8-2022

Revisiting Ad-hoc Polymorphism

Apurav Khare
ak2816@rit.edu

Revisiting Ad-hoc Polymorphism

by

Apurav Khare

THESIS

Presented to the Faculty of the Department of Computer Science

Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

Rochester Institute of Technology

August 2022

Revisiting Ad-hoc Polymorphism

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Arthur Nunes-Harwitt, Supervisor

Dr. Matthew Fluet, Reader

Dr. James Heliotis, Observer

Abstract

Revisiting Ad-hoc Polymorphism

Apurav Khare, M.S.
Rochester Institute of Technology, 2022

Supervisor: Dr. Arthur Nunes-Harwitt

Ad-hoc polymorphism is a type of polymorphism where different function definitions can be given the same name. Programming languages utilize constructs like Type classes and Object classes to provide a mechanism for implementing ad-hoc polymorphism.

System O, by Odersky, Wadler, and Wehr is a language which defines a dynamic semantics that supports ad-hoc polymorphism. It also describes static type checking for its programs and a transformation to the Hindley/Milner system.

In this study, we present extensions to System O by defining constructs that make the language more practical to use. We utilize the dynamic semantics to define the ability to express type classes. We define additional optimizations on the transform that aim to reduce redundant function calls at run-time and simplify the generated code.

Finally, we implement an interpreter for this programming language in Clojure, and provide several examples of programs utilizing ad-hoc polymorphism with the constructs we have defined.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Arthur Nunes-Harwitt, whose guidance and insight were paramount to pursuing the proposed work. The assessment and observations offered by him at every step were instrumental in shaping the work as we progressed through developing the thesis. Dr. Matthew Fluet and Dr. James Heliotis' valuable feedback throughout the process of developing our work was extremely helpful.

I am deeply grateful to the Computer Science Department and my graduate advisor for their continued support, advice, and resources, that have been of great help throughout my Master's degree.

Finally, I would like to thank my family for their continued support and encouragement that have made it possible for me to pursue a Master's degree.

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

Modern programming languages provide a wide array of tools to programmers for writing reusable and easy-to-understand code, with features like inheritance, encapsulation, and polymorphism. These tools also include various safety net features, like type checking, that provide feedback allowing the programmers to write code that doesn't fail at run-time.

Ad-hoc polymorphism is one such feature that allows application of polymorphic functions to arguments of different types, for example, overloading the + operator to work on integers and floating point numbers, hiding the primitive implementation of these operations from the programmer [23].

A type class [26] is a construct that supports ad-hoc polymorphism, and is implemented in languages like Haskell and Coq. It encapsulates the behavior to be overloaded, and instances of the type classes provide definitions for the overloaded operations. They are intuitive and expressive in their way of expressing and implementing the overloading behavior. Type classes also have the flexibility of separating type definitions from type class instantiation.

An alternate approach to implementing ad-hoc polymorphism is System O, defined as an extension to the Hindley/Milner system by Odersky, Wadler, and Wehr [18]. It eliminates type class declaration, allowing overloaded functions to be declared at the top program level. It also exhibits type soundness and principal type properties analogous to the Hindley/Milner system.

One approach to implementing ad-hoc polymorphism is statically during the type checking process. These implementations require some form of program transformation. For instance, Haskell implements type classes by creating dictionaries for the type class instances that contain the overloaded functions [20]. This requires more plumbing in order to provide the context beforehand [24]. For existing implementations of ad-hoc polymorphism with type classes, a dynamic semantics does not exist, which would be desirable to have for demonstrating theoretical results, and have practical applications in programming pedagogy.

Ad-hoc polymorphism can also be implemented dynamically, resolving the function instance to be called at run-time. The values passed to the function are used to decide the overloaded instance to call. Though easier to implement, this approach can have a run-time performance overhead.

In this study, we show how it is possible to construct a dynamic semantics for implementing ad-hoc polymorphism using type classes. We define this language as an extension of System O. By utilizing the static type checking for the language and augmenting that with a transformation, the run-time overhead of ad-hoc polymorphism is eliminated. Additional optimizations are made possible by the transformation that make the resulting code more efficient and easier to understand. We develop an interpreter in Clojure with these features to execute programs that utilize the ad-hoc polymorphism constructs we have introduced.

Chapter 2

Background

This study focuses on ad-hoc polymorphism in the context of functional programming languages, specifically using type classes to implement ad-hoc polymorphism. In this chapter, we cover the concepts of polymorphism and the related terminology. We then review the implementations of ad-hoc polymorphism in different paradigms, with focus on type classes and its variations. Finally, we look at the semantics for implementing ad-hoc polymorphism and their implications relevant to our study.

2.1 Ad-hoc Polymorphism

In programming languages, polymorphism is a mechanism that provides a single interface to entities of different types [5]. Strachey distinguished between the two major kinds of polymorphism [23]:

- *Parametric Polymorphism*, where a function works uniformly across a range of types, which typically exhibit a common structure.
- *Ad-hoc Polymorphism*, where a function works, or appears to work, on different types that may not exhibit a common structure, and the function may behave in different ways for each type.

Cardelli and Wegner further refined this categorization by introducing a hierarchy of polymorphism, which, at the top level, comprises *universal polymorphism* and *ad-hoc polymorphism* [5]. This categorization is described below:

- *Universal Polymorphism* is a generalization of functions that can a be applied to an infinite number of types, given that they all exhibit a common structure. It is further categorized into the following:
  - *Parametric Polymorphism*, where functions determine the type of the arguments for each application with implicit or explicit type parameters. Functions exhibiting parametric polymorphism are also called *generic functions*. An example of parametric polymorphism is the generic function length, and can be defined with the type definition List $\alpha \to$ Integer. Here, α can be any type, allowing length to work on lists of integers, floats, or any other type.
  - *Inclusion Polymorphism*, where an object may belong to multiple classes which may not be disjoint, i.e., there is *inclusion* of classes. It is used to model subtypes and inheritance. It is also called run-time polymorphism, because the right function to be invoked is determined at run-time.

    Consider the classes FileWriter and StreamWriter, defined under a Writer class that implement the virtual write function. When a program calls the function write on an instance of Writer, the actual function to be called would be determined at run-time, based on the concrete instance that the variable holds.

- *Ad-hoc polymorphism* works on a finite set of potentially unrelated types. It is worth noting that ad-hoc polymorphism gets its name from the fact that "there is no single systematic way of determining the type of the result from the type of the arguments", and that there may be rules that reduce the number of cases, but they are themselves ad-hoc in scope and content [23]. Ad-hoc polymorphism is further categorized into the following:
  - *Overloading*, where the same variable name is used to denote different functions. An example of overloading is the + operator, which can be applied to integers and floating point, and in some implementations, strings and lists.
  - *Coercion*, which is a semantic operation used to convert arguments to a type that the function expects. It can be implemented statically or dynamically.

Both coercion and overloading can be used to implement ad-hoc polymorphism, depending on the implementation details of the language. Consider the expression $3 + 4.0$; one way to implement this operation is by *overloading* the + operator to accept one integer and one double. Another option is to *coerce* the integer argument to a double type in this case.

Sometimes, “overloading” is used to refer to static polymorphism, which can be resolved at compile time, and “polymorphism” is used to refer to the dynamic case of inclusion polymorphism. In this study, we focus on overloading, and it is taken to be synonymous with ad-hoc polymorphism, unless otherwise specified.

2.2 Approaches to implementing Ad-hoc polymorphism

Ad-hoc polymorphism implementations solve the problem of selecting which function definition to call when an overloaded function is called, without having to specify the exact function being called. In this section, we review existing practical and theoretical implementations of ad-hoc polymorphism, with focus on type classes and its variations.

2.2.1 Object classes

In Object-Oriented languages, an *object* is an encapsulation of data and behavior, or methods, that can be invoked on the object. A “class” acts as a template for objects. It is the type, and all instances of the class have that type [12].

Ad-hoc polymorphism can be achieved by defining multiple methods in the same class that have the same return type. The overloaded function to call is resolved at compile time. The overloaded functions are allowed to vary in the following ways:

- The number of arguments
- The order of arguments
- The type of arguments

Some languages allow for the return type of the function to be different, as long as it is not the only thing that is different. Listing 2.1 shows an example of ad-hoc polymorphism in Java, using method overloading. The operator + is defined to operate on arguments of primitive types integer and double. The call to the overloaded function sum is resolved at compile time, using the arguments passed to the function.

```
public class Demo {

  public static int sum(int x, int y)
  {
    return (x + y);
  }

  public static double sum(double x, double y)
  {
    return (x + y);
  }

  public static void main(String args[])
  {
    System.out.println(Demo.sum(10, 20));
    System.out.println(Demo.sum(10.5, 20.5));
  }
}
```

Listing 2.1: Ad-hoc polymorphism in Java

Inclusion polymorphism is achieved through inheritance. The behavior is also referred to as overriding. The function to be called is then determined at run-time based on the instance of the class that the function was called on. Inheritance is usually restricted to class definitions, meaning that overriding behavior across a class can be defined only when the class is defined.

