an error check on new declarations to see if they induce the possibility of an infinite loop.

3. Devise a error check on application expressions to see if they result in an infinite loop.

4. Perform an 'iteration' check based on some arbitrary maximum number of iterations.

I do not believe option 1 to be reasonable. I think it requires restricting the expressiveness of the language to an unacceptable degree.

I don't know if option 2 is possible; I suspect it is an undecidable problem. If it is possible, then it would be my preferred approach for the language specification.

I believe option 3 is possible, and it probably represents the lower bound in terms of an acceptable approach for the language specification.

Option 4 is what is implemented in the current compiler. I believe it is practically equivalent and preferable to option 3 for compiler performance reasons.

3.7.4 Polymorphism and State

There are problems with imperative constructs under plain Hindley-Milner typing that forms the basis of ML typing. Consider the following canonical example:

```
let c = ref (fun x -> x) in
c := (fun x -> x + 1);
(!c) true
```

The expression `(!c) true` would cause a runtime type error—through a hole in ML typing resulting from the naive introduction of imperative features.

There has been much work in this area: SML's imperative type variables[26], SML/NJ's weak typing [2], various work on effect systems [29], the mono-morphism restriction in Haskell[13]. The most recent version of Caml Light has adopted a solution due to Wright[37].

Briefly, Wright's solution works by classifying expressions either as transparent or non-transparent. An expression is transparent if all sub-expressions that could possibly cause side-effects are enclosed by a function definitions. If the expression bound by a let is transparent, then generalization of the bound type is allowed; otherwise, the declaration is held monomorphic in a fashion similar to function parameters.

Wright's solution is conservative because it causes the type system to reject many programs that are well typed under Hindley-Milner. Wright showed that for real programs, this was not a significant problem. The effectiveness of Wright's solution is very encouraging given the level of complexity of the other proposals.

Under parametric overloading, we have a problem with delay of side effects. Parametric overloading must not wrap code in such a way as to alter the order of side effects. For example:

```
let double = (print "Side-effect!"; fun x -> x + x);;
```

In this example, according to ML semantics the `print` expression should execute when the declaration of `double` is evaluated. However, if Oasis were to parameterize double with the implicit `+`, then the print statement would not occur until application of `double`.

We need a way to judge whether the expressions bound to an identifier could cause a side effect. This is very similar to ensuring type safety in the presence of imperative features. In Oasis, let declarations which pass the Wright test, are called *proper* lets. Proper lets are subject to generalization and have implicit overload parameters. Non-proper lets do not generalize, and further do not have overload parameters.

If we extend ML with imperative features as in SML, that is we add the ref constructor and associated operators, then Wright's restriction is sufficient for Oasis. Unfortunately, Caml-Light has its own variation on imperative features that complicates things a bit. Under SML, to access the value pointed at by a reference variable `x`, one must write `!x`. Since this is syntactically an application expression, Wright's test will flag it as non-transparent. In Caml-Light, a there is a *mutable* mechanism which allows a plain variable expression `x` to refer the value; that is, the `!` operator is not required.

A slightly modified version of Wright's restriction is enough to solve our problem. The modification further restricts which expressions are allowed to generalize in the case of variable expressions. The restriction now ensures syntactically that no expression which depends on, or alters state will generalize. If the expression does not generalize, it can not be wrapped with implicit parameters.

Wright uses a very simple argument to show soundness and completeness of ML typing when his restriction is applied. Wright points out in his paper that we can think of his generalization restriction as a conditional transformation; when the let binding expression is not transparent, the type is not generalized so that:

```
let v = e1 in e2
```

becomes equivalent to:

```
((fun v -> e2) e1)
```

A simple preprocessor could transform ML programs in this way, and the resulting programs would then be safe to type by normal ML type inference. This thesis will refer to let bindings that are not transformed by the Wright restriction as *proper*. Proper let bindings will play a key role in the algorithms that implement Oasis parametric overloading.

Since I am further restricting the class of programs which are typable, it is possible that I might cross a line that Wright's solution managed to avoid. That is, I might increase the amount of *real* code that does not type even though it is Hindley-Milner typable. Wright's study concerned a body SML code while I am working with Caml, thus I do not have the same body of code to test against. The only significant test case I have is the Caml Light system itself; at least in this case, there are no new type errors introduced by my strengthened restriction.

Chapter 4

Specification for Core Oasis

In this chapter, I will review the semantics of ML and I will present a language specification for the core subset of Oasis, hereafter referred to as Core Oasis (CO). The specification's goal is to provide a rigorous account of the semantics of parametric overloading, without resorting to the level of detail in the algorithmic presentation of chapter 5.

Typically, language specifications for ML divide the semantics into two separate parts: a static semantics, and a dynamic semantics. The static semantics define how types may be assigned to a program. The dynamic semantics define the operational behavior of the program. The CO specification includes only a static semantics. In addition to defining the allowable type assignments, the rules of the CO static semantics define a source language transformation from CO programs to plain ML programs; these plain ML programs specify the operational behavior of the original CO programs.

$$v \in \text{Var}$$
$$e ::= v \mid e\ e \mid \lambda v.e \mid \text{let } v = e \text{ in } e \mid \text{let}_{rec}\ v = e \text{ in } e$$

Figure 4.1: Syntax of Core-Oasis

4.1 Syntax

The syntax of CO, see Figure 4.1, is just the standard set of mini-ML expressions: the lambda calculus plus let. Expressions include variables, function applications, function definitions, let bindings, and recursive let bindings.

4.2 Semantics of ML

The semantics of ML are traditionally specified by a set of Post-system style inference rules. First, I will present the semantic objects which appear in the rules, and then I will present the rules themselves.

4.2.1 Semantic Objects

Types, Type Schemes, and Environments

Figure 4.2 defines semantic objects corresponding to the types of plain ML: type variables (TyVar), function types (FunType), and type schemes (TyScheme). The format is *sym* $\in$ *name* $=$ *type*. *Sym* is symbolic representation of the object as it appears in the rules. *Name* is the name of the object domain. *Type* describes the structure of the domain.

Type schemes are pairs composed of a set of type variables $\alpha^{(k)}$ and a type τ; the type scheme is written $\forall\alpha^{(k)}.\tau$ to denote that $\tau$is polymorphic with respect to

$$
\begin{array}{rclcl}
\alpha & \in & \text{TyVar} & & \\
\tau & \in & \text{Type} & = & \text{TyVar} \cup \text{ FunType} \\
\tau \rightarrow \tau\text{'} & \in & \text{FunType} & = & \text{Type} \times \text{ Type} \\
(\alpha_1, \ldots, \alpha_k) \text{ or } \alpha^{(k)} & \in & \text{TyVar}^k & & \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \text{TyScheme} & = & \cup_{k \geq 0}\text{TyVar}^k \times \text{ Type} \\
\mathcal{E} & \in & \text{TypeEnv} & = & \text{Var} \stackrel{fin}{\rightarrow} \text{TyScheme}
\end{array}
$$

Figure 4.2: Semantic Objects of ML

$\alpha^{(k)}$, the set of quantified type variables. We call the type componenent of type scheme the *matrix*.

A type environment (TypeEnv), analogous to a symbol table, is a mapping from variables to type schemes.

Instantiation

A type τ is said to instantiate a type scheme σ, written $\sigma \succ \tau$, if τ can be obtained from the matrix τ via an appropriate substitution[1]:

$$\forall\alpha^{(k)}.\tau' \succ \tau \Leftrightarrow \exists\tau^{(k)}.\theta = \{\alpha_i \mapsto \tau_i\} \wedge \tau = \theta\tau'$$

A substitution, θ, is a mapping from type variables to types. We write $\theta\tau$ for the application of a substitution to a type, which has the effect of replacing occurrences of domain type variables with their corresponding range type. A substitution can be composed, as in function composition; we write this as $\theta_2 \circ \theta_1$.

[1]Alternatively, we can say that σ generalizes τ.

```
let gen = fun (ℰ,τ) ⇒
  let α^(k) = tyvars(τ) \ tyvars(ℰ) in
  ∀α^(k).τ
;;
```

Figure 4.3: Generalization Algorithm

Generalization

Figure 4.3 gives the algorithm for *generalization* of types. We note that the Wright restriction (see 3.7.4) is absent from this definition. For the purpose of this chapter, we assume the program has been pre-processed according to the Wright restriction so that only proper lets remain. This assumption allows simplification of many of the rules.

4.2.2 Inference Rules

The inference rules in the following section define valid type assignment for ML.

The inference rules for expressions have the following form:

$$\boxed{\text{RULENAME}} \qquad \frac{X_1 \ \ldots \ X_n}{\mathcal{E} \vdash e \Rightarrow \tau} \tag{1}$$

The terms X_i are premises in the form of recursive inference rule applications or *side-conditions*—logical terms that must be true for the inference rule to hold. An entire rule can be read as: in context $\mathcal{E}$, infer from premises X_i that expression e has type τ.

I present the rules into two parts: the lambda calculus expressions (figure 4.4), and let expressions (figure 4.5).

$\boxed{\text{ML-VAR}}$
$$\frac{\mathcal{E}(v) \succ \tau}{\mathcal{E} \vdash v \Rightarrow \tau} \tag{1}$$

$\boxed{\text{ML-APP}}$
$$\frac{\mathcal{E} \vdash e_1 \Rightarrow \tau' {\rightarrow} \tau \quad \mathcal{E} \vdash e_2 \Rightarrow \tau'}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow \tau} \tag{2}$$

$\boxed{\text{ML-ABS}}$
$$\frac{\mathcal{E}_v \cup \{v \mapsto \tau'\} \vdash e \Rightarrow \tau}{\mathcal{E} \vdash \lambda v.e \Rightarrow \tau' {\rightarrow} \tau} \tag{3}$$

Figure 4.4: ML Inference Rules for Lambda Expressions

Lambda Calculus Expressions

Rule ML-VAR assigns a type to a variable expression. The type scheme of the variable is projected out of the environment and instantiated to a type (see section 4.2.1).

Rule ML-APP defines typing for function application expressions. The type of the first expression, the function expression, must be a function type, with argument type that matches the type of the second expression, the argument expression. The result of the entire expression is the return type of the function expression.

Rule ML-ABS defines typing for function definition, alternatively called λ-abstraction. The environment of the function definition is augmented with a binding for the variable v in typing the body of the function. The binding hides any previous binding of v. The type of the expression is then a function type from the type of the argument to the type of the body.

Let Expressions

Rule ML-LET types a non-recursive let expression. The first premise types the binding expression e_1. The type of the binding is generalized (see section 4.2.1) by the second

$$\boxed{\text{ML-LET}} \qquad \frac{\mathcal{E} \vdash e_1 \Rightarrow \tau' \quad \sigma = gen(\mathcal{E}, \tau') \quad \mathcal{E}_V \cup \{v \mapsto \sigma\} \vdash e_2 \Rightarrow \tau}{\mathcal{E} \vdash \text{let } v = e_1 \text{ in } e_2 \Rightarrow \tau} \qquad (4)$$

$$\boxed{\text{ML-LETR}} \qquad \frac{\begin{array}{c}\mathcal{E}_V \cup \{v \mapsto \tau'\} \vdash e_1 \Rightarrow \tau' \\ \sigma = gen(\mathcal{E}, \tau') \quad \mathcal{E}_V \cup \{v \mapsto \sigma\} \vdash e_2 \Rightarrow \tau\end{array}}{\mathcal{E} \vdash \text{let}_{rec}\ v = e_1 \text{ in } e_2 \Rightarrow \tau} \qquad (5)$$

Figure 4.5: ML Inference Rules for Let Expressions

premise, and it is bound to the variable v in the context of the third premise. Any previous binding of v will be hidden.