2.2.2 Type Classes

Type Classes are a construct that support ad-hoc polymorphism, introduced by Wadler and Blott, and implemented in Haskell, to allow overloading of arithmetic operators. They were introduced as a generalization of `eqtype` variables of Standard ML [26].

Type classes encapsulate the names and type signatures of overloaded functions in their declaration, and instances of these type classes implement the behavior for these functions. Parametric polymorphism is used to define type constraints, and in turn, quantify the function definition for types that instantiate the type class. An example of type class declaration and instantiation in Haskell is shown in Listing 2.2.

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a

-- The 'primitive' functions are assumed
-- to have been defined for given types
instance Num Int where
  (+)    = primitiveAddInt
  (*)    = primitiveMulInt
  negate = primitiveNegateInt

instance Num Float where
  (+)    = primitiveAddFloat
  (*)    = primitiveMulFloat
  negate = primitiveNegateFloat

square :: Num a => a -> a
square x = x * x
```

Listing 2.2: Haskell Type Class example

In the listing 2.2, the function square can be understood as a function of type a $\rightarrow$ a, for every a that belongs to the type class Num, i.e., every type a that has the functions (+), (*), and (negate) defined on it. That makes it possible to make the following function calls:

```
square 3
square 3.14
```

The programming language Axiom [25], defines algebraic constructs like groups, rings, and fields, with a type class-like construct called *Categories*. Categories can extend other categories to define overloaded behavior.

Type classes provide a concise and expressive way to implement ad-hoc polymorphism. By separating type definition and type class instantiation, type classes allow programmers to write more modular code.

Type class features have been adopted in other languages to varying degrees, such as traits in Rust, the template extension “Concepts” in C++, and generalized interfaces in Java.

2.2.3 Alternative approaches

Odersky, Wadler, and Wehr describe “System O”, an alternative approach to ad-hoc polymorphism, that eliminates type class declaration and instead allows overloaded functions to be declared directly at the top program level [18]. This allows overloaded instances to have unique type signatures, as they don’t have to conform to a defined signature.

Listing 2.3 shows an example of overloading the + operator using System O. It is worth noting in the listing that this construct allows defining overloaded instances of the + function on values of type String, as the overloading is not restricted by a type class. In a system with type classes, it would be required to define all the functions associated with the type class.

A workaround to solving this problem would be to define a more fine-grained system of type classes. However, this may not always be a possible solution. For example, a system with type classes cannot define an exponentiation function to cover all choices for basis and exponent, and would need to define different exponentiation operators for Integral and Fractional types. Even with the multi-parameter type class extension, one common exponentiation operator cannot be implemented due to ambiguity arising from cases where the exponent can be of type Int or Integer.

```
-- Define the overloaded variable
over (+)

-- The 'primitive' functions are assumed
-- to have been defined for given types
inst (+) :: Int -> Int -> Int
     (+) = primitiveAddInt

inst (+) :: String -> String -> String
     (+) = primitiveStringConcat

-- Unary plus
inst (+) :: Double -> Double
     (+) x = x
```

Listing 2.3: An example of overloaded addition in System O

System O defines a dynamic semantics for its programs, that extends the semantics for the language Exp [13] to include overloaded variables and function definitions. This extension requires all instances of the overloaded functions to be distinguished by the type of the function's first argument. This limits the overloaded functions that can be defined. For instance, in Listing 2.3, it wouldn't be possible to define another overload that represents unary plus operation of type $\mathsf{Int} \to \mathsf{Int}$.

Shields and Peyton Jones describe λ^O, a system for modeling object-oriented style ad-hoc overloading and specialization in the context of type class overloading [22]. Overloading in λ^O resembles System O, where each class specifies a single overloaded name and each instance has a single interpretation. In addition, λ^O introduces "closed classes" – classes for which no instance is defined. They are added as a part of an additional unification step during type checking, and reduce the need for type annotations by improving the inferred types. Listing 2.4 shows an example of overloading the + operation in λ^O.

```
-- Define the closed class +
class closed (+) a where a

-- The 'primitive' functions are assumed
-- to have been defined for given types
instance (+) (Int -> Int -> Int) where primitiveAddInt
instance (+) (String -> String -> String) where
   primitiveStringConcat
```

Listing 2.4: An example of overloaded addition in λ^O

Consider a function inc, that uses the constraint (+) defined above, with the signature $\mathsf{inc} :: ((+)\ (\mathsf{a} \to \mathsf{Int} \to \mathsf{b})) \Rightarrow \mathsf{a} \to \mathsf{b}$. With closed classes, the inference algorithm is able to commit to a more concrete signature $\mathsf{Int} \to \mathsf{Int}$, since the class is "closed", and can have no other instances where the second type parameter is of type Int.

λ^O also includes "overlapping instances", which allows for specialization of type class implementations. This kind of specialization is common in object-oriented languages. It does so by raising an error only when an overlapping instance arises that creates an ambiguity in selecting an overloaded instance. For example, the instances in Listing 2.5 would be rejected in Haskell because the constraint Overlap (Int, Int) matches both declarations. In contrast, λ^O would raise an error only if the constraint Overlap (Int, Int) comes up in practice.