Rule ML-LETR types a recursive let expression. Note that the type of the variable is held monomorphic within the binding expression.

4.3 Semantics for Oasis

I now present the semantics for Oasis parametric overloading. As for ML, I will present the objects first and then the inference rules.

4.3.1 Semantic Obects

In order to specify semantics for parametric overloading, a number of new semantic objects are introduced (see Figure 4.6).

Overload Restrictions

An overload restriction (Rstn) is a pair composed of a variable and a type; it represents a requirement for an instance of the variable that satisfies the type. Recalling the canonical

$$
\begin{array}{rccl}
\omega \text{ or } (v,\tau) & \in & \text{Rstn} & = \text{Var} \times \text{Type} \\
\Omega \text{ or } \omega^{(j)} & \in & \text{RstnSet} & = \cup_{j \geq 0}\text{OvldRstn}^{j} \\
(\sigma,\Omega) & \in & \text{OvldInst} & = \text{TypeScheme} \times \text{RstnSet} \\
\Gamma & \in & \text{TemplEnv} & = \text{Var} \stackrel{fin}{\to} \text{TypeScheme} \\
\Theta & \in & \text{InstEnv} & = \text{Var} \stackrel{fin}{\to} \text{OvldInst-list}
\end{array}
$$

Figure 4.6: Semantic Objects new in Core Oasis

example double: double has type `[?+:'a -> 'a -> 'a].'a ->'a`, which is an instance with matrix type `'a -> 'a` and a restriction `(+,'a -> 'a ->'a)`. Informally, we say that an overload restriction can have two possible outcomes: it can *abstract* to an implicit overload parameter, or it can *resolve* to an instance.

Instance Environments

A declaration of variable in CO produces an instance (OvldInst). An instance is defined by a type scheme and a set of overload restrictions (RstnSet). All variables are bound to a list of instances, as defined in the instance environment (InstEnv). In order to simplify the inference rules, we assume that attempting a projection $\Theta(v)$, where v is not in the domain of Θ, will result in an empty list. A particular instance is valid provided that associated unrolled restrictions are valid too.

Restricted Instantiation

The ML notion of instantiation is redefined for instances. We write $(\sigma, \Omega) \succ (\sigma', \Omega')$ defined by:

$$(\forall \alpha^{(k)}.\tau,\Omega) \succ (\tau\text{'},\Omega\text{'}) \Leftrightarrow$$
$$\exists \tau^{(k)}, \tau'^{(l)} \;.\; \theta_1 = \{\alpha_i \mapsto \tau_i\} \wedge \tau' = \theta_1 \tau \wedge$$
$$\alpha'^{(l)} = tyvars(\Omega) \setminus tyvars(\tau) \wedge \theta_2 = \{\alpha'_i \mapsto \tau'_i\} \wedge \Omega' = \theta_2 \theta_1 \Omega$$

The notation is a bit cumbersome, but the definition is actually quite simple. Instantiation converts a type scheme into a type as for instantiation from type schemes to types. The overload restrictions are also restricted in a manner consistent with the matrix, with the addition that uncoupled type variables are also instantiated.

Instance Environment

The overload instance environment is a map from variables to instance lists, where an instance is identified by its type scheme. The instance lists must be constructed so that the order of elements reflect the scoping of the respective declarations.

We note that the instance lists will be constructed in a manner that ignores issues of overlap. The overload resolution process will deal with overlap by converting instance lists into non-overlapping instance lists when appropriate; Figure 4.7 contains the algorithm sift that performs the conversion.

4.3.2 Template Environment

The template environment is map from variables to templates. Structurally, a template is just a non-restricted type scheme, but semantically, it must conform to a particular property: a template must tightly approximate the common structure of the instance list

```
let sift = fun instances ⇒
   let overlap = fun (σ,σ') ⇒ ∃τ.σ ≻ τ ∧ σ' ≻ τ in
   let atop = fun result ⇒ fun (σ,Ω)) ⇒
      if ∃(σ',Ω')∈result.overlap(σ,σ')
      then result
      else result @ [(σ,Ω)] in
   fold atop [] instances
;;
```

Figure 4.7: Algorithm: Sift Overlapping Instances

to which it corresponds, so that the template may serve as a suitable starting point for inferring the type of its associated variable.

More formally, we specify the template property as follows. First, the instance relation is extended to denote the notion of a set of types conforming to a single type:

$$\sigma \succ \Theta \Leftrightarrow \forall(\sigma', \Omega) \in \Theta.\sigma \succ \sigma'$$

We say that σ satisfies Θ. Then we define the template property in terms of the new instance relation. The template property is true for a type scheme σ and an instance set Θ if σ satisfies Θ and σ is an instance of all other type schemes that satisfy Θ:

$$\mathit{templ}(\Theta(v)) ::= \sigma \iff \sigma \succ \Theta(v) \wedge \forall\sigma'.\sigma' \succ \Theta(v) \Rightarrow \sigma' \succ \sigma$$

The inference rules ensure that the template property holds for all reasonable contexts. See section 4.3.3 for more details.

4.3.3 Inference Rules

The inference rules described in the following sections define valid type assignment, restriction assignment and program transformation semantics for CO.

$$\boxed{\text{O-VAR}} \quad \frac{\Gamma(v) \succ \tau \quad \Theta \Vdash (v,\tau) \Rightarrow \Omega, e}{\Theta, \Gamma \vdash v \Rightarrow \tau, \Omega, e} \quad (1)$$

$$\boxed{\text{O-APP}} \quad \frac{\Theta, \Gamma \vdash e_1 \Rightarrow \tau' \rightarrow \tau, \Omega_1, e_1' \quad \Theta, \Gamma \vdash e_2 \Rightarrow \tau', \Omega_2, e_2'}{\Theta, \Gamma \vdash e_1\ e_2 \Rightarrow \tau, \Omega_1 \cup \Omega_2, e_1'\ e_2'} \quad (2)$$

$$\boxed{\text{O-ABS}} \quad \frac{\Theta_v \cup \{v \mapsto [(\tau', \{\})]\}, \Gamma_v \cup \{v \mapsto \tau'\} \vdash e \Rightarrow \tau, \Omega, e'}{\Theta, \Gamma \vdash \lambda v.e \Rightarrow \tau' \rightarrow \tau, \Omega, \lambda v_{\mathcal{T}}.e'} \quad (3)$$

Figure 4.8: Oasis Inference Rules for Lambda Calculus Expressions

The inference rules for expressions have the following form:

$$\boxed{\text{RULENAME}} \quad \frac{X_1 \ \ldots \ X_n}{\Theta, \Gamma \vdash e \Rightarrow \tau, \Omega, e'} \quad (1)$$

The terms X_i are premises in the form of recursive inference rule applications or *side-conditions*—logical terms that must be true for the inference rule to hold. An entire rule can be read as: in context (Θ,Γ), infer from premises X_i that expression e has type τ, produces a set of abstract overload restrictions Ω, and transforms into expression e'.

The rules for solving overload restrictions, given in section 6, have a different form:

$$\boxed{\text{RULENAME}} \quad \frac{X_1 \ \ldots \ X_n}{\Theta \Vdash \omega \Rightarrow \Omega, e} \quad (1)$$

These rules may be read as: in context Θ, infer from premises X_i that the overload restriction ω produces a set of abstract overload restrictions Ω, and an expression e.

λ-Calculus Expressions

Figure 4.8 contains the CO inference rules for the basic λ-calculus expressions.

$$\boxed{\text{O-LET}}\quad \frac{\begin{array}{c}\Theta,\Gamma \vdash e_1 \Rightarrow \tau', \Omega_1, e_1' \\ \sigma = gen(\Theta,\tau') \quad e_1'' = wrap(\Omega_1, e_1') \\ \Theta' = \Theta_{\bar{v}} \cup \{v \mapsto ((\sigma,\Omega_1) :: \Theta(v))\} \quad \Gamma' = \Gamma_{\bar{v}} \cup \{v \mapsto templ(\Theta'(v))\} \\ \Theta',\Gamma' \vdash e_2 \Rightarrow \tau, \Omega_2, e_2'\end{array}}{\Theta,\Gamma \vdash \text{let } v = e_1 \text{ in } e_2 \Rightarrow \tau, \Omega_2, \text{let } v_\sigma = e_1'' \text{ in } e_2'} \quad (4)$$

$$\boxed{\text{O-LETR}}\quad \frac{\begin{array}{c}\Theta' = \Theta_{\bar{v}} \cup \{v \mapsto ((\tau',\Omega_1) :: \Theta(v))\} \quad \Gamma' = \Gamma_{\bar{v}} \cup \{v \mapsto templ(\Theta'(v))\} \\ \Theta',\Gamma' \vdash e_1 \Rightarrow \tau', \Omega_1, e_1' \\ \sigma = gen(\Theta,\tau') \quad e_1'' = wrap(\Omega_1, e_1') \\ \Theta'' = \Theta_{\bar{v}} \cup \{v \mapsto ((\sigma,\Omega_1) :: \Theta(v))\} \quad \Gamma'' = \Gamma_{\bar{v}} \cup \{v \mapsto templ(\Theta''(v))\} \\ \Theta'',\Gamma'' \vdash e_2 \Rightarrow \tau, \Omega_2, e_2'\end{array}}{\Theta,\Gamma \vdash \text{let}_{rec}\ v = e_1 \text{ in } e_2 \Rightarrow \tau, \Omega_2, \text{let}_{rec}\ v_\sigma = e_1'' \text{ in } e_2'} \quad (5)$$

Figure 4.9: Oasis Inference Rules for Let Expressions

In rule O-VAR, a variable expression produces a new overload restriction. The overload restriction premise triggers a solution derivation. The result of the solution derivation includes any resulting abstract restrictions and an ML expression representing the solution of the variable v with type τ. The type τ is derived from the template for v and the outcome of the solution derivation.

The O-APP rule for function application is very simple. We only note that abstract overload restrictions are simply propagated from the premises.

The O-ABS rule for function definition (λ-abstraction), binds the parameter v in a *non-overloaded* fashion; i.e. all instances of v in the context are hidden regardless of overlap. We introduce the notation v_τ for a canonical transformation of the pair (v,τ) into an ML identifier.