```
instance Eq a => Overlap (Int, a) where ...
instance Eq a => Overlap (a, Int) where ...
```

Listing 2.5: Overlapping instances in λ^O

Morris proposes a theoretical formulation for implementing ad-hoc polymorphism [15], building on Ohori's simple semantics for ML polymorphism [19]. In this approach, polymorphic expressions are interpreted as type-indexed collections of monomorphic terms. This is referred to as a specialization-based approach, because it relates polymorphic terms to their ground-typed specializations. A functional language called H^- is introduced, with denotational semantics using the specialization-based approach.

2.3 Semantics for overloading

In this section, we look at the semantics for the overloading implementations that we have reviewed, and the implications of static and dynamic approaches to implementing ad-hoc polymorphism. The semantics of a programming language specifies the meaning of programs.

2.3.1 Dynamic Semantics

Dynamic semantics specify *how* a program is to be executed; they're concerned with what happens at run-time. This involves describing the effect of executing programs, by defining the steps of computation of the program, or as elements of some suitable mathematical structure.

For instance, System O defines a dynamic semantics for its programs, where the type of the first argument of the overloaded function is used to identify the function instance to be used at run-time. Overloaded functions are associated with functions that choose the instance to be executed at run-time, starting with the most recent definition of the overloaded variable.

2.3.2 Static Semantics

Static semantics specify the rules about the program pertaining to information that can be ascertained at compile time. This can include data typing, variable declarations, and valid function or operator names, etc. In many languages, type checking is part of the static semantics.

In Haskell, type classes are resolved as a part of the static analysis during type checking [20]. Type class instances are associated with a 4-tuple containing the data type, type class, a dictionary, and the context associated with the instance. The dictionary contains the overloaded functions, as manifested in the context of the type class instance declaration. The context is a (possibly empty) list of class constraints to be applied to the type variables defined by the instance.

In the resulting generated code, overloaded function definitions receive additional parameters to bind dictionaries, and references to the overloaded functions are passed dictionaries. This is done by adding a dictionary passing transform during the code walk performed by the type checker [20].

Functions that select appropriate methods from the dictionary are also defined during static analysis, and they simply extract a component of the dictionary tuple. This requires more plumbing in order to provide the context beforehand [24]. Implementations of type classes typically do not have a dynamic semantics, so it is not possible to express the soundness result [13].

Another example of a static semantics for ad-hoc polymorphism is System O, which describes a static semantics by defining a transformation to the Hindley/Milner system. Type annotations are optional, as a complete type inference algorithm is defined. The typing rules of the language are augmented with a function passing transform that eliminates overloaded variables by generating unique function instances for each of the overloaded instances [18].

Similarly, polymorphic expressions are transformed to accept concrete implementations of the overloaded variables as function arguments. It is further shown that the semantic soundness result [13] holds for programs written in System O.

Chapter 3

System O

System O is an extension of the Hindley/Milner system, introduced by Odersky, Wadler, and Wehr as an alternative to using type classes for supporting ad-hoc polymorphism [18]. It exhibits type soundness and principal type properties analogous to the Hindley/Milner system. In this chapter, we review the syntax, semantics, and type system of System O, and discuss the limitations posed by the system.

3.1 Abstract Syntax

The terms of the language described by System O are based on the language *Exp*, described by Milner [13]. The terms and type schemes are extended to accommodate for overloading. The syntax allows overloaded functions to be defined at the program level. This syntax is summarized in Figure 3.1.

Unique Variables	$u \in \mathcal{U}$
Overloaded Variables	$o \in \mathcal{O}$
Constructors	$k \in \mathcal{K} = \bigcup\{\mathcal{K}_D \mid D \in \mathcal{D}\}$
Variables	$x = u \mid o \mid k$
Terms	$e = x \mid \lambda u.e \mid e\ e' \mid \mathsf{let}\ u = e\ \mathsf{in}\ e'$
Programs	$p = e \mid \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$
Type Variables	$\alpha \in \mathcal{A}$
Datatype Constructors	$D \in \mathcal{D}$
Type Constructors	$T \in \mathcal{T} = \mathcal{D} \cup \{\rightarrow\}$
Types	$\tau = \alpha \mid \tau \rightarrow \tau' \mid D\ \tau_1 \ldots \tau_n$
Type Schemes	$\sigma = \tau \mid \forall \alpha.\pi_\alpha \Rightarrow \sigma$
Constraints on α	$\pi_\alpha = o_1 : \alpha \rightarrow \tau_1, \ldots, o_n : \alpha \rightarrow \tau_n$
Typotheses	$\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$

Figure 3.1: Abstract Syntax of System O

The variables, ranged over by x are divided into $\mathcal{U}$ for unique variables, $\mathcal{O}$ for overloaded variables, and $\mathcal{K}$ for data constructors. The datatypes are constructed from the datatype constructors D. The type constructors T range over all the data type constructors D, and the function type constructor $(\rightarrow)$. The types, ranged over by τ are comprised of type variables, α, functions, $\tau \rightarrow \tau'$, and data types, $D\ \tau_1 \ldots \tau_n$.

Type schemes, σ, consist of a type τ and quantifiers for the type variables in τ. A constraint on a type variable is represented as π_α, and is a set of bindings $o : \alpha \rightarrow \tau$. An overloaded variable o can appear at most once in a constraint π_α. These constraints ensure that the overloaded types are defined at the given types, and hence restrict the instance types of a type scheme.

Overloaded variables are declared with an explicit type scheme σ_T, and it is required that the type constructor T is different in each overloaded instance. This is necessary to ensure that principal types always exist. These syntactic restrictions also ensure that the argument types of the overloaded instances uniquely determine the result type, and that they work uniformly for all arguments of a given type constructor. σ_T ranges over the closed type schemes that have T as the outermost argument type constructor:

$$\begin{array}{rlr} \sigma_T = & T\ \alpha_1 \ldots \alpha_n \to \tau & (\mathrm{tv}(\tau) \in \{\alpha_1, \ldots, \alpha_n\}) \\ \mid & \forall \alpha . \pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \in \{\mathrm{tv}(\sigma'_T)\}) \end{array}$$

Instance declarations ($\mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$), are used to define overloaded instances, where the meaning of an overloaded variable o is overloaded with the expression e, where the first argument is constructed from the type constructor T. Programs in System O consist of a nested sequence of instance declarations and a term.

3.2 Semantics

The compositional semantics of System O are defined, which specify lazy evaluation for functions. However, overloaded functions are strict in their first argument [18]. The meaning of a term is a value in the CPO $\mathcal{V}$, where $\mathcal{V}$

is the least solution of Equation 3.1. The value **W** denotes a type error, and is pronounced “wrong”.

$$\mathcal{V} = \mathbf{W}_\perp + \mathcal{V} \to \mathcal{V} + \sum_{k \in \mathcal{K}} (k\ \mathcal{V}_1 \ldots \mathcal{V}_{arity(k)})_\perp \tag{3.1}$$

The meaning function, $[\![\cdot]\!]$ takes a term and an environment η, and yields an element of $\mathcal{V}$. Unique variables are mapped to arbitrary elements of $\mathcal{V}$, and overloaded variables are mapped to strict functions, as described in equation 3.2

$$\eta : \mathcal{U} \to \mathcal{V}\ \cup\ \mathcal{O} \to (\mathcal{V} \circ\!\!\to \mathcal{V}) \tag{3.2}$$

The notation $\eta[x := v]$ indicates that the variable x is bound to the value v, and the environment η is extended with this association. The dynamic semantics for System O is defined in Figure 3.2.

The meaning of a variable x yields the value stored against that variable in the environment, which is a value in the semantic domain $\mathcal{V}$. The meaning of a λ expression is evaluated by recursively evaluating the meaning of the function expression.

Similarly, the meaning of data type constructors k is evaluated by recursively finding the meaning of the arguments to the constructor, represented in the semantics with the variables $e_1 \ldots e_n$. The semantics assumes that all data type constructors are defined in a fixed initial environment.

The meaning of a let expression is evaluated by finding the meaning of the expression e', where, the variable u has meaning of the expression e.

$$
\begin{aligned}
&[\![x]\!]\eta = \eta(x)\\
&[\![\lambda u.e]\!]\eta = \lambda v.[\![e]\!]\eta[u := v]\\
&[\![k\ e_1 \ldots e_n]\!]\eta = k([\![e_1]\!]\eta)\ldots([\![e_n]\!]\eta),\\
&\qquad\qquad\qquad \text{where } n = \text{arity}(k)\\
&[\![e\ e']\!]\eta = \text{if } [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \text{ then } ([\![e]\!]\eta)([\![e']\!]\eta) \text{ else } \mathbf{W}\\
&[\![\textsf{let } u = e \textsf{ in } e']\!]\eta = [\![e']\!]\eta[u := [\![e]\!]\eta]\\
&[\![\textsf{inst } o : \sigma_T = e \textsf{ in } p]\!]\eta = \text{if } [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \text{ then}\\
&\qquad\qquad\qquad\qquad\qquad [\![p]\!]\eta[o := \text{extend}(T, [\![e]\!]\eta, \eta(o))\\
&\qquad\qquad\qquad\qquad\quad \text{else } \mathbf{W}
\end{aligned}
$$

where

$$
\begin{aligned}
\text{extend}((\to), f,\ g) &= \lambda v.\text{if } v \in \mathcal{V} \to \mathcal{V} \text{ then } f(v) \text{ else } g(v)\\
\text{extend}(D, f,\ g) &= \lambda v.\text{if } \exists k \in \mathcal{K}_D.v \in k\ \underbrace{\mathcal{V}\ldots\mathcal{V}}_{\text{arity}(k)} \text{ then } f(v) \text{ else } g(v)
\end{aligned}
$$

Figure 3.2: Semantics of System O

Function application is implemented using the expression form $e\ e'$, and its meaning is evaluated by applying the meaning of the expression e' to the meaning of the expression e. It is ensured that the expression e is a valid function of type $\mathcal{V} \to \mathcal{V}$ in the semantic domain $\mathcal{V}$.

Finally, inst expressions, that implement the overloading definitions, use the extend operation to create a nesting of expressions that select the appropriate function definition when an overloaded function is applied.

3.3 Typing Rules

The typing rules for System O are derived from those of the language *Exp* [7], with modifications to the rules ($\forall$I) and ($\forall$E). These typing rules are summarized in Figure 3.3. The substitution of type variables with types is denoted with $[\tau_1/\alpha, \ldots, \tau_n/\alpha_n]\sigma$, or $[\tau_i/\alpha_i]\sigma$, representing the type obtained by replacing the type variable α_i with τ_i in σ [7].

$$\text{(TAUT)}\ \frac{x : \sigma \in \Gamma}{\Gamma\ \vdash\ x : \sigma}$$

$$\text{(SET)}\ \frac{\Gamma\ \vdash\ x_1 : \sigma_1 \qquad \ldots \qquad \Gamma\ \vdash\ x_n : \sigma_n}{\Gamma\ \vdash\ x_1 : \sigma_1 \ldots x_n : \sigma_n}$$

$$(\to\text{I})\ \frac{\Gamma, u : \tau \vdash e : \tau'}{\Gamma\ \vdash\ \lambda u.e : \tau \to \tau'}$$

$$(\forall\text{I})\ \frac{\Gamma, \pi_\alpha\ \vdash\ e : \sigma \qquad (\alpha \notin \text{tv}(\Gamma))}{\Gamma\ \vdash\ e : \forall\alpha.\pi_\alpha \Rightarrow \sigma}$$

$$(\to\text{E})\ \frac{\Gamma\ \vdash\ e : \tau' \to \tau \qquad \Gamma\ \vdash\ e' : \tau'}{\Gamma\ \vdash\ e\ e' : \tau}$$

$$(\forall\text{E})\ \frac{\Gamma\ \vdash\ e : \forall\alpha.\pi_\alpha \Rightarrow \sigma \qquad \Gamma\ \vdash\ [\tau/\alpha]\pi_\alpha}{\Gamma\ \vdash\ e : [\tau/\alpha]\sigma}$$

$$\text{(LET)}\ \frac{\Gamma\ \vdash\ e : \sigma \qquad \Gamma, u : \sigma\ \vdash\ e' : \tau}{\Gamma\ \vdash\ \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

$$\text{(INST)}\ \frac{\begin{array}{c}(o : \sigma_{T'} \in \Gamma \Rightarrow T \neq T')\\ \Gamma\ \vdash\ e : \sigma_T \qquad \Gamma, o : \sigma_T\ \vdash\ p : \sigma'\end{array}}{\Gamma\ \vdash\ \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p : \sigma'}$$

Figure 3.3: Typing Rules for System O

The typing rules ($\forall$I) and ($\forall$E) are symmetric to the rules ($\rightarrow$ I) and ($\rightarrow$ E). In the rule ($\forall$I), the constraint π_α on the introduced bound variable α is traded between typothesis and type scheme. The rule ($\forall$E) checks for a valid substitution of the constraint π_α, by replacing the type variable α with a type τ.

The rule for instance declarations, (INST) is similar to the rule (LET), and differs in that the overloaded variable o has an explicit type scheme σ_T, and that the constructor T is different in each instance of o. This is also enforced through the syntactic restrictions.

3.4 Function-passing Transform

Odersky, Wadler, and Wehr describe a function-passing transform to translate System O terms to the Hindley/Milner system [18]. The core idea of this transform is to transform terms of type $\forall\alpha.\pi_\alpha \Rightarrow \tau$ to a function that accepts implementations of the overloaded variables as arguments. This transform is formulated as a function of type derivations. The component e^* represents the translation of a term, added to the typing judgement as $\Gamma \vdash e : \tau \succ e^*$.

Figure 3.4 defines the translation of types and type schemes.

$$\begin{aligned}\tau^* &= \tau \\ (\forall\alpha.\epsilon \Rightarrow \sigma)^* &= \forall\alpha.\sigma^* \\ (\forall\alpha.o : \alpha \rightarrow \tau, \pi_\alpha \Rightarrow \sigma)^* &= \forall\alpha.(\alpha \rightarrow \tau) \rightarrow (\forall\pi_\alpha \Rightarrow \sigma)^*\end{aligned}$$

Figure 3.4: Translation of types and type schemes

Overloaded variables o are transformed into a new unique variable u_{o,σ_T}, whose identity depends on the name o and the type scheme σ_T. This translation is defined in Figure 3.5.

$$\begin{aligned}(u : \sigma)^* &= u : \sigma^* \\ (o : \sigma)^* &= u_{o,\sigma} : \sigma^* \\ o_1 : \sigma_1, \ldots, o_n : \sigma_n &= (o_1 : \sigma_1)^*, \ldots, (o_n : \sigma_n)^*\end{aligned}$$

Figure 3.5: Translation of bindings and typotheses

The rule ($\forall$I) transforms an expression to a λ expression that expects as arguments the implementations of the overloaded functions. The corresponding rule ($\forall$E) applies the function with the appropriate overloaded instance. To ensure coherence of the transformation, it is assumed that the overloaded identifiers o are ordered lexicographically in a type variable constraint $\{o_1 : \alpha \rightarrow \tau_1, \ldots, o_n : \alpha \rightarrow \tau_n\}$. The transform is summarized in Figure 3.6.

$$\text{(TAUT)}\ \frac{u : \sigma \in \Gamma}{\begin{array}{c}\Gamma \ \vdash\ u : \sigma \\ \succ u\end{array}} \qquad \frac{k : \sigma \in \Gamma}{\begin{array}{c}\Gamma \ \vdash\ k : \sigma \\ \succ u\end{array}} \qquad \frac{o : \sigma \in \Gamma}{\begin{array}{c}\Gamma \ \vdash\ o : \sigma \\ \succ u_{o,\sigma}\end{array}}$$

$$(\to\text{I})\ \frac{\Gamma, u : \tau \vdash e : \tau' \succ e^*}{\begin{array}{c}\Gamma \ \vdash\ \lambda u.e : \tau \to \tau' \\ \succ \lambda u.e^*\end{array}}$$

$$(\forall\text{I})\ \frac{\Gamma, o_1 : \tau_1, \ldots, o_n : \tau_n \vdash e : \sigma \succ e^* \qquad \alpha \notin \text{tv}(\Gamma)}{\begin{array}{c}\Gamma \ \vdash\ e : \forall\alpha.(o_1 : \tau_1, \ldots, o_n : \tau_n) \Rightarrow \sigma \\ \succ \lambda u_{o_1,\tau_1} \ldots u_{o_n,\tau_n}.e^*\end{array}}$$

$$(\to\text{E})\ \frac{\Gamma \ \vdash\ e_1 : \tau' \to \tau \ \succ\ e_1^* \qquad \Gamma \ \vdash\ e_2 : \tau' \ \succ\ e_2^*}{\begin{array}{c}\Gamma \ \vdash\ e_1\ e_2 : \tau \\ \succ e_1^*\ e_2^*\end{array}}$$

$$(\forall\text{E})\ \frac{\begin{array}{c}\Gamma \ \vdash\ e : \forall\alpha.(o_1 : \tau_1, \ldots, o_n : \tau_n) \Rightarrow \sigma \succ e^* \\ \Gamma \ \vdash\ o_i : [\tau/\alpha]\tau_i \succ e_i^* \qquad (i = 1, \ldots, n)\end{array}}{\begin{array}{c}\Gamma \ \vdash\ e : [\tau/\alpha]\sigma \\ \succ e^*\ e_1^* \ldots e_n^*\end{array}}$$

$$(\text{LET})\ \frac{\Gamma \ \vdash\ e : \sigma \succ e^* \qquad \Gamma, u : \sigma \ \vdash\ e' : \tau \succ e'^*}{\begin{array}{c}\Gamma \ \vdash\ \textsf{let}\ u = e\ \textsf{in}\ e' : \tau \\ \succ \textsf{let}\ u = e^*\ \textsf{in}\ e'^*\end{array}}$$

$$(\text{INST})\ \frac{\begin{array}{c}(o : \sigma_{T'} \in \Gamma \Rightarrow T \neq T') \\ \Gamma \ \vdash\ e : \sigma_T \ \succ\ e^* \quad \Gamma, o : \sigma_T \ \vdash\ p : \sigma' \ \succ\ p^*\end{array}}{\begin{array}{c}\Gamma \ \vdash\ \textsf{inst}\ o : \sigma_T = e\ \textsf{in}\ p : \sigma' \\ \succ \textsf{let}\ u_{o,\sigma_T} = e^*\ \textsf{in}\ p^*\end{array}}$$

Figure 3.6: Function passing transform

3.5 Programs in System O

Consider the Haskell program in Listing 3.1. The program defines overloading behavior using type classes, then defines a polymorphic function using that type class.

We first define the type class Pointed, with the functions xcoord and ycoord to extract the x and y coordinates of an instance of Pointed. The types Point, with the two fields of type Float representing the x and y coordinates of a point, and CPoint, for colored points, containing the x, y coordinates and a color of type String. Using pattern matching on their respective constructors, the types Point and CPoint implement the functions defined by the type class Pointed, creating instances of the type class.

Finally, the polymorphic function dist is defined, that operates on instances of Pointed to calculate the distance of points from the origin.