Let Expressions

Figure 4.9 contains the inference rules for the let expressions of CO.

```
let wrap = fun (ω^(j), e) ⇒
   if j = 0 then e
   else
      let (v_i, τ_i) = ω_i in --i∈[j]
      λv_1τ_1 ... λv_jτ_j.e
;;
```

Figure 4.10: Algorithm: Wrap Expression with Overload Parameters

$$\boxed{\text{RSLVRSTN}} \qquad \frac{\exists!(\sigma,\Omega) \in sift(\Theta(v)).(\sigma,\Omega) \succ (\tau,\omega^{(j)}) \quad j \geq 0 \qquad \overline{\Theta \Vdash \omega_i \Rightarrow \Omega_i, e_i} \quad i \in [j]}{\Theta \Vdash (v,\tau) \Rightarrow \cup_{i\in[j]}\Omega_i, (v_\sigma\ e_1\ \ldots\ e_j)} \tag{6.1}$$

$$\boxed{\text{ABSRSTN}} \qquad \frac{\nexists(\sigma,\Omega) \in sift(\Theta(v)).(\sigma,\Omega) \succ (\tau,\Omega') \quad |\text{tyvars}\ \tau| > 0}{\Theta \Vdash (v,\tau) \Rightarrow \{(v,\tau)\}, v_\tau} \tag{6.2}$$

Figure 4.11: Inference Rules for Solving Overload Restrictions

The O-LET rule is for non-recursive let expressions. The set of overload restrictions produced by the binding expression e_1 is captured by the instance for the bound variable v. The call to `wrap` produces a transformation of e_1 where overload restrictions become function parameters. Figure 4.10 defines `wrap`.

The O-LETR adds premises for building the environments of the recursive binding.

Overload Resolution

Figure 4.11 contains the inference rules for overload restrictions. There are two rules corresponding to the possible outcomes of the resolve process for a given restriction and context: a restriction can resolve (possibly *unrolling* other restrictions) or it can abstract.

An overload restriction resolves if there is a unique member of the instance environment that matches the restriction's type. A match is defined in terms of restricted instantiation (see section 4.3.1). Recursive inference on each of the restrictions of the match are discharged by the second premise (the over-bar is notation for repetition over i). Note the form of the produced expression: a function application is generated making explicit the overload parameters, if any exist, of the match.

A restriction abstracts if there are no members of the instance environment that match and the restriction's type is *non-ground*. There are two types of error implied by this rule. The first occurs when the restriction's type is ground. The second error occurs when there is more than one match to choose from; ambiguity is not allowed.

4.4 On the weakness of the semantics

A major success of the design of the original semantics for ML is the very strong correctness criteria the design supports. There are proven results of soundness and completeness that describe quite precisely how the ML type system is correct: the *soundness* of the static semantics means well-typed programs can not go wrong, and the *completeness* of the type inference algorithm means that a well-typed program is assigned the correct types. In particular, the correct types are known as *principal types*.

This thesis leaves to future work the task of establishing the same sorts of correctness results for the semantics of parametric overloading, but I will now sketch out how the future work might proceed. I believe the semantics are sound, but that the ML notion of principal types is inadequate with respect to the Oasis type semantics and inference algorithm. Mark Jones has made very similar observations in the development of his Qualified Type theory for overloading in Haskell[16]. Because of some fairly significant semantic and notational differences, I can not compare parametric overloading to qualified types directly, but I believe that the intuition behind Jone's *principal satisfying types* applies well to parametric overloading of Oasis.

Plain principal types say that the most general type is the correct type. In a system with overloading, there are many possible policies for overload resolution. Unlike a pure ML system, an ML system with overloading will have difficulty in adhering to policy of assigning the most general type advocated by principal types. The most general type bias conflicts with the need to resolve overloaded values. There are several alternative policies which provide various levels of balance between the generality and resolution. Oasis uses the liberal uncoupled resolution policy (LURP) [25].

The overload restriction rules for Oasis (see figure 4.11) are constructed in a way that the structural form of a type derivation reflects whether overload restrictions resolve or abstract. Thus, the structural form of a type derivation may need to change depending on what substitutions are applied to the types; for example, an overload restriction that resolved under the original type assignment, might abstract after a substitution is applied. *Coherence* seems to be lost.

Under LURP, types are first assigned to a program in nearly the same manner as the original ML algorithm W; the only modification is the addition of overload templates for overloaded values. The result of the assignment is a program decorated with types; the types in this initial decoration contain two distinct class of type variable: coupled and uncoupled. Coupled variables are those that have been *generalized.* By definition of the inference rules, a generalized variable must appear in the matrix of a type assigned to a let bound variable—hence the name coupled. LURP says that uncoupled variables are *fair game* for substitution in the second phase of type assignment: overload resolution. However, unlike plain ML with principal types, a theory for LURP type inference should not consider all substitutions equal.

For LURP, we need to stratify substitutions to allow a suitable notion of coherence. We borrow the intuition of an *improving* substitution from the theory of Qualified types. The notion of improvement is defined relative to the set of overload restrictions that a substitution may affect. A substitution improves those restrictions if it can contribute to the process of resolving one or more of them without hindering the resolution of any others. Here, *contribution* means that the overload restrictions in question will not resolve unless the substitution is applied. *Hindering* means a substitution conflicts with possible solutions for some restriction. We would also define *neutral* substitutions as those which neither contribute to nor hinder the resolution process. I would then define a type as *Satisfied* if there exist no improvements for any type variables in the type. I would define Principal Satisfying Types then to be *most general satisfied types.* A corollary to the existence of Principal Satisfying Types would be the coherence of the type assignment under neutral substitutions.

The goals for future work are to formalize the above ideas and integrate them into semantics of Oasis, then to prove soundness, and finally to prove completeness of the inference algorithm relative to the notion of principal satisfying types.

The current specification of CO semantics is particularly weak in the sense that it under-specifies how types are assigned with respect to *resolution policy.* I have sketched out how we might strengthen the specification to allow a new notion of *Principal Satisfying Types.*

Chapter 5

Implementation of Oasis

This chapter will overview the design of the prototype Oasis compiler, *oasisc*. Oasisc is a modified version of the Caml Light system compiler, camlc. The Oasis system mirrors the Caml-Light system, with most Oasis tools nearly identical to the Caml-Light versions. The two systems are not directly compatible because of changes in basic system data structures. In particular, Caml-Light and Oasis object code can not be interchanged.

The overall structure of oasisc is depicted in figure 5.1.

The scanning/parsing step performs a lexical scan and grammatical parse of the input source code, validating that the input program conforms to lexical and syntax conventions of ML. The Caml-Light system includes a pair of tools, camlyacc and camllex, which are used to generate the camlc scanner and parser from declarative style specifications. The output of the scanning/parsing step is an abstract syntax tree (AST) representation of the input program.

The dotted box in figure 5.1 represents the augmented typing step of Oasis; the typing step includes a number of sub-steps: an ML like type inference step, plus additional steps for annotation of overload restrictions, iterative resolution and propagation of overload restrictions, and transformation of overload restrictions into implicit function parameters.

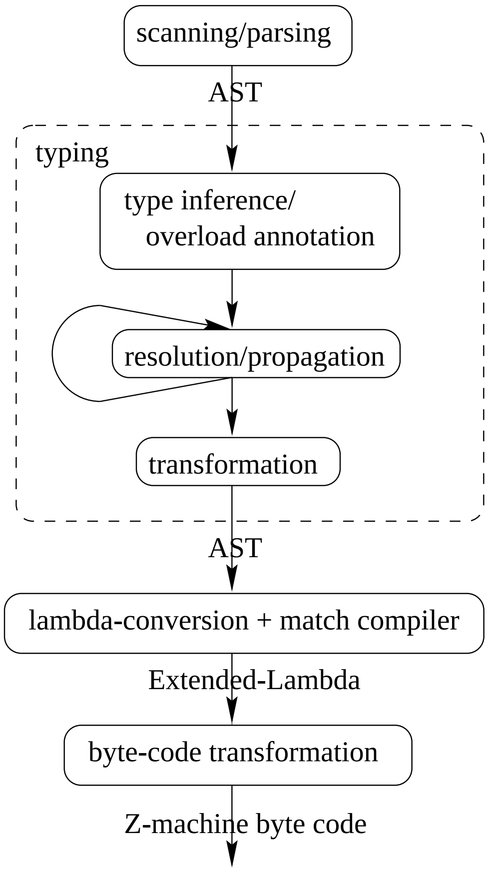

Figure 5.1: Oasis Overview

$$\begin{array}{cccll} \mathcal{E} & \in & \text{TypeEnv} & = & \text{Var} \stackrel{fin}{\rightarrow} \text{TypeScheme} \\ \theta & \in & \text{Subst} & = & \alpha \stackrel{fin}{\rightarrow} \tau \end{array}$$

Figure 5.2: Data structures for ML type inference

Each of these steps will be discussed in greater detail in the remaining sections of this chapter.

After the typing phase is complete, a lambda conversion phase transforms the AST into an extended lambda calculus representation; the lambda conversion phase includes the pattern match compiler which converts arbitrary ML patterns into simpler forms supported by the extended lambda calculus. Finally, the byte-code transformation step produces object-code ready for linking and subsequent execution by the Caml-Light run-time interpreter.

Of the four steps contained in the diagram of 5.1, only the typing step of oasisc differs significantly from the corresponding part of camlc. Xavier Leroy's technical report on Zinc[17][1] presents the design of camlc, and so it also describes all but the typing step of oasisc. The remainder of this chapter then, is concerned with providing detailed descriptions of the data structures and algorithms which form the typing step of oasisc.

5.1 Data Structure Definitions

5.2 ML Type Inference

This section will review the basic algorithm for ML type inference $\mathcal{W}_{ML}$, as preparation for a subsequent presentation of the Oasis type inference algorithm $\mathcal{W}_{Oasis}$. Figure 5.3 contains Caml-style pseudo-code for $\mathcal{W}_{ML}$. For the purpose of brevity, the algorithms

[1] Caml-Light was called Zinc early in it's history.

$$
\begin{array}{l}
\mathbf{let}_{rec}\ \mathcal{W}_{ML}\ \texttt{=}\ \mathbf{fun}\ \texttt{(}\mathcal{E}\texttt{,e)}\ \Rightarrow \\
\quad \mathbf{match}\ \texttt{e}\ \mathbf{with} \\
\quad\quad \texttt{v}\ \Rightarrow\ \mathbf{let}\ \forall\alpha^{(k)}.\tau\texttt{=}\ \mathcal{E}\texttt{(v)}\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \alpha'^{(k)}\ \langle \textit{be fresh type variables} \rangle\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \theta = \{\alpha_i \mapsto \alpha'_i\}\ \mathbf{in} \\
\quad\quad\quad\quad (\theta\tau, \emptyset) \\
\quad \mid\ e_1\ e_2\ \Rightarrow\ \mathbf{let}\ (\tau_1,\theta_1) = \mathcal{W}_{ML}(\mathcal{E}, e_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ (\tau_2,\theta_2) = \mathcal{W}_{ML}(\theta_1\mathcal{E}, e_2)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \alpha\ \langle \textit{be a fresh type variable} \rangle\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \theta_3 = mgu(\theta_2\tau_1, \tau_2 \to \alpha)\ \mathbf{in} \\
\quad\quad\quad\quad (\theta_3\theta_2\tau_1, \theta_3 \circ \theta_2 \circ \theta_1) \\
\quad \mid\ \lambda v.e\ \Rightarrow\ \mathbf{let}\ \alpha\ \langle \textit{be a fresh type variable} \rangle\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ (\tau,\theta) = \mathcal{W}_{ML}(\mathcal{E}_v \cup \{v \mapsto \alpha\}, e)\ \mathbf{in} \\
\quad\quad\quad\quad (\theta\alpha \to \tau, \theta) \\
\quad \mid\ \texttt{let}\ v = e_1\ \texttt{in}\ e_2\ \Rightarrow\ \mathbf{let}\ (\tau_1,\theta_1) = \mathcal{W}_{ML}(\mathcal{E}, e_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \sigma = genexp(\theta_1\mathcal{E}, \tau_1, e_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ (\tau_2,\theta_2) = \mathcal{W}_{ML}(\theta_1\mathcal{E}_v \cup \{v \mapsto \sigma\}, e_2)\ \mathbf{in} \\
\quad\quad\quad\quad (\tau_2, \theta_2 \circ \theta_1) \\
\quad \mid\ \texttt{let}_{rec}\ v = e_1\ \texttt{in}\ e_2\ \Rightarrow\ \mathbf{let}\ \alpha\ \langle \textit{be a fresh type variable} \rangle\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \mathcal{E}' = \mathcal{E}_v \cup \{v \mapsto \alpha\}\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ (\tau_1,\theta_1) = \mathcal{W}_{ML}(\mathcal{E}', e_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \theta_2 = mgu(\theta_1\alpha, \tau_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ \sigma = genexp(\theta_2\theta_1\mathcal{E}', \theta_2\tau_1, e_1)\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ (\tau_2,\theta_3) = \mathcal{W}_{ML}(\theta_2\theta_1\mathcal{E}_v \cup \{v \mapsto \sigma\}, e_2)\ \mathbf{in} \\
\quad\quad\quad\quad (\tau_2, \theta_3 \circ \theta_2 \circ \theta_1) \\
\texttt{;;}
\end{array}
$$

Figure 5.3: ML type inference algorithm $\mathcal{W}_{ML}$

given in this chapter deal with only the five core language constructs of ML/Oasis; extension to the full language is straightforward.