```
class Pointed a where
  xcoord :: a -> Float
  ycoord :: a -> Float

data Point = MkPoint Float Float
data CPoint = MkCPoint Float Float String

instance Pointed Point where
  xcoord (MkPoint x y) = x
  ycoord (MkPoint x y) = y

instance Pointed CPoint where
  xcoord (MkCPoint x y c) = x
  ycoord (MkCPoint x y c) = y

dist :: (Pointed a) => a -> Float
dist p = sqrt (sqr (xcoord p) + sqr (ycoord p))
```

Listing 3.1: A program implementing overloading in Haskell

In Listing 3.2, we rewrite the Haskell example in System O. We observe the impact of eliminating type class declarations in the definitions of the functions xcoord and ycoord. These functions are now defined using the keyword inst, indicating that they are overloaded for the types Point and CPoint. The function definitions also differ in the type of the first parameter to the overloaded function definitions, as required by System O. For brevity, we have written the functions using pattern matching, as in the Haskell program, though System O doesn't explicitly specify pattern matching in its formal syntax.

Furthermore, the type annotation on the function dist is now more verbose, as there is no mechanism like type classes to group together the related functions. This type annotation explicitly lists the names and types of the overloaded functions it uses. These type annotations can become lengthy, potentially affecting the readability of the code.

System O specifies a complete type inference algorithm, eliminating the need for type annotations. However, writing type annotations improves the readability and maintainability of programs. In certain cases, they are required to resolve ambiguity in programs. Further, type annotations can reduce the burden on a type reconstruction algorithm. Odersky and Läufer present extensions to the Hindley/Milner system with this goal [17]. We discuss the implications of verbose type annotations in the following section, as this is a problem we aim to address in this study.