The input to $\mathcal{W}_{ML}$ is a type environment $\mathcal{E}$ and an ML expression e. A type environment, also known as the symbol table, is a mapping from variables to types. The result of the type inference algorithm is the type τ of the input expression and a substitution θ. The substitution portion of the result records type information inferred by the algorithm; it allows for the presentation of the algorithm to maintain a functional style, making it easier to understand and reason about the algorithm's operation. For performance reasons, the type inference algorithm actually used in Caml Light eliminates the use of explicit substitutions by embedding state in the representation of type variables[2]. The resulting algorithm retains the general structure of the original functional version, so I will present the slightly more abstract functional version here.

5.2.1 Variable Expressions (lines 3–6)

$\mathcal{W}_{ML}$ assigns to a variable expression an instance of the type scheme associated with the variable in the type environment.

$\mathcal{W}_{ML}$ produces the instance by substituting *fresh* type variables for the bound variables of the type scheme. If the type scheme has no bound type variables, the instance type will be the same as the original type scheme. It is an error if the variable is not present in the domain of the type environment, but we omit such minor details to make the presentation concise.

5.2.2 Function Application Expressions (lines 7–11)

Given an application expression, $e_1\ e_2$, $\mathcal{W}_{ML}$ infers the types of the subexpressions (lines 7–8) and then uses mgu to enforce that the sub-expressions have suitable types (line 10); that is, e_1 must be a function type with formal parameter type that matches the actual parameter e_2. The magic that underlies $\mathcal{W}_{ML}$ is the unification algorithm mgu, described in Figure 5.4. For a more complete discussion of unification, see [33, 8].

5.2.3 Function Definition Expresssions (lines 12–14)

$\mathcal{W}_{ML}$ types a function definition $\lambda v.e$ by creating a fresh type for v, and then adding this type to the type environment used for the body of the definition e. The type of the whole

[2]Most other ML implementations share this approach

```
let_rec mgu = fun (τ1, τ2) ⇒
   match (τ1, τ2) with
     (τ'1 → τ'2, τ''1 → τ''2) ⇒
      let θ1 = mgu(τ'1, τ''1) in
      let θ2 = mgu(θ1τ'2, θ1τ''2) in
      θ2 ∘ θ1
   | (α, τ) | (τ, α) ⇒
      if α = τ then {}
      else
         occur-check(α,τ); --raise UnifyError if α occurs in τ
         {α↦τ}
   | _ ⇒ raise UnifyError
;;
```

Figure 5.4: Most General Unifier *mgu* Algorithm

expression is naturally a function from the type of v to the type of e.

5.2.4 Non-recursive Let Expressions (lines 15–18)

The let expression has a special status in ML. From an operational perspective, a let expression *let* $v = e_1$ *in* e_2 is exactly the same thing as $((\lambda v.e_2)\ e_1)$, but they are not the same with respect to static typing. A let bound value can have a polymorphic type, whereas a function parameter always has a monomorphic type. The reason for the distinction is to maintain the tractability of the type-inference problem; if function parameters were allowed to be polymorphic, type inference would become undecidable[12][3].

The type of v is made polymorphic by the call to *genexp* (line 16).

Proper Let Bindings

Although, the core subset of ML presented here does not include any of the reference type or mutable data features of Caml-Light, we should note that the reason *genexp* takes the

[3] An analogous result holds for recursive occurrences of let bound values

expression as one of its parameters is to allow correct treatment of these features, as discussed in section 3.7.4.

5.2.5 Recursive Let Expressions (lines 19–25)

$\mathcal{W}_{ML}$ types a recursive let similar to a combination of a function binding and non-recursive let binding. The bound variable is assigned a monomorphic type in the scope of its own definition (lines 19–21) as with a function parameter, and then the type is generalized and bound to a polymorphic type in the body of the let (lines 23–24) as with a non-recursive let. The call to mgu (line 22) ensures that occurrences v within its own definition have the same type as the entire definition, acting something like a retro-active crystal ball.

5.3 Data structures

The algorithms for Oasis typing will operate on the data structures of figures 4.2 and 4.6 as well as a number of new data structures which I introduce now in figure 5.5. Two of the objects types in 5.5, overload restrictions (Rstn) and overload restriction sets (RstnSet), are re-definitions of objects defined in figure 4.6. These redefinitions reflect the transition from abstract specification issues to more concrete implementation details.

5.3.1 Overload Restriction

The overload restriction is defined with an explicit status, to be used in tracking the restriction's progress through the typing step. An overload restriction always starts out with an open status (ST_OPEN). Through successful typing, the status will *solve* either to abstracted (ST_ABSD) or resolved (ST_RSLVD).

$$
\begin{array}{rccl}
(v,\tau,\mathrm{status}) \text{ or } \omega & \in & \mathrm{Rstn} & = \mathrm{Var} \times \mathrm{Type} \times \mathrm{RstnStatus} \\
\mathrm{status} & \in & \mathrm{RstnStatus} & = \mathrm{ST\_OPEN\ of\ RstnCtxInfo} \\
 & & & \mid \mathrm{ST\_ABSD} \\
 & & & \mid \mathrm{ST\_RSLVD\ of\ RslvKind} \\
(\Theta,\mathrm{scope}) & \in & \mathrm{RstnCtxInfo} & = \mathrm{InstEnv} \times \mathrm{ScopeNum} \\
\mathrm{scope} & \in & \mathrm{ScopeNum} & = \mathrm{int\ list} \\
\mathrm{rslv\_kind} & \in & \mathrm{RslvKind} & = \mathrm{RK\_CLOSED\ of\ Rstn\ list} \\
 & & & \mid \mathrm{RK\_OPEN\_NO\_INST\ of\ WrapDesc} \\
 & & & \mid \mathrm{RK\_OPEN\_INST\ of\ PropDesc} \\
\Pi & \in & \mathrm{PropMap} & = \mathrm{Rstn} \stackrel{fin}{\to} \mathrm{Rstn} \\
(\mathrm{mark},\omega^k,\mathrm{scope}) & \in & \mathrm{WrapDesc} & = \mathrm{Bool} \times \mathrm{Rstn\text{-}list} \times \mathrm{ScopeNum} \\
\mathrm{prop\_desc} & \in & \mathrm{PropDesc} & = \mathrm{Subst} \times \mathrm{PropMap} \times \\
 & & & \quad \mathrm{RstnCtxInfo} \times \mathrm{WrapDesc} \\
\Omega & \in & \mathrm{RstnSet} & = \mathrm{RS\_OPEN\ of\ WrapDesc} \\
 & & & \mid \mathrm{RS\_CLOSED\ of\ Rstn\ list} \\
 & & & \mid \mathrm{RS\_NONE}
\end{array}
$$

Figure 5.5: Data Structures for Oasis typing

Open Status

The argument to the ST_OPEN data constructor contains restriction context information: an instance environment which contains the set of instances against which the restriction is solved, and a scope number. The context information contains the entire instance environment, rather than just the set of instances for the specific variable, in order to support unrolling when a restriction resolves. We will discuss unrolling more when we come to that part of the algorithm. Scope numbers are used in managing propagation of restrictions; they are explained fully in section 5.3.1.

Resolved Status

The argument to ST_RSLVD is a resolved kind descriptor, which describes the relationship between the expression that spawned the restriction and the expression that defined the instance to which the restriction resolved. This relationship determines how overload restrictions are to propagate.

Scope Numbers and Resolution Descriptors

The Oasis algorithm for propagation has to take care to avoid non-termination when it deals with recursion in programs. A naive algorithm could easily end up with an ever diverging set of overload restrictions. Recall that ML restricts the type of a recursively declared value to be non-polymorphic within the scope of the value's own definition; it has been shown that lifting this restriction would cause type inference to become undecidable[12]. From the specification of ML, we see that no explicit distinction is made between recursive and non-recursive variable expressions. The restriction to a non-polymorphic type in the recursive case is achieved implicitly by virtue of how type environments are constructed by the ML type inference algorithm: a type environment is generalized only after the recursive expressions have been typed. Thus, when a recursive variable expression is typed, the instantiation step will have no effect because the variable's type can have no quantified type variables. The ML definition of instantiation is extended in Oasis to include generation of new overload restrictions. Since ML typing is defined so that instantiation has no effect for recursive variable expressions, it follows that no new overload restrictions are to be generated either, which turns out to be crucial in avoiding non-termination in the propagation algorithm.

In the Caml-Light implementation, the type inference algorithm does generalization of types via side-effect; consequently, the implicit distinction between recursive and non-recursive variable expressions is no longer visible after the type inference step is complete. In adapting Caml-Light typing to Oasis, I found it necessary to find a way of distinguishing recursive and non-recursive variable expressions after type inference, that is during the resolution/propagation step, so that non-termination could be avoided. One option would have been to re-structure the Caml-Light type inference to remove the unwanted side-effects; my experience with WatML-NJ would have provided a reference point, since SML/NJ's type inference did generalization in a manner that preserved the recursive/non-recursive distinction. I decided against this option because in my opinion, Caml-Light's type inference algorithm is much cleaner than SML/NJ's, and furthermore the WatML propagation algorithm suffered greatly from trying to distinguish between recursive and non-recursive expressions through information implicit in types.

Instead, I decided it would be better to make the distinction between recursive and non-recursive applications explicit in the Oasis algorithms and data structures. The Oasis typing step generates *scope numbers* which are used to achieve the desired distinction. The relationship of scope numbers to let bindings is similar to the relationship of section numbers to this thesis document.