```
data Point  = MkPoint Float Float
data CPoint = MkCPoint Float Float String

inst xcoord :: Point -> Float
xcoord (MkPoint x y) = x

inst ycoord :: Point -> Float
ycoord (MkPoint x y) = y

inst xcoord :: CPoint -> Float
xcoord (MkCPoint x y c) = x

inst ycoord :: CPoint -> Float
ycoord (MkCPoint x y c) = y

dist :: (xcoord, ycoord :: a -> Float) => a -> Float
dist p = sqrt (sqr (xcoord p) + sqr (ycoord p))
```

Listing 3.2: A program implementing overloading in System O

3.5.1 Limitations and Proposed Extensions

From the example in Listing 3.2, we observe the verbosity in type annotations for polymorphic functions in System O. The function names and types have to be explicitly stated in the type annotation. For brevity, we grouped together the functions xcoord and ycoord as they share the same signature, but a type annotation in this format can become increasingly tedious to write for more complex functions.

In contrast, in Listing 3.1, we observe the merits of expressing types using type classes in this context. The type annotations are concise, as type classes provide a mechanism to group together related overloaded names. It follows that the type annotations are more readable and make the purpose of the functions clearer.

One possible approach to overcoming this issue of conciseness is to extend System O to add the ability to provide constraint aliases to groups of functions for the purpose of using them in type annotations. These aliases can then be used in the type annotations. In our example from Listing 3.2, it would require providing an alias to the group of functions xcoord and ycoord.

In this study, we present extensions to System O to add the ability to define and instantiate type classes, closely following their implementation in Haskell. This is an elegant way to solve the problem discussed above, with a mechanism to write more expressive and readable code.

Another limitation of System O is that the type of the first argument of overloaded functions should be different. As our implementation of type classes is a direct extension of the System O dynamic semantics, this limitation is inherently carried over to it.

Consider the type class Parser, that defines the function parse, with the type signature $\mathsf{String} \rightarrow \mathsf{a}$. The type of the type variable a is defined by instances of the type class. If we were to dispatch on only the type of the first argument, implementing this type class would not be possible. As a part of implementing an extended System O, we aim to address this restriction.

Another related limitation arises when trying to implement type classes with functions where dispatch depends on the return type of a function, like Functor or Reader. Consider the type class Functor, which defines the function fmap. The type signature of this function is $(\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$, . Given the dynamic semantics that we are implementing, addressing this scenario is out of the scope of this study. This limits the type classes that we can define using our construct, which impacts the flexibility of the programming language. However, retaining the System O dynamic semantics has a value in making the programming language easy to understand, and hence reducing the complexity of the interpreter implementation. We present workarounds that can help alleviate the problem.

In the rest of this study, we present extensions and modifications to System O to make the programming language more practical. Then we define type classes on top of the extended System O, and introduce optimizations to the function-passing transform that aim to improve the performance and readability of code.

Chapter 4

Implementing an Extended System O

In this chapter, we present extensions to System O that aim to make the programming language more practical and overcome some of the limitations described in Chapter 3. With these extensions in place, we describe the construction of an interpreter to run programs using this extended programming language.

We develop a "metacircular evaluator", inspired by the interpreters of Abelson and Sussman [1] to implement the dynamic semantics. The interpreter is designed to load and run programs written in System O, or run interactively as a read–eval–print loop (REPL) [9]. The implementation language chosen for the interpreter is Clojure [21].

We proceed with the development in multiple steps, starting with a fully functioning interpreter based on the dynamic semantics with our extensions. This system is capable of handling ad-hoc overloading and resolves the overloaded instances at run-time.

We then implement the type checker for this interpreter, implementing the rules defined for System O in Chapter 3, with modifications to accommodate for the syntactic extensions.

Finally, we implement the function-passing transform, adding the ability to resolve overloaded instances during the type checking process, hence reducing the run-time overhead.

4.1 The Interpreter Environment

The environment is a core data structure used throughout the interpreter. It provides a store for variable names and their values. It determines the context in which an expression should be evaluated. The implementation is based on the "Environment Model of Evaluation" by Abelson and Sussman [1].

The environment is implemented as a sequence of *frames*, which are (possibly empty) tables of bindings that associate variable names with their values. A frame is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values. A frame also contains a pointer to its enclosing environment.

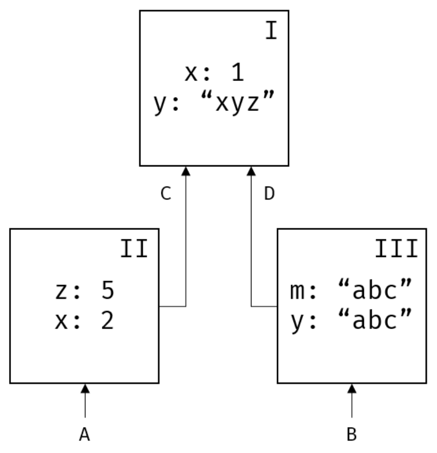

Figure 4.1: Structure of a simple environment

Figure 4.1 shows the structure of a simple environment, with the frames I, II, and III. The pointers to these frames are labeled A, B, C, and D. The process of looking up a variable in the environment begins with looking at the first (outermost) frame, II or III, in this figure, and successively looking at the enclosing environments if the variable is not found.

For example, looking for the variable `x` starting with frame I will return the value 1, while looking for the same variable starting with frame II will return the value 2.

The last (innermost) frame has no enclosed frames, and is defined as the *global environment*. In our implementation, we use this global environment to define the constant data types and the operations on them, which are defined with the corresponding operations in the implementation language.

The following operations on the environment allow us to store and access variables. All functions take as argument the environment that the program being executed has in its current context, denoted as $\langle\mathsf{env}\rangle$. The function parameter $\langle\mathsf{var}\rangle$ denotes the variable to look for, or add to, the environment. The parameter $\langle\mathsf{val}\rangle$ denotes the value to be set against a variable in the environment.

- lookup $\langle\mathsf{var}\rangle\langle\mathsf{env}\rangle$: Finds and returns the value associated with the variable $\langle\mathsf{var}\rangle$ in the environment $\langle\mathsf{env}\rangle$, throws an error if the variable is unbound.
- extend $\langle\mathsf{var}\rangle\langle\mathsf{val}\rangle\langle\mathsf{env}\rangle$: Returns an environment with a new frame which points to the current environment $\langle\mathsf{env}\rangle$. The new frame binds the variable $\langle\mathsf{var}\rangle$ to the value $\langle\mathsf{val}\rangle$. This operation is often used to create a new context when evaluating nested expressions, where variables are assigned new values in the context of an inner expression.
- define! $\langle\mathsf{var}\rangle\langle\mathsf{val}\rangle\langle\mathsf{env}\rangle$: Binds the variable $\langle\mathsf{var}\rangle$ to the value $\langle\mathsf{val}\rangle$ in the first frame of the environment $\langle\mathsf{env}\rangle$. Overrides the value for a variable if it has already been defined. This is essentially a side effect that allows us to implement declaration of top level variables.

4.2 Syntactic Extensions

We define the syntax for our language by extending the syntax of System O. These extensions are added to provide common programming constructs to the language, making it more palatable.

We begin with adding primitive types, ranging over the types Integer, Double, Char, String, and Boolean. The constant values are the set of all values ranging over the primitive types. These types correspond to primitive types in the implementation language, Clojure.

We use the datatype constructors to model "record" structures found in common programming languages, with constructors k, that accept zero or more arguments. We add this declaration with the keyword record. A record is defined as (record k $[u_1 \ldots u_n]$), where k is the name of the record, and $u_1 \ldots u_n$ are the names of the fields.

In addition to datatype constructors, we add the ability to define algebraic data types, with the keyword data. These are defined as a set of one or more constructors associated with a datatype. Algebraic data types would allow programmers to define additional data structures, such as Maybe or List. Algebraic data types are defined with the syntax (data k_a $(k_1\ [u_{1_1} \ldots u_{1_m}])$ $\ldots (k_n\ [u_{n_1} \ldots u_{n_m}])$), where k_a is the name of the algebraic data type, $k_1 \ldots k_n$ are its constructors with fields $u_{1_1} \ldots u_{n_m}$.

All datatype constructors are associated with a predicate function and functions for selecting fields from an instance of the datatype. These functions accept one argument, the datatype instance. For a constructor k, a predicate takes the form k?. The record selectors are defined as k-u$_1$ $\ldots$ k-u$_n$ where k is the constructor with fields $u_1 \ldots u_n$. These functions are generated by the interpreter when a data type constructor is defined.

We also extend the terms of the language e to contain constants. We modify the λ expressions to accept multiple arguments, to avoid the overhead of desugaring, and to keep the interpreter simple.

Consequently, function application is also modified to allow multiple arguments to functions. if expressions and cond for conditionals are also added to the terms.

Another addition to the syntax is the ability to associate names with terms, using the define construct. Functions declared using this construct are allowed to be recursive, referring to the name that they are defined with. This is an implementation detail defined in the interpreter.

We make explicit the declaration of overloaded variables, o, which are declared in a program using the over construct. Instance declarations (inst o τ e) are used to overload the definition of o for type τ with the expression e. The interpreter ensures that the expression e is a λ expression. Consequently, programs are a sequence of expression, overload, and instance definitions.

4.3 Semantics

Implementing the semantics involves defining a procedure corresponding to the meaning function $[\![\cdot]\!]$, defined in Chapter 3. It accepts a term and the environment η. In this section, we describe the implementation of the meaning function, accounting for the extensions in syntax.

The terms are represented as S-expressions and resemble expressions in Clojure. The environment initially supplied to this function is the global environment defined previously.

The interpreter is implemented by performing a case analysis on the form of the expression. This section describes these cases implemented on the dynamic semantics of System O, described in Figure 3.2, accounting for the modifications to the syntax. Expressions in the semantics that evaluate to **W** throw an error at run-time that stops the execution of the program. For constants of a valid type, their value is returned as-is.

Variable values are retrieved from the environment in the current context using the lookup function on the environment, as (lookup x η), where x is the variable and η is the environment.

λ expressions are transformed into a closure by packaging together the parameters and body of the expression with the environment of the evaluation. We define the structure of this closure as a triple that contains the function body, the list of arguments that it accepts, and the environment in which the λ expression was created. In implementing the expression defined above, we also account for λ expressions accepting multiple arguments.

Instances of data type constructors are created by storing the values obtained by evaluating the parameters $e_1 \ldots e_n$ with the constructor in a tagged structure. Consequently, the predicate and accessor functions defined for the constructors use these tagged structures to type check data type instances or extract specific parameters. These predicate and accessor functions ensure that they operate on values of the type that they are created for, by checking the tagged structure.

if expressions are evaluated using the underlying implementations in Clojure. By extension, cond expressions are essentially treated as nested if expressions and are evaluated recursively till a predicate is satisfied, or there are no more predicates to check.

Function application of the form $(e\ e_1 \ldots e_n)$ gathers the closure corresponding to the value of the expression e, evaluates the expressions $e_1 \ldots e_n$ to get the values associated with the arguments, then applies the closure to these values. By nature of this implementation, recursive calls are handled as the values are supplied to the interpreter with the extend operation on the environment. The implementation of the above expression in the interpreter accounts for functions accepting multiple arguments.

When applying overloaded functions, the interpreter attempts to find a suitable overload for the type of arguments specified, and throws an exception if none is found. As the interpreter operates on multiple arguments, the process of finding an overloaded instance verifies the entire type signature of the functions.

let expressions evaluate the meaning of the expression e, and associate it with the variable u using the extend operation on the environment, then evaluate the expression e' with this new context. By the nature of the implementation of the environment and the meaning function, this context is discarded once the expression is evaluated.

The define construct evaluates the value of the expression e and stores it against the variable u in the current environment using the define! operation. In a similar vein, the over construct defines a default value against the name of the overloaded expression that throws an error.

Programs are evaluated in the order that they are defined, and declared expressions can refer to variables declared before them.

Overloaded expressions defined using the inst construct are restricted to be functions, and are added to the environment against the overloaded variable o using the define! operation on the environment. When applied, these definitions are evaluated in order from the most recent definition to the oldest to find an applicable overload, and result in an error (expression evaluates to $\mathbf{W}$) if no matching overload can be found for a function call. In defining the overloaded instances, we check that an instance with the same signature has not been previously defined, using all the types defined for the overload.

4.4 Typing Rules

The type checker implements the typing rules defined by System O in Figure 3.3, accounting for the extensions in syntax that we have added. Types are specified in a program with the construct shown in Figure 4.2. Here, u is the name of an expression declared using the define construct, and τ is its type. For functions, the types are declared in the form $[\tau_1 \ldots \tau_n]$ to keep the syntax concise.

$$(\textsf{type}\ u\ \tau)$$

Figure 4.2: Syntax for Type Declarations

Before a program is type checked, it undergoes preprocessing to split the code into expressions declared using the define construct, type classes, and data types. The declared expressions undergo type checking, and the type classes and data types are used in the process of checking the type. The implementation of type checking with type classes is discussed in Chapter 5. This process also ensures that type annotations are specified for all expressions declared at the top level.

Similar to the implementation of the semantics, the type of expressions is checked by performing a case analysis on the form of the expression. The typing judgement function takes a typothesis and an expression, and uses the typing environment to recursively determine the type of the expression. The typing environment Γ reuses the environment structure we defined previously in Section 4.1.

Starting with the topmost expression, that the type annotation is specified for, the function recurses down to the innermost expressions. As the recursion unwinds, the type of each subsequent expression is checked, finally checking the type of the topmost expression. If the type check is successful, the type of the expression is returned. For the type checker, this return value is not used, but it is important that the function returns, indicating that the type checking was successful. However, these returned values are required for the function-passing transform.

In the following section, we describe the implementation for the System O typing rules case-by-case, with emphasis on the details accounting for the extensions we have added.

Values of constant types are defined as default implementations in the type checker, and use standard Clojure functions to identify the type of values of constant type.

The type of variables is checked by looking up their types in the typing environment, using the lookup function defined on the environment. This corresponds to the rule (TAUT) defined by System O.

Similarly, for let expressions of the form, let $u = e$ in e', we check for the type of the expression e' in the context of an extended typing environment, where the variable u has the type of the expression e. The type of the expression e is inferred. It is assigned a unification variable, which is used to extend the environment with the inferred type, and the type of the overall let expression is checked under the inferred type.

For conditionals, it is verified that the predicates of the expressions are of type Boolean, and the resultant expressions all yield the same declared type.

To check the type of λ expressions, we extend the typing environment with the function arguments and their types using the `extend` operation on the environment, and recursively check that the expression of the body of the function conforms to the declared type. This corresponds to the rule ($\rightarrow$I) defined by System O, accounting for functions accepting multiple arguments.

Type checking λ expressions also accounts for expressions that use overloaded functions in their definitions. The function definitions collected during preprocessing are used to identify when an expression uses an overloaded function. It is assumed to be of the general type, $\forall\alpha.\pi_\alpha \Rightarrow \sigma$, which would be the type defined in the type class definition.

Note that in the process of developing this interpreter, we implemented type classes before adding type checking, and the general type of the expression is taken from the type of the function as defined in the type class. Specifying a constraint with a type class for a type variable in the annotation restricts the type variable α to instances of the type class. To check that the expression conforms to the type annotation, it is checked that the type of the arguments matches one unique overloaded definition of the function. This involves checking the type of the arguments against every overloaded definition for the function in the typing environment, which is similar to the computation of the constraint $\pi_\alpha = o_1 : \alpha \rightarrow \tau_1, \ldots, o_n : \alpha \rightarrow \tau_n$. This ensures that the type specified in the annotation is no more general than the type of the type class itself. This is an extension of the rule ($\forall$I).

Type checking function application is more intricate, as there are several types of functions, including those generated internally by the interpreter, like, the predicate and accessor functions. Additionally, this process needs to handle type checking on overloaded functions, which involves checking that the function application conforms to one unique overloaded instance. In general, there are three cases for type checking function application:

- The first case applies to primitive functions, accessors, predicates, and declared functions that are not overloaded. In this case, we identify the type of the function, and check that each of the arguments correspond to the types expected by the function, in order. Then, we can say that this function application is of the type denoted by the annotation. This corresponds to the rule ($\rightarrow$E).
- The second case pertains to application of overloaded functions. Type checking in this case is similar to the first case, where a variable name is associated with multiple function definitions. As a part of the preprocessing, overloaded function definitions are added to the typing environment against the overloaded variable name, and evaluated in order from the most recent definition to the oldest. This implementation also handles the ambiguity arising from overlapping instances, discussed in Chapter 2 with λ^O, by ensuring that no two overloaded functions resolve to the same signature with the concrete type of the arguments provided.
- The third case applies to functions that use overloaded functions. The constraint specified in the type annotation is used to type check the

arguments in order to ensure that they conform to the type of one unique overloaded instance. This corresponds to the typing rule (∀E).

Finally, type checking expressions declared using the define construct involves checking the type of the expression that is being assigned to the variable.

4.5 Function-passing transform

The type checker is augmented with the function-passing transform that translates overloaded functions and expressions that use overloaded functions to be resolved during the type checking. The implementation follows the rules defined in Figure 3.6, accounting for the changes in syntax and the typing rules.

The function passing transform is also implemented by performing a case analysis on the form of the input expression, albeit with fewer cases undergoing any major transformation.

The cases of interest are expressions that use overloaded functions in their definitions, or application of overloaded functions, which correspond to the rules (∀I) and (∀E).

Expressions that use overloaded functions are transformed into λ expressions, that accept implementations of the overloaded arguments. Listing 4.1 shows an excerpt of a program before it is transformed. The polymorphic function double uses the overloaded function add, and can operate on values of type Integer or Double.