Consider the example in figure 5.6. The subscripts on let bound variable bindings and variable expressions are the scope numbers. The top-level phrase begins with the null scope number $\emptyset$. A let declaration, in the form **let** v = *binding* **in** *body* is analogous to

$$
\begin{array}{l}
\text{let}_{rec}\ foo_1 = \text{fun } x \Rightarrow \\
\quad\quad \text{let}_{rec}\ bar_{1.1} = \text{fun } y \Rightarrow \\
\quad\quad\quad\quad \text{if } y_{1.1} > 1 \text{ then} \\
\quad\quad\quad\quad\quad y_{1.1} * ((bar_{1.1}(y_{1.1} - 1)) + (foo_{1.1}(x_{1.1} - 1))) \\
\quad\quad\quad\quad \text{else } y_{1.1} \text{ in} \\
\quad\quad bar_1\ x_1 \text{ in} \\
\text{let } baz_2 = (foo_2\ 5) + (foo_2\ 6) \text{ in} \\
baz_{\emptyset}\ ;;
\end{array}
$$

Figure 5.6: Example: Scope numbers

declaring a new sub-scope containing *binding* and then ending the sub-scope in continuing with *body*. The binding for v is marked with a scope number of the sub-scope, and all variable expressions are marked with appropriate scope-numbers. A variable expression is recursive with respect to a binding if the binding's scope number forms a prefix for the variable expression's scope number. In the example, the *bar* in *bar* $(y - 1)$ is recursive with respect to the binding of *bar* because the scope numbers match, but *bar* in *bar* x is not recursive because 1.1 is not a prefix for 1.

5.3.2 Restriction Set

The Oasis typing step will construct restriction sets through an iterative solution and propagation process. Recall from figure 4.6 that restriction sets form a component of overload instances. Proper let bound variables capture overload restrictions arising from their defining expressions[4]. The instances corresponding to these variables are assigned an RS_OPEN instance set within the context of the toplevel let in which they are defined.

The argument to RS_OPEN is a wrap descriptor (WrapDesc). A wrap descriptor contains three fields: a mark, a set of overload restrictions, and a scope number. The annotation step will produce a set of wrap descriptors corresponding to all of the proper let bound values within an expression. The annotation step collects overload restrictions

[4]See section 3.7.4 for an explanation of proper let bindings.

from variable expressions and binds them to appropriate wrap descriptors. In this way the wrap descriptors and initial overload restrictions represent a set of call graphs. The resolution/propagation step does an interpretation over the graphs, inferring the flow of overload restrictions within the program. As the interpretation proceeds, new edges are added to the graph representing resolutions. When recursion is present in the program, these edges can cause cycles to form in the graph. The mark field allows safe (terminating) traversals of these cyclic graphs through a mark and sweep algorithms. Scope numbers also guide the interpretation as described in section 5.3.1.

When the typing of a toplevel implementation phrase is complete, any exported instances will have their restriction sets converted to RS_CLOSED for simplification of typing later declarations. All other variables, (function parameters, non-transparent let bindings, exception parameters, etc.) can not capture overload restrictions; an instance of one of these variables has restriction set RS_NONE.

Oasis has greatly improved on WatML-NJ in terms of choice of data structures and algorithm for restriction set construction. In WatML-NJ, overload restrictions are propagated fully prior to type inference and the solution step. WatML-NJ has a separate pruning step, located after the solving step, to remove restrictions which turn out to be unnecessary. This part of WatML-NJ is particularly messy, and replacing it was part of the motivation behind the Oasis re-write.

5.4 Oasis Type Inference

The Oasis type inference algorithm is given in figures 5.7 and 5.9. $\mathcal{W}_{Oasis}$ takes an instance environment, a template environment, and an expression as input.

The output of $\mathcal{W}_{Oasis}$ is a five-tuple. The meaning of each field is as follows:

1. An annotated version of the input expression. The annotations will direct transformation from the Oasis expression to a plain ML expression.

$$
\begin{array}{l}
\mathbf{let}_{rec}\ \mathcal{W}_{Oasis}\ \texttt{=}\ \mathbf{fun}\ (\Theta,\Gamma,\texttt{scope},e)\ \Rightarrow \\
\quad \mathbf{match}\ e\ \mathbf{with} \\
\qquad v\ \Rightarrow\ \mathbf{let}\ \forall\alpha^{(k)}.\tau\ \texttt{=}\ \Gamma(v)\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \theta\ = \{\alpha_i{\mapsto}\alpha'_i\}\ \langle \textit{where } \alpha'_i \textit{ are fresh type variables}\rangle\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \omega\ = (v, \theta\tau, \texttt{ST\_OPEN}(\Theta, \texttt{scope}))\ \mathbf{in} \\
\qquad\qquad (\omega,\theta\tau,\{\},[\omega],[]) \\
\quad\ \mid\ e_1\ \ e_2\ \Rightarrow\ \mathbf{let}\ (e'_1,\tau_1,\theta_1,\omega s_1,\Omega s_1) = \mathcal{W}_{Oasis}(\Theta,\Gamma,\texttt{scope},e_1)\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (e'_2,\tau_2,\theta_2,\omega s_2,\Omega s_2) = \mathcal{W}_{Oasis}(\theta_1\Theta,\theta_1\Gamma,\texttt{scope},e_2)\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \alpha\ \langle \textit{be a fresh type variable}\rangle\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \theta_3 = \textit{mgu}(\theta_2\tau_1, \tau_2{\to}\alpha)\ \mathbf{in} \\
\qquad\qquad (\theta_3\theta_2 e'_1\ \ \theta_3 e'_2, \theta_3\alpha, \theta_3\circ\theta_2\circ\theta_1, \theta_3\theta_2\omega s_1 @ \theta_3\omega s_2, \theta_3\theta_2\Omega s_1 @ \theta_3\Omega s_2) \\
\quad\ \mid\ \lambda v.e\ \Rightarrow\ \mathbf{let}\ \alpha\ \langle \textit{be a fresh type variable}\rangle\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \Theta' = \Theta_v \cup \{v{\mapsto}[(\alpha,\texttt{RS\_NONE})]\}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (e',\tau,\theta,\omega s,\Omega s) =\ \mathcal{W}_{Oasis}(\Theta', \Gamma_v\cup\{v{\mapsto}\alpha\}, \texttt{scope}, e)\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \tau' = \theta\alpha\ \mathbf{in} \\
\qquad\qquad (\lambda v:\tau'.e', \tau'\to\tau, \theta, \omega s, \Omega s) \\
\quad\ \mid\ \texttt{let}\ \ v = e_1\ \ in\ \ e_2\ \Rightarrow \\
\qquad\qquad \mathbf{let}\ \ \texttt{scope}_1 = 1 :: \texttt{scope}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \texttt{scope}_2 = ((\texttt{hd scope})+1) :: (\texttt{tl scope})\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (e'_1,\tau_1,\theta_1,\omega s_1,\Omega s_1) = \mathcal{W}_{Oasis}(\Theta,\Gamma,\texttt{scope}_1,e_1)\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \Omega\ = \texttt{RS\_NONE}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (\sigma,\omega s'_1)\ \texttt{=} \\
\qquad\qquad\quad \mathbf{if}\ \ \textit{transparent}(e_1)\ \ \mathbf{then} \\
\qquad\qquad\qquad \Omega\ \leftarrow \texttt{RS\_OPEN}(\mathbf{false}, \omega s_1, \texttt{scope})\texttt{;} \\
\qquad\qquad\qquad (\textit{gen}(\theta_1\Theta,\tau_1), []) \\
\qquad\qquad\quad \mathbf{else}\ (\tau_1,\omega s_1)\ \ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \Theta' = \theta_1\Theta_v\cup\{v{\mapsto}((\sigma,\Omega) :: \theta_1\Theta(v))\}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ \Gamma' = \theta_1\Gamma_v\cup\{v{\mapsto}\textit{templ}(\Theta'(v))\}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (e'_2,\tau_2,\theta_2,\omega s_2,\Omega s_2) = \mathcal{W}_{Oasis}(\Theta',\Gamma',\texttt{scope}_2,e_2)\ \mathbf{in} \\
\qquad\qquad (\texttt{let}\ \ v:(\theta_2\sigma,\Omega) = \theta_2 e'_1\ \ \texttt{in}\ \ e'_2,\ \ \tau_2, \theta_2\circ\theta_1, \theta_2\omega s'_1 @ \omega s_2, \theta_2\Omega :: (\theta_2\Omega s_1 @ \Omega s_2))
\end{array}
$$

--Continued in Figure 5.9 ...

Figure 5.7: Oasis type inference algorithm $\mathcal{W}_{Oasis}$

2. The type of the expression, as in $\mathcal{W}_{ML}$.

3. A substitution, as in $\mathcal{W}_{ML}$.

4. A list of escaping overload restrictions. The annotation task of $\mathcal{W}_{Oasis}$ includes marking points in the AST, called wrap points, that capture overload restrictions, and propagating restrictions through the expression to the wrap points. An overload restriction escapes if it originates at a point in the expression that is not enclosed by a wrap point; such restrictions will not be allowed to abstract during a resolution/propagation step.

5. A list of overload restriction sets, corresponding to the wrap points in the expression. Each restriction set represents a partial call graph. The solve step will interpret the call graphs in order to infer the flow of implicit overload parameters resulting from abstracted restrictions.

The remainder of this section will provide commentary on the pseudo-code in figures 5.7 and 5.9, with emphasis placed on the differences between $\mathcal{W}_{Oasis}$ and $\mathcal{W}_{ML}$.

5.4.1 Variable Expressions (lines 3–6)

Line 3 projects the variable's overload template out of the template environment. Line 4 creates a substitution that maps each of the bound type variables of the template to fresh type variable. The substitution produces a fresh instantiation of the template (lines 5 and 6). For variables not bound by let, there will typically be one instance of the variable in the instance environment, and the template will be the same as the type of the sole instance; in this case, the type assignment to variables is equivalent to what $\mathcal{W}_{ML}$ would assign.

Line 5 creates a new overload restriction corresponding to the variable expression. This restriction is returned in the first field of the result tuple (line 6); this makes it easy for the transform step convert the Oasis variable expression into an plain ML expression. If the solution of the restriction results in resolving and subsequently unrolling nested restrictions, then the transformation step can convert the unrolled restrictions into overload parameters and form a function application expression from the results.

5.4.2 Function Application Expressions (lines 7–11)

$\mathcal{W}_{Oasis}$ types function applications in the same manner as $\mathcal{W}_{ML}$. In line 11, the @ symbol is the ML list concatenation operator; escaping overload restrictions and wrap point restriction sets simply propagate up through an application expression.

5.4.3 Function Definition Expressions (lines 12–16)

A function definition in Oasis binds the parameter variable such that it hides any previous instances of the same variable (line 13) and has a template with the same monomorphic type as the parameter (line 14). Unlike ML, it is possible that a subsequent let-bound instance might coexist with the parameter. Initial experience with compiling Caml-Light under Oasis indicates that this does not occur frequently, so that in the common case, $\mathcal{W}_{Oasis}$ typing usually corresponds to $\mathcal{W}_{ML}$ typing for function parameters. The RS_NONE instance set (line 13) denotes that function parameters may not capture overload restrictions.

5.4.4 Non-recursive Let Expressions (lines 18–30)

Lines 18 and 19 generate new scope numbers for e_1 and E_2 respectively. Typing the binding expression e_1 is straightforward for a non recursive let (line 20).

Unlike as in chapter 4, I do not assume that the input program is pre-processed according to the Wright rule. There are two practical reasons for forgoing a pre-processor.

First, the pre-processor would transform the AST so that it would be difficult to do a good job of error reporting in later stages of the compiler. Second, the pre-processor would impose a performance penalty.

Lines 21–26 effect the Oasis version of the Wright restriction. If the binding expression e_1 is transparent, then the instance of v appearing in the context of the body expression e_2 will capture the overload restrictions of e_1, and its type will be generalized (lines 24–25). Otherwise, the instance type will not be generalized and the overload restrictions of e_1 are allowed to escape (line 26).

Line 27 constructs the instance environment to be used in typing the body. Line 28 constructs the template environment.

5.4.5 Templates

Figure 5.8 contains the algorithm for overload template construction. The goal of the algorithm is to calculate a template in a manner conforming to the specification of section 4.3.1. This section will informally describe how the algorithm works. Formal proof of conformance to the specification is left as future work.

In the general case, a template is constructed for a set of instances, but the special case of constructing a template for a pair of instances is enough to construct a general case algorithm. Templ_pair and templ_var solve the special case, while fold_templ and templ extend to the general case.

Templ_pair traverses the pair of types (τ_1, τ_2) in parallel, constructing the template as it goes. When components of the two types match, the template gets the same type (line 4). If components clash, then templ_var is used to put a type variable into the template (lines 5 and 10).

The two substitutions (θ_1, θ_2) record how the new type variables in the constructed template correspond to components of (τ_1, τ_2). If a particular clash occurs for the first

```
let_rec templ_pair = fun (τ1,τ2,θ1,θ2) ⇒
   match (τ1,τ2) with
     (α1,α2) ⇒
        if α1 = α2 then (α1,θ1,θ2)
        else templ_var(τ1,τ2,θ1,θ2)
   | (τ1 → τ2,τ'1 → τ'2) ⇒
      let (τ''1,θ'1,θ'2) = templ_pair(τ1,τ'1,θ1,θ2) in
      let (τ''2,θ''1,θ''2) = templ_pair(τ2,τ'2,θ'1,θ'2) in
      (τ''1 → τ''2,θ''1,θ''2)
   | (_,_) ⇒ templ_var(τ1,τ2,θ1,θ2)

and templ_var = fun (τ1,τ2,θ1,θ2) ⇒
   if ∃τ.θ1α = τ1∧θ2α = τ2 then (α,θ1,θ2)
   else (α,θ1 ∪ {α↦τ1},θ2 ∪ {α↦τ2}) ⟨where α is fresh⟩

and fold_templ = fun (τ,τs) ⇒
   match τs with
     [] ⇒ τ
   | (τ' :: τs') ⇒
      let (τ'',_,_) = templ_pair(τ,τ',{},{}) in
      fold_templ(τ'',τs')

and templ = fun (τs) ⇒ fold_templ(hd(τs),tl(τs))
 ;;
```

Figure 5.8: Template Algorithm

time, templ_var generates a new type variable (line 14). When a clash repeats, templ_var will reuse the existing type variable (line 13).

I now note a discrepancy between the semantics defined by the Oasis type rules for let and the corresponding parts of the $\mathcal{W}_{Oasis}$. The template is calculated by the instance set as it appears up to a given point in typing. It is possible that instances have free variables in their instance types, such that the the template algorithm will recognize a type clash in constructing the template, producing a bound variable in the template. Subsequent typing could cause clashing free type variables to unify to the same type. Had the template algorithm known this, it would not have seen a clash. In this way, the template assigned might in fact be more general than what is specified by the non-deterministic Oasis type rules. I believe it would be non-trivial to try to re-formulate the type rule to express this aspect of the algorithm's behavior. I point out the discrepancy not as a deficiency of the algorithm, but rather a problem with the specification method of chapter 4. Perhaps formulating the type rules according to another specification method, such as small step reduction semantics, might be better.

5.4.6 Recursive Let Expressions (lines 27–44)

A recursive let differs from the non-recursive case only in how the binding expression e_1 is typed. Lines 29–34 effect proper typing of a recursive binding much in the same manner as $\mathcal{W}_{ML}$. The variable v is associated with a fresh type variable (lines 31–32) in typing e_1 (line 33). Then the unification algorithm mgu enforces that the inferred type of v matches the inferred type of e_1 (line 34).

We note that the outcome of the template algorithm in line 32 falls into one of two cases here. If v did not occur in the initial instance environment Θ, then the template will turn out to be α from line 29; in this case all occurrences of v within e_1 will be forced to the same type as in $\mathcal{W}_{ML}$. If v *did* occur in Θ, the template is guaranteed to become $\forall\alpha.\alpha$, which means each occurrence of v within e_1 will get a completely fresh type

-- ...*Continued from Figure 5.7*
$|$ $\texttt{let}_{rec}\ v = e_1\ \ in\ \ e_2\ \Rightarrow$
  let $\texttt{scope}_1 = 1 :: \texttt{scope}$ **in**
  let $\texttt{scope}_2 = ((\texttt{hd scope}) + 1) :: (\texttt{tl scope})$ **in**
  let α $\langle$*be a fresh type variable*$\rangle$ **in**
  let Ω = RS_NONE **in**
  let $\Theta' = \Theta_{\overline{v}} \cup \{v \mapsto ((\alpha, \Omega) :: \Theta(v))\}$ **in**
  let $\Gamma' = \Gamma_{\overline{v}} \cup \{v \mapsto templ(\Theta'(v))\}$ **in**
  let $(e_1', \tau_1, \theta_1, \omega s_1, \Omega s_1) = \mathcal{W}_{Oasis}(\Theta', \Gamma', \texttt{scope}_1, e_1)$ **in**
  let $\theta_2 = mgu(\tau_1, \theta_1\alpha)$ **in**
  let $(\sigma, \omega s_1') =$
    if $transparent(e_1)$ **then**
      $\Omega \leftarrow$ RS_OPEN(**false**, $\theta_2\omega s_1$, scope);
      $(genexp(\theta_2\theta_1\Theta, \theta_2\tau_1), [])$
    else $(\theta_2\tau_1, \theta_2\omega s_1)$ **in**
  let $\Theta'' = \theta_2\theta_1\Theta_{\overline{v}} \cup \{v \mapsto ((\sigma, \Omega) :: \theta_2\theta_1\Theta(v))\}$ **in**
  let $\Gamma'' = \theta_2\theta_1\Gamma_{\overline{v}} \cup \{v \mapsto templ(\Theta''(v))\}$ **in**
  let $(e_2', \tau_2, \theta_3, \omega s_2, \Omega s_2) = \mathcal{W}_{Oasis}(\Theta'', \Gamma'', \texttt{scope}_2, e_2)$ **in**
  $(\texttt{let}_{rec}\ v : (\theta_3\sigma, \Omega) = \theta_3\theta_2 e_1'\ \texttt{in}\ e_2',$
    $\tau_2, \theta_3 \circ \theta_2 \circ \theta_1, \omega s_1' @ \omega s_2, \Omega :: (\Omega s_1' @ \Omega s_2))$
;;

Figure 5.9: $\mathcal{W}_{Oasis}$ (continued)

```
let solve_overloading = fun wraps ⇒
   let_rec solve = fun (count,θ) ⇒
      if count ≥ max_iterations then raise InfiniteLoop;
      let ωs = collect_rstns wraps in
      let (unrolled,θ') = resolve_rstns θωs in --fig. 5.11
      if |θ'| > 0 ∨ unrolled
      then solve(count + 1,θ' ∘ θ)
      else
         let θ'' = filter_nonimprovements((θ' ∘ θ),ωs) in --fig. 5.15
         abs_rstns(θ''ωs); --fig. 5.16
         if prop_rstns(θ''ωs) --fig. 5.17
         then solve(count + 1,θ'')
         else θ'' in
   solve(0,{})
;;
```

Figure 5.10: Resolve-Propagate algorithm

through instantiation. This represents a worst case for templates as $\forall\alpha.\alpha$ does nothing to help guide the resulting overload restriction to a resolvable type.

5.5 Overload Solution Step

This section will describe the algorithms for the overload solution step of Oasis. The general strategy is in the spirit of the Liberal Uncoupled resolution policy for parametric overloading described in [25].

Figure 5.10 contains the algorithm for the overload solution step. The input parameter to `solve_overloading`, `wraps`, comes from the output of the type inference step. As described in section 5.4, `wraps` is a list of call graphs for the proper let bindings of the input program. The result of `solve_overloading` is a substitution representing the type assignments made in resolving overload restrictions. The domain of the substitution will contain only type variables which were uncoupled after type inference. There are also side-effects generated by the solution step: at the end of a successful solve step, all overload restrictions will have either have status resolved (ST_RSLVD) or abstracted(ST_ABSD).

The `solve_overloading` function contains a single iterator function `solve`. The number of iterations is limited to an empirically chosen value `max_iterations`; if the number of iterations exceeds `max_iterations`, then the input program is declared erroneous (line 3). See section 3.7.3 for a discussion of non-termination.

The `collect_rstns` function (line 4) does a simple traversal of the call graphs to collect all of the overload restrictions contained within into a flat list. This list may grow through subsequent iterations as the resolved restrictions unroll new restrictions and abstracted restrictions propagate through the call graph.

Lines 5–7 effect the first phase of `solve`: find solutions to uncoupled type variables, which cause overload restrictions to resolve, as long as the previous step produced solutions or caused new restrictions to unroll. The solution process works so that an application of one solution can cause others to follow. Unrolling new restrictions can also cause new solutions, since each new restriction will likely induce solving some of the uncoupled type variables against different instance sets. This phase repeats until all solutions are exhausted (line 8). Section 5.5.1 will present the details of `resolve_rstns` and associated algorithms.

Lines 9–12 effect the second phase of `solve`: propagation. Any restrictions that still have open (ST_OPEN) status after the resolve phase must abstract (line 10). These restrictions are interpreted as new formal parameters attached to the appropriate wrap points. The call to `prop_rstns` (line 11) will then attempt to propagate formal parameters through the call graph to generate new restrictions for the actual parameters. If propagation occurs, another round of the resolve phase is in order (line 12). Otherwise, there are no remaining open overload restrictions and so the overload solving step is complete (line 13). Section 5.5.2 will describe the complete details of the propagation phase algorithms.

```
1       let resolve_rstns = fun ωs ⇒
          let (rstns_unrolled,solns) = (false,{}) in
          let resolve = fun ω ⇒
            let (rstn_solns,matches) = resolve_rstn(rstn) in --fig. 5.12
5           solns ← range-∩(solns,rstn_solns);
            if |matches| = 1 ∧ |rstn_solns| = 0
            then rstns_unrolled ←
              rstns_unrolled ∨ resolve_n_unroll(ω,hd matches) --fig. 5.13
          | _ ⇒ () in
10        app resolve ωs;
          let θ= {(α↦τ) | (α↦{τ})∈solns} in--Occurs check here too ...
          (rstns_unrolled,θ)
      ;;
```

Figure 5.11: Resolve Overload Restrictions

5.5.1 Resolve Phase

Figure 5.11 contains `resolve_rstns`. This algorithm iterates over the list of overload restrictions, collecting solutions to uncoupled type variables, and resolving restrictions when appropriate. The function `resolve_rstn` produces a set of type variable solutions and matching overload instances for a single overload restriction (line 4). The call to $range - \cap$ (line 5) merges type variable solutions of the single solution with the cumulative result of preceding solutions; if a type variable has solutions in both sets, then the result of the merge will be the intersection of both solutions. In this way, a type variable solution survives only if it satisfies all overload restrictions. At the end of the iteration, the unique solutions (solutions sets with size one) are transformed into a substitution (line 10). These unique solutions represent the *obvious choice* of the liberal uncoupled resolution policy; the overload solving process will continue by speculating that this substitution will lead to further resolutions. The transformation includes an occurs check to prevent cycles from appearing in the substitution.