```
(overload add)

(inst add [Integer Integer Integer] (lambda [x y] (*prim+i x y)))
(inst add [Double Double Double] (lambda [x y] (*prim+d x y)))

(define double (lambda [x] (add x x)))

(double 2.0)
```

Listing 4.1: Program before transformation

Before the transformation being implemented, calling the function double with some arguments would require the interpreter to look at the type of the arguments, verify if the arguments conform to the type defined by any of the overloads, then call the function with the arguments. At run-time, this can cause a significant overhead during code execution. The transform aims to reduce this overhead by deciding the overloaded function to call, if any, before the code execution begins. Listing 4.2 shows an excerpt of the function double after the transformation.

```
(define double (lambda [add:a] (lambda [x] (add:a x x))))

((double (lambda [x y] (*prim+d x y))) 2.0)
```

Listing 4.2: Code generated by the transformation

As a result of the transformation, the call to the function double has been resolved during the type checking process, and the interpreter immediately knows what function definition to use with the argument 2.0, reducing the overhead of making this decision at run-time.

Note that the above listings demonstrate the effect of transformation without taking type classes into account, resulting in slightly different transformed code, as the name-mangled functions use type class instance names in the final implementation.

Since the type checker, and by extension, the transformation work recursively bottom-up, we notice that the transformation results in code littered with these λ expressions. We devise a method to clean these up as a part of optimizations in Chapter 6.

As a result of the above transformation, function application for overloaded functions results in supplying the definitions for the overloaded functions where a concrete type for the arguments to the overloaded functions can be determined, and an overload exists with those types.

4.6 Examples

Using the interpreter developed in this chapter, we have a platform to write and execute programs using the dynamic semantics. In this section, we look at several examples that utilize the overloading semantics and features that we have developed so far.

In Listing 4.3, we rewrite the Point example from Listing 3.2 using the extended System O. We define two datatypes, Point, with the fields x and y, and CPoint for colored point, with the fields c, x, and y. We then define an overloaded function second that retrieves the y value for any type of point. We use the instance record accessors to define the underlying functionality.

The function dist takes in a point and returns its distance from the origin. As noted in the previous sections, the interpreter was developed to support type classes before type checking, and hence, the type constraints on the function dist are a representation of what they would look like in an extended System O, accounting for the changes in syntax we have added.

```
(record Point [x y])
(record CPoint [c x y])

(overload first)
(inst first [Point] (lambda [p] (Point-x p)))
(inst first [CPoint] (lambda [cp] (CPoint-x cp)))

(overload second)
(inst second [Point] (lambda [p] (Point-y p)))
(inst second [CPoint] (lambda [cp] (CPoint-y cp)))

(type dist [a Double] [(first :: a -> Double, second :: a ->
   Double)])
(define dist (lambda [p] (sqrt (sqr (first p) + sqr (second p)))))
```

Listing 4.3: An example of overloading with record data types

Listing 4.4 demonstrates a program that overloads the function add to work on types Integer, Double, and String. The listing declares the overloaded variable with the construct overload add. The type of the overloaded functions is given in the instance declaration using the syntax $[\tau_1 \ldots \tau_n]$, as a shorthand for the type declaration $\tau_1 \to \cdots \to \tau_n$.

```
(overload add)

(inst add [Integer Integer Integer] (lambda [x y] (*prim+i x y)))
(inst add [String String String] (lambda [x y] (*prim+str x y)))
```

```
(inst add [Double Double Double] (lambda [x y] (*prim+d x y)))

(add 1 1) ;; outputs 2
(add "Hello, " "World!") ;; outputs "Hello, World!"
(add 3.0 2.0) ;; outputs 5.0
```

Listing 4.4: An example of overloading using the dynamic semantics

Listing 4.5 demonstrates the extensions allowing definition of algebraic data types. It defines the type Maybe, with the constructors Just, and Nothing.

The constructor Just for the type Maybe accepts an argument of any type, declared with the type variable a in the declaration of the algebraic data type (Maybe a) and the constructor (Just [x] [a]). We notice the type checking for the concrete type of this type variable for the declared variable j.

We define a simple function, hasValue that takes an instance of type Maybe, and returns a boolean indicating the type of the instance. It uses instance predicates to check for the specific type constructor.

```
(data (Maybe a) (Just [x] [a])
                (Nothing))

(type hasValue [(Maybe a) boolean])
(define hasValue (lambda [v] (cond
                               (Just? v)    true
                               (Nothing? v) false)))

(type j (Maybe integer))
(define j (Just 1))

(type n (Maybe a))
(define n (Nothing))

(hasValue (Just 1)) ;; outputs true
(hasValue (Nothing)) ;; outputs false
```

Listing 4.5: Implementing the type Maybe

Listing 4.6 defines the operation append on the type List, as a means to concatenate two lists together. We define a structurally recursive data type List using an approach similar to Listing 4.5, with the constructors Cons and Empty. Cons holds an element of the type of the list as the head, a recursive instance of List as its tail.

The function append accepts a list of any type, and an accumulator to build the result. The value of the accumulator is Empty in the examples below. The function first appends the values from the list l1 to the accumulator, and once it is empty, it appends the values from the list l2.

By the nature of the implementation, this result in the accumulator has the last value from the list l2 as its "head" – the result is in reverse. The helper function rev is defined to reverse this result, and return the concatenated list.

```
(data (List a) (Cons [h t] [a (List a)])
               (Empty))

(type rev [(List a) (List a)])
(define rev
 (lambda [l a]
  (cond (Empty? l) a
        (Cons? l) (rev (Cons (Cons-h l) a)))))

(type append [(List a) (List a) (List a)])
(define append
 (lambda [l1 l2 acc]
  (cond
   (*prim-and (Empty? l1) (Empty? l2)) (rev acc (Empty))
   (Cons? l1) (append (Cons-t l1) l2 (Cons (Cons-h l1) a))
   (Cons? l2) (append l1 (Cons-t l2) (Cons (Cons-h l2) a)))))

(append (Cons 1 (Cons 2 (Empty))) (Cons 3 (Cons 4 (Empty))) (Empty
   ))
;; outputs (Cons 1 (Cons 2 (Cons 3 (Cons 4 (Empty)))))
```

```
(append (Cons 1.0 (Cons 2.0 (Empty))) (Cons 3.0 (Cons 4.0 (Empty))
   ) (Empty))
;; outputs (Cons 1.0 (Cons 2.0 (Cons 3.0 (Cons 4.0 (Empty)))))
```
Listing 4.6: An example of using the overloaded functions with the type List

Chapter 5

A Semantics for Type Classes

Type classes provide a framework for implementing ad-hoc polymorphism by encapsulating overloading behavior in their definition [26]. Instances of the type classes provide implementation for overloading for any declared type. With the semantics for overloading defined in Chapter 4, we proceed to define type classes on top of the dynamic semantics. We utilize the underlying mechanism for defining instances of overloaded functions at the program level, using the inst declarations.

5.1 Syntax

We extend the abstract syntax from Chapter 4 to include definition and instantiation of type classes. Implementing type classes consists of two parts [26]:

- Declaring the type class. For instance, to overload the methods (+), (*), and negate, we declare the type class Num, that declares the name and signatures of the functions to be overloaded. It can be understood as, "a type belongs to the type class Num if it has the functions (+), (*), and negate of the appropriate types defined."

- Instantiating the type class. The instantiation Num Int can be understood as "declaring these are the definitions of (+), (*), and negate, for the type Int, the function definitions justify the assertion with appropriate bindings for each function, as required by Num."

Type Class $T_c = \textsf{typeclass}\ u\ \alpha\ (o_1\ [\tau_{1_1} \ldots \tau_{1_m}\ \tau_1']) \ldots (o_n\ [\tau_{n_1} \ldots \tau_{n_m}\ \tau_n'])$
Type Class Instance $= \textsf{typeclass-inst}\ u\ \tau\ (o_1[x_{1_1} \ldots x_{1_m}]\ e_1) \ldots (o_n[x_{n_1} \ldots x_{n_m}]\ e_n)$

Figure 5.1: Syntax for declaring and instantiating Type Classes

The syntax for type classes follows the two parts described above. The syntax is summarized in Figure 5.1. A type class declaration T_c defines a type class with the name u. The type variable α is replaced in function definitions by the type specified in the instances of the type class. The type class declares function signatures that the instances must implement with type signatures $[\tau_1 \ldots \tau_n\ \tau']$, where n is the arity of the functions and τ' is the co-domain. The types τ may themselves by type variables α. These declarations have the name of the overloaded function o, from the set of overloaded variables $\mathcal{O}$.

Type class instances are defined with the typeclass-inst construct. The datatype D replaces the type variable α in the function type signatures, and the function definitions justify the types. The instance declaration provides implementations for the functions declared by the type class. The syntax presented is a shorthand for defining a λ expression, where $[x_1 \ldots x_m]$ are the arguments to the function of arity m, and e is the body of the function.

5.2 Implementation

The implementation of type classes utilizes the constructs from the dynamic semantics implemented in Chapter 4. In order to translate these terms, the interpreter implements the following steps:

- A type class declaration creates "default" functions for each function defined in the type class, that would result in an error. This is a translation to the over construct defined in the dynamic semantics. The type definition of these functions with the type variables is preserved, to create function signatures for the overloaded instances.

- Members of a type class instance are translated to an inst declaration. The two operations specified below help implement this translation:

  - Functions on type class instances are defined with a shorthand, eliminating the need to explicitly declare a λ expression. This expression is desugared to the λ expression syntax expected by the semantics.

  - In order to specify the full type of the overloaded instance, the function signature declared as a part of the type class declaration is revised. The type τ of the type class instance replaces the type variable α defined in the type class. This revised function signature is used to create an instance declaration of the form:

    inst o $[\tau_1 \ldots \tau_n\ \tau']$ (lambda $[x_1 \ldots x_n]$ e).

5.3 Type Checking

Type classes are used to add constraints to type signatures. To implement this, we extend the syntax for defining types for declarations by adding an extra optional parameter for specifying type classes, described below.

$$(\mathit{type}\ u\ \tau\ [(u_1\ \alpha_1)\ldots(u_n\ \alpha_n)])$$

The type constraints are specified for the type variables in the type declaration, with type class names and respective type variables. Multiple type class constraints can be specified on the same type variable, further restricting the type of the declaration.

Listing 5.1 shows an example of defining type annotations for the function div using the type classes Num and Eq. The constraint specifies that the type variable a is restricted to only those types that instantiate both Num and Eq.