Line 6 decides if a restriction is ready to resolve: when there is a unique matching instance, and the match occurs without requiring any type variable solutions. Line

```
1    let resolve_rstn = fun (υ,τ, ST_OPEN(Θ,_)) ⇒
       let (rstn_solns,matches) = ({},{}) in
       let check_inst = fun (∀α^(k).τ',Σ) ⇒
         try
5          let θ = mgu(τ',τ) in
           let inst_soln = restrict_domain(θ, tyvars(∀α^(k).τ) ∪ tyvars(τ)) in
           rstn_solns ← range-∪ (rstn_solns,inst_soln)
           matches ← matches ∪ {(∀α^(k).τ',Σ)}
         with Unify ⇒ () in
10     let avail_instances = filter_overlaps Θ(υ) in
       app check_inst avail_instances;
       (rstn_solns,matches)
     ;;
```

Figure 5.12: Solve a single overload restriciton

7 calls **resolve_n_unroll** to effect the change of status and unroll new restrictions if necessary. The variable **rstns_unrolled** tracks whether new restrictions are actually generated, which later determines whether the resolve phase should loop or continue with the propagation phase.

Figure 5.12 contains pseudo-code for **resolve_rstn**.

The algorithm iterates **check_inst** against the set of the of available instances (line 10). The available instances are determined by projecting from the instance environment, and removing overlapping instances (line 10). The instance lists are stored so as to preserve the lexical order of instance declarations (most recent declaration at the front of the list); an instance is hidden because of overlap if it can unify with any instance that is closer to the front of the list.

An instance can be made to match the restriction if the type of the instance can be made to instantiate the type of the restriction; if so, the call to **mgu** will return a substitution that indicates how to perform the instantiation. Line 6 discards mappings on the bound type variables of the instance type, as the solution process is concerned only with uncoupled type variables. Lines 6 and 7 update the type variables solutions

```
1    let resolve_n_unroll = fun (((v,τ,status) as ω),((∀α^(k).τ' as σ),Ω)) ⇒
       let ST_OPEN(Θ,scope) = status in
       match Ω with
         RS_OPEN(wrap_desc) ⇒
5         let rslvd_kind =
            if prefix(wrap_desc.scope, scope)
            then RK_OPEN_NO_INST(wrap_desc)
            else
              let θ = mgu(τ',τ) in
10            RK_OPEN_INST(θ, [],(Θ,scope),wrap_desc) in
          status ← ST_RSLVD(σ,rslvd_kind);
          false
       | RS_CLOSED(ωs) ⇒
          let θ = mgu(τ',τ) in--fig. 5.14
15        let (_,Π) =
            inst_rstns(θ,(Θ,scope),σ,[],ωs) in --fig. 5.14
          let ωs' = range(Π) in
          ω.status ← ST_RSLVD(σ,RK_CLOSED(ωs'));
          |ωs'| > 0
20   ;;
```

Figure 5.13: Resolve Overload Restruction and Unroll

and the set of matched instances as appropriate. If unification (mgu) failed, line 9 will catch the resulting exception, effectively ignoring the instance as a non-match.

Figure 5.13 contains the `resolve_n_unroll` function, which given an open overload restriction (ST_OPEN) and its matching overload instance, will set change the status of the restriction to resolved (ST_RSLVD). Recall from figure 5.5, the argument to ST_RSLVD is a resolved kind (RslvKind) descriptor. RslvKind relates the circumstances of resolution to how interpretation of the call graph should occur according to three cases.

Lines 5–12 handle the first two cases which correspond to resolving to an instance with an open restriction set (RS_OPEN); in other words, the target instance is declared by a proper let somewhere in the toplevel phrase that the typing step is currently processing. The two cases are distinguished (line 6) by whether the calling context of the

```
let inst_rstns = fun (θ,ctx_info,∀α^(k).τ,Π,ωs) ⇒
   let uncoupled_tyvars = tyvars(ωs) \ tyvars(τ) in
   let α'^(j) = uncoupled_tyvars \ dom(θ) in
   let θ' = {α'_i↦α''_i} ⟨where α''_i are fresh type variables⟩ in
   let θ'' = θ' ∘ θ in
   let inst_rstn = fun Π ⇒ fun ((v,τ,ST_ABSD) as ω) ⇒
      if ω∈dom(Π) then Π
      else
         let ω' = (v,θ''τ,ST_OPEN(ctx_info)) in
         Π∪{ω↦ω'} in
   let Π' = fold inst_rstn Π ωs in
   (θ'',Π')
;;
```

Figure 5.14: Overload Restriction Instantiation Algorithm

overload restriction is within the recursive binding of the target instance, as determined by the respective scope numbers (section 5.3.1 discusses scope numbers). If the restriction is recursive , then (line 7) creates a RK_OPEN_NO_INST resolve kind descriptor. Otherwise, lines 9–10 create a RK_OPEN_INST resolve kind descriptor. The information in RK_OPEN_INST will allow the creation of the actual parameters corresponding to the formal parameters that are attached, during the propagation phase, to the wrap descriptor of the target instance. Line 12 sets the return value to false; the return value indicates that no new overload restrictions are generated.

Lines 14–18 deal with the third case: the target instance has a closed restriction set (RS_CLOSED). A closed restriction set either means that the target instance is not bound by a proper let (and so has no overload restrictions) or that the target instance is bound in a previous toplevel phrase. Line 18 sets the return value according to whether any restrictions were unrolled.

Figure 5.14 contains the `inst_rstns` algorithm. The purpose of inst_rstns is to create new open overload restrictions which represent the actual parameters (actuals) corresponding to formal parameters defined by a given list abstracted overload restrictions (formals). The actuals must be created in a manner consistent with the Oasis specification of restricted instantiation (see section 4.3.1). The algorithm takes a tuple of five input parameters:

- A substitution θ that records type variable instantiations. Note that this function may be called multiple times for a given node in the graph, because of the iterative nature of the overload solving step. The set of formal parameters may grow over successive iterations, so that new actual parameters must also be generated. The input substitution allows the generation of new actuals to instantiate type variables in a manner consistent with previous actual parameters.
- A propagation mapping Π that records previous formal-actual propagations. This is used to identify new formals.
- The context information of the call-site, to attach on newly generated actual parameters.
- The type scheme σ of the called instance. This will be used to identify uncoupled type variables in formals.
- A list of formals ωs.

Lines 2–5 build the substitution of the restricted instantiation θ''. The substitution is constructed to extend the input substitution θ with any new uncoupled type variables; these new type variables come from restrictions in ωs that have arrived during the latest iteration. The function `inst_rstn` produces the new propagation mapping (lines 6–11). If a formal parameter is already in the input propagation mapping, the result is unchanged (line 7). Otherwise a new actual parameter is generated and a mapping inserted into the result (lines 9–10).

5.5.2 Propagation Phase

The resolve phase constructs solutions in a speculative manner; each iteration analyzes uncoupled type variables individually, often causing a cascade of solutions which lead to obvious resolutions. However, there is no guarantee inherent in the algorithm that a type variable solution chosen in the reslove phase will actually cause or help any restriction to resolve. For example, suppose we have the following scenario:

- At the begining of the resolution phase, an open overload restriction has two uncoupled type variables in its type. These two type variables do not appear in the type of any other overload restriction.
- The instances available are such that at the end of the resolve phase, only one of the variables has been solved, and the restriction itself has not resolved.

When the overload restriction in this example is abstracted, the effect will be that the remaining uncoupled type variable is generalized; this effect follows from how restricted instantiation is defined. The other uncoupled type variable has been solved, so it is not generalized. When the restriction propagates, it will generate a new open restriction, but the new restriction will be solved against a different instance set. The solution found for the uncoupled type variable in the old restriction may not be valid in the context of the new restriction–given that the original restriction didn't resolve anyway, it would have been better to leave the type variable to generalize.

The function `filter_nonimprovements` in figure 5.15 removes *non-improving solutions* prior to next step of the propagation phase: abstracting the remaining open restrictions. The algorithm works by scanning all resolved overload restrictions, and checking which members of the solution substitution are necessary to cause the restrictions to resolve.

Figure 5.16 contains the function `abs_rstns`. There a few special error cases to consider before the restriction can abstract:

```
let filter_nonimprovements = fun (θ,rstns) ⇒
   let filter = fun rstn ⇒
      match rstn.status with
        ST_RSLVD(σ,_) ⇒
          {(α ↦ τ) | |snd(solve_rstn((θ \ {(α↦τ)})rstn))| ≠ 1}
      | _ ⇒ {} in
   ⋃_{rstn∈rstns} filter(rstn)
;;
```

Figure 5.15: Filter non-improving substitutions

```
let abs_rstns = fun ωs ⇒
   let result = false in
   let abs_rstn = fun ω ⇒
      match ω.status with
        ST_OPEN(_,scope) ⇒
           if |tyvars(ω.type)| = 0 then raise RstnMustResolve;
           let (rstn_solns,matches) = resolve_rstn(ω) in --fig. 5.12
           if |matches| > 1 then raise RstnAmbiguous;
           if scope= toplevel_scope then raise RstnMustResolve;
           ω.status← ST_ABSD;
           result ← true
      | _ ⇒ () in
   app abs_rstn ωs;
   result
;;
```

Figure 5.16: Make Overload Restrictions Abstracted

```
1   let prop_rstns = fun ωs ⇒
      let result = false in
      let prop = fun ω ⇒
        match ω.status with
5         ST_RSLVD(σ, RK_OPEN_INST(θ,Π, (Θ,scope),wrap_desc)) ⇒
            let ωs' = collect_absd_rstns(wrap_desc) in --fig. 5.18
            if |ωs'| > |Π| then
              result ← true;
              let (θ',Π') = inst_rstns(θ,(Θ,scope),σ,Π, ωs') in
10            ω.status ←
                ST_RSLVD(σ,
                  RK_OPEN_INST(θ',Π', (Θ,scope),wrap_desc))
            else ()
        | _ ⇒ () in
15    app prop ωs;
      result
    ;;
```

Figure 5.17: Propagate Abstracted Overload Restrictions

- If the type of the restriction has no type variables, then it can not abstract (line 6): it just does not make sense for an abstract restriction to have a ground type.
- If after the resolve phase the restriction could not be narrowed down to one match, then the restriction is ambiguous (lines 7–8). This may seem strange, but not so. A restriction with uncoupled type variables in its type, such that there are multiple solutions, will likely have no match since type variable solutions are only accepted if unique.
- If the restriction's calling context is not enclosed by a proper let binding, it can not abstract (line 9).

Figure 5.17 contains `prop_rstns`. The algorithm scans the set of all overload restrictions for those which resolved so as to require interpretive unrolling (line 5). The call to `collect_absd_rstns` (described next) will gather the formal parameters of the target. If the number of formal parameters is larger than the propagation mapping (line

```
1   let collect_absd_rstns = fun wrap_desc ⇒
      let_rec collect_wrap = fun (false as mark,_,ωs) ⇒
        mark ← true;
        collect_rstns(ωs)
5     | (true, _, _) ⇒ {}
      and collect_rstns = fun ωs ⇒
        ⋃_{ω∈ωs} collect_rstn(ω)
      and collect_rstn = fun ω ⇒
        match ω.status with
10        ST_ABSD ⇒ {ω}
        | ST_RSLVD(RK_OPEN_NO_INST(wrap_desc) ⇒
            collect_wrap(wrap_desc)
        | ST_RSLVD(RK_OPEN_INST(_,ωs,_,_) ⇒
            collect_rstns(ωs)
15      | ST_OPEN(_) ⇒ {} in
      let result = collect_wrap(wrap_desc) in
      sweep_wrap(wrap_desc);
      set_to_list(result) --Ensure canonical ordering
    ;;
```

Figure 5.18: Collect Abstracted Overload Restrictions

7), then new actual parameters are required. Line 9 calls `inst_rstns` to generate the actual parameters, and lines 10–12 update the status of the restriction with the result.

The function `collect_absd_rstns`, in figure 5.18, does a mark and sweep traversal of a call graph. The input parameter is a wrap descriptor representing the root of call graph. The traversal needs to mark because it crosses boundaries from the call graph of one definition to another (line 12); since the definitions may be fully recursive, the call graphs may be cyclic. Line 18 converts the result from a set representation to a list. The ordering of the elements in the list is chosen in a canonical fashion, so that a calling convention for implicit overload parameters can be implemented during the syntax transformation.

5.6 Syntax Transformation

The final step in the typing step of Oasis is the syntax transformation: the objective of the transformation is to produce a plain ML AST suitable for execution according to the plain ML dynamic semantics.

Two functions, `transform_expr` and `transform_rstn`, are given in figure 5.19. The entry point for the transformation step is `transform_expr`; it traverses the AST performing three transformations:

1. All identifiers, variable expressions and variable patterns, were annotated with a type by the typing step. In the transformation step, each identifier is transformed into a canonical representation reflecting the identifier and the type. Oasisc uses the identifier concatenated with a string representation of the type as the canonical representation.

2. Each proper let binding has a `wrap_desc` which is used to collect abstracted overload restriction list of the binding expression. If the list is non-empty, the binding expression is wrapped: it is converted into a function that has a formal parameter for each abstracted overload restriction. The formals have names adhering to the same canonical representation as all other identifiers.

3. Overload restriction expressions are converted to variable expressions. If an overload restriction resolves to an instance with a non-empty overload restriction set, then the instance set produces a list of argument expressions, and the variable expression is applied to the argument expressions.

```
let_rec transform_expr = fun e ⇒
   match e with
     ω ⇒ transform_rstn(ω)
   | e1 e2 ⇒ transform_expr(e1) transform_expr(e2)
   | λ(v : τ).e ⇒
      let v' = canon_name_munge(v,τ) in
      λv'.transform_expr(e)
   | let_x (v : σ,RS_NONE) = e1 in e2 ⇒
      let v' = canon_name_munge(v,σ) in
      let_x v' = transform_expr(e1) in transform_expr(e2)
   | let_x (v : σ,RS_OPEN(wrap_desc)) = e1 in e2 ⇒
      let v' = canon_name_munge(v,σ) in
      let ωs= collect_absd_rstns(wrap_desc) in
      let wrap_exp =
         fun e ⇒ fun (v,τ,status) ⇒ λcanon_name_munge(v,τ).e in
      let e1' = fold wrap_exp e1 ωs in
      let_x v' = e1' in transform_expr(e2)
and transform_rstn = fun (v,τ,status) ⇒
   let (σ,ωs) =
      match status with
        ST_ABSD ⇒ (τ,[])
      | ST_RSLVD(σ,rslvd_kind) ⇒
         let ωs= match rslvd_kind with
                RK_CLOSED(ωs) ⇒ ωs
              | RK_OPEN_INST(_,Π,_,wrap_desc) ⇒
                 let ωs= collect_absd_rstns wrap_desc in
                 map (fun ω ⇒ Π(ω)) ωs
              | RK_OPEN_NO_INST(wrap_desc) ⇒
                 collect_absd_rstns(wrap_desc) in --fig. 5.18
         (σ,ωs) in
   let rstn_expr = canon_name_munge(v,σ) in
   if |ωs| = 0
   then rstn_expr
   else rstn_expr (map transform_rstn ωs)
;;
```

Figure 5.19: Transformation Algorithm

Chapter 6

Conclusion

This thesis presents a useful and feasible proposal for parametric overloading in ML. A prototype compiler, Oasisc was implemented and provided valuable insight into the design issues of Oasis.

The work so far on Oasis has been mostly based on experimentation. This has proved fruitful in examining out several design issues. It might be argued that the approach here was wrong, in that a stronger emphasis on design of semantics should precede implementation. I don't discount the value of strong semantics, but I would contend that neither activity should completely precede the other. As an example, I would argue that a result of this thesis is confirmation that allowing non-termination in the overloading algorithm is actually the best alternative, and that had there been no implementation it is highly unlikely that I would have made this argument. This is analogous to the situation with respect to algorithmic complexity of ML type inference. Had it been widely known that ML type inference has exponential complexity before there were any ML implementations, it probably would have been accepted that type inference was impractical. In practice, it is clear that both scenarios, exponential blowup and non-termination, do not arise very often at all. That said, formulating a proper set of formal semantics should be a top priority for future work on Oasis.

A weakness of Oasis is the relative weight of the extension. The algorithms for overload processing now form a very significant proportion of the overall type system. I would propose that the analysis for parametric overloading could be extended naturally to deal with several other analyses:

- Polymorphic equality presents a significant performance penalty in Caml-Light. Equality could be treated as a special form of overloading in Oasis. Instead of trying to solve an equality restriction, we simply propagate it until its type becomes ground. Once ground, a non-polymorphic equality parameter can be generated. If this approach turns out to perform well, other aspects of polymorphism might be lifted out of the implementation too; for example, it might be possible to use implicit parameters to avoid all requirements for boxed data structures [24].

- Exception tracking [38]. This is a mechanism where the compiler would generate warnings to indicate exceptions that might escape a given section of code, much in the vein of pattern match exhaustiveness checks performed by most ML compilers. The analysis for exception tracking would be very much like overload resolution. An exception resolves if it matches an exception bound by an enclosing handler. If an exception abstracts, then warnings can be generated.

- Static memory management [32, 1]. Research into techniques for the elimination of garbage collection is progressing at an exciting pace. The region and effect systems [30, 31] that are used as the basis for static memory management annotation have many similarities to overload solution. Merging the two into a common framework might have benefits.

The objective of pursuing these extensions, in addition to their inherent benefits, would be to leverage the post type inference analysis mechanism to more than just overloading.

Bibliography

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *PLDI95*, San Diego, California, June 1995.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, New York, August 1991. Springer-Verlag.

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

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] Glen Ditchfield. *Contextual Polymorphism.* PhD thesis, University of Waterloo, 1992.

[6] Gordon Cormack Dominic Duggan and John Ophel. Kinded type inference for parametric overloading. Technical Report CS-93-12, University of Waterloo, 1993. To appear in Acta Informatica.

[7] Dominic Duggan, Gordon Cormack, and John Ophel. Kinded type inference for parameteric overloading. *Acta Informatica*, 1995. To appear. Available as University of Waterloo Technical Report CS-93-12.

[8] John S. Fitzgerald. Unification: Specification and development. In *Case studies in systematic software development* [14], chapter 5.

[9] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James O' Toole. Report on the fx-91 programming language. Technical Report MIT/LCS/TR-531, MIT Laboratory for Computer Science, Feb 92.

[10] Cormack G.V. and Wright A.K. Type-dependent parameter inference. In *ACM SIGPLAN 90 Conference on Programming Language Design and Implementatation*, Sigplan Not, pages 127–136, 1990.

[11] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. In *ESOP*, Jan 94.

[12] Fritz Henglein and Harry Mairson. The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming (JFP)*, 4(4):435–477, October 1994.

[13] P. Hudak et al. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.

[14] Cliff B. Jones. *Case studies in systematic software development.* Prentice Hall International, Englewood Cliffs, N.J., 1989.

[15] Mark P. Jones. *Qualified Types: Theory and Practices.* PhD thesis, University of Nottingham, July 1992.

[16] Mark P. Jones. Simplifying and improving qualified types. In *Functional Programming and Computer Architecture*, San Diego, California, June 1995. ACM Press.

[17] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990. Available by anonymous FTP on ftp.inria.fr.

[18] Xavier Leroy. *The Caml Light system release 0.7.* INRIA, July 1995.

[19] Dave Mason. Poor man's functors. Personal Communication, 1995.

[20] Michel Mauny. *Functional programming using Caml Light.* INRIA, January 1995.

[21] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[22] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming and Computer Architecture*, San Diego, California, June 1995. ACM Press.

[23] Atsushi Ohori. A polymorphic record calculus and its compilation. RIMS Preprint 1013, Research Inst. for Mathematical Sciences, Kyoto University, Kyoto, Japan, April 1995.

[24] Atsushi Ohori and Tomonobu Takamizawa. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. Technical report, RIMS, Kyoto University, May 1995.

[25] John Ophel and Dominic Duggan. Multi-parameter parametric overloading. Submitted for publication, 1995.

[26] Robert Harper Robin Milner, Mads Tofte. *The Definition of Standard ML.* The MIT Press, 1990.

[27] Catherine Dubois Francois Rouaix and Pierre Weis. Extensional polymorphism. In *Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press.

[28] Jun Shen. *Parametric Overloading, Higher-Order Modules and Dynamic Dispatch: An Ada-Based Approach.* PhD thesis, University of Waterloo, 1993.

[29] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proceedings of the Seveneth Annual Symposium on Logic in Computer Science*, pages 162–163, June 1992.

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[31] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[32] M. Tofte and J. P. Talpin. Implementation of the typed lambda calculus using a stack of regions. In *Principles of Programming Languages*. ACM Press, January 1994.

[33] Sunil Vadera. Building a theory of unification. In *Case studies in systematic software development* [14], chapter 6.

[34] Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ml typability with overloading. Technical Report TR91-1210, Cornell University, Computer Science Department, May 1991.

[35] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, Jan 89.

[36] Andrew K. Wright. Design of the programming language ForceOne. Master's thesis, University of Waterloo, 1986.

[37] Andrew K. Wright. Simple imperitve polymorphism. In *Lisp and Symbolic Computation*, 1994. (to appear in late 1995).

[38] Kwangkeun Yi. Compile-time detection of uncaught exceptions in standard ml programs. In *International Static Analysis Sympoisum*, pages 238–254. Springer-Verlag, LNCS 864, September 1994.

[39] David Ziegler. Adding overloading polymorphism to C. Master's thesis, University of Waterloo, 1992.