```
(type div [a a a] [(Num a) (Eq a)])
(define div ...)
```

Listing 5.1: Type annotations with type classes

Specifying type constraints using type classes also provides a much more concise way to represent the constraints, as we can avoid having to specify explicitly the type of each individual overloaded function in use, and instead group them together with a potentially more understandable name.

The type checker is modified for handling type constraints on declared types, with the constraints being passed down with each recursive call for checking the expressions. When a type class constraint is specified in the type annotations, the type checker ensures that the type of the expression is no more general than the type classes, and that it conforms to all type classes specified in the constraint. These constraints are discussed in detail with examples in the following section.

5.4 Examples

With the ability to define type classes in place, the language now allows more expressive type definitions, where related types can be grouped together with type classes. The type constraints introduced in the type checking allow for defining concise constraints on functions using the type classes.

We now proceed to create a Prelude environment for the language, and define the type classes Num, Eq, and Ord, and create instances for these type classes for the primitive types.

Consider the type class Num in Listing 5.2, defined with the type variable a. The functions (+), (-), (*) have the type signature [a a a], indicating that the functions accept two arguments of a type that instantiates Num, and return a value of the same type. Similarly, the function neg accepts one argument of an instance of Num, and returns a value of the same type.

A type class instance, such as the one for Integer provides definitions for these function declarations, with implementations that adhere to the form defined by the type signatures in the type class. For instance, the function (+) for the type Integer accepts two arguments, of type Integer, and uses the primitive add operation to return an Integer value. The operations on the primitive types are defined as a part of the interpreter implementation Chapter 4. This example is shown in the program fragment in Listing 5.2.

```
(typeclass Num a
        (+ [a a a])
        (- [a a a])
        (* [a a a])
        (neg [a a]))
(typeclass Eq a
        (== [a a Boolean])
        (!= [a a Boolean]))
(typeclass Ord a
        (< [a a Boolean])
        (> [a a Boolean])
        (<= [a a Boolean])
        (>= [a a Boolean]))

(typeclass-inst Num Integer
        (+ [x y] (*prim+i x y))
        (- [x y] (*prim-i x y))
        (* [x y] (*prim*i x y))
        (neg [x] (*prim*i -1 x)))
(typeclass-inst Eq Integer
        (== [x y] (*prim=i x y))
        (!= [x y] (*prim!bool (*prim=i x y))))
(typeclass-inst Ord Integer
        (< [x y] (*prim<i x y))
        (> [x y] (*prim>i x y))
        (<= [x y] (*prim<=i x y))
        (>= [x y] (*prim>=i x y)))

(typeclass-inst Num Double
        (+ [x y] (*prim+d x y))
        (- [x y] (*prim-d x y))
        (* [x y] (*prim*d x y))
        (neg [x] (*prim*d -1.0 x)))
```

```
...
```

Listing 5.2: An example of type classes defining the Prelude environment

Listing 5.3 defines a sorting program on the List data type we defined in Listing 4.6. Unlike the sum function, insertionSort can operate only on lists that contain certain types of elements, namely, those that implement functions that allow comparison between elements of the type. This is reflected in the type annotation of the helper function insert, and the function insertionSort.

To implement this, we constrain the type variable a defined in the function to instances of the type class Ord, defined in Listing 5.2. The function insert uses the operation <= defined in the type class Ord to compare the elements. The calls to the function then resolve the <= function based on the type of the arguments passed, as a part of the function-passing transform.

```
(data (List a) (Cons [h t] [a (List a)])
               (Empty))

(type insert [a (List a) (List a)] [(Ord a)])
(define insert (lambda [x L]
                 (if (Empty? L)
                  (Cons x (Empty))
                  (if (<= x (Cons-h L))
                     (Cons x L)
                     (Cons (Cons-h L) (insert x (Cons-t L)))))))

(type insertionSort [(List a) (List a)] [(Ord a)])
(define insertionSort (lambda [xs]
                        (if (Empty? xs)
                         (Empty)
                         (insert (Cons-h xs)
                                 (insertionSort (Cons-t xs))))))

(insertionSort (Cons 3 (Cons 2 (Cons 1 (Empty)))))
;; outputs (Cons 1 (Cons 2 (Cons 3 (Empty))))
```

```
(insertionSort (Cons 3.0 (Cons 2.0 (Cons 1.0 (Empty)))))
;; outputs (Cons 1.0 (Cons 2.0 (Cons 3.0 (Empty))))
```

Listing 5.3: Implementing sort on Lists

Listing 5.4 defines a "Pair" type that accepts values of any type, defined with the type variables a and b. It implements the function addPairs to add the first and second values of two pairs.

The type class constraints are specified as the last argument to the type declaration construct, restricting both elements of the pair to be of type Num, defined in Listing 5.2. This allows us to use the + operation on the individual elements of pairs.

We use record accessors to extract the individual elements of the pairs, where the field x of type a is type checked under the constraint (Num a), and the field y of b is type checked under the constraint (Num b).

Finally, we call the function addPairs with pairs of type (Pair Integer Integer) and (Pair Integer Double), with each call resolving the + function to use an overloaded instance of appropriate type.

```
(record Pair [x y] [a b])

(type addPairs [(Pair a b) (Pair a b) (Pair a b)]
               [(Num a) (Num b)])
(define addPairs (lambda [p1 p2]
  (Pair (+ (Pair-x p1) (Pair-x p2)) (+ (Pair-y p1) (Pair-y p2)))))

(addPairs (Pair 1 2) (Pair 3 4))
;; outputs (data-inst Pair {x 4, y 6})

(addPairs (Pair 1 2.0) (Pair 3 4.0))
```

```
;; outputs (data-inst Pair {x 4, y 6.0})
```
Listing 5.4: Implementing type checking for a polymorphic function

Listing 5.5 defines a simple recursive function that uses the overloaded operators defined on the type class Num and Eq. The function accepts three arguments of the generic type a, that are constrained to those types that instantiate Num *and* Eq. This allows us to use the functions -, *, and == defined on those type classes for all the values passed to the function.

As noted earlier, since we have not defined type-coercion, which requires the function to accept default values (zero and one) to use in the definition.

When the function is called, the function-passing transform selects an appropriate overload when valid arguments are specified.

```
(type fact [a a a a] [(Num a) (Eq a)])
(define fact
 (lambda [x zero one]
  (if (== zero x)
      one
      (* x (fact (- x one) zero one)))))

(fact 5 0 1) ;; outputs 120
(fact 5.0 0.0 1.0) ;; outputs 120.0
```
Listing 5.5: Recursive polymorphic function with type classes

Listing 5.6 presents a more practical example of using type classes. We define a structurally recursive definition for polynomials with the algebraic datatype MPoly. The type contains two cases: one to define constants with the constructor Const, and the other to define the form $a * x + b$ with the constructor ProdPlus, where a and b are themselves polynomials.

We then define the functions addPolynomials and mulPolynomials to define addition and multiplication on type MPoly. The functions use the helper functions scale, normalPoly, and mul-var.

Finally, we create an instance of the type class Num with the type MPoly. This allows us to write programs that use the overloaded functions like (+) and (-) on polynomials, just like any other instances of the type class Num.

```
(data MPoly
 (Const [c] [double])
 (ProdPlus [p1 x p2] [MPoly string MPoly]))

(define scale (lambda [a p]
 (cond (== 0.0 a)     (Const 0.0)
       (Const? p)     (Const (* a (Const-c p)))
       (ProdPlus? p) (ProdPlus (scale a (ProdPlus-p1 p))
                      (ProdPlus-x p)
                      (scale a (ProdPlus-p2 p))))))

(define normalPoly (lambda [p1 x p2]
 (if (*prim-and (Const? p1) (== 0.0 (Const-c p1))) p2
     (ProdPlus p1 x p2))))

(define addPolynomials (lambda [p1 p2]
 (cond (*prim-and (Const? p1) (Const? p2))
       (Const (+ (Const-c p1) (Const-c p2)))
       (*prim-and (ProdPlus? p1) (Const? p2))
       (ProdPlus (ProdPlus-p1 p1)
                 (ProdPlus-x p1)
                 (addPolynomials p2 (ProdPlus-p2 p1)))
       (*prim-and (Const? p1) (ProdPlus? p2))
       (ProdPlus (ProdPlus-p1 p2)
                 (ProdPlus-x p2)
                 (addPolynomials p1 (ProdPlus-p2 p2)))
       (*prim-and (ProdPlus? p1) (ProdPlus? p2))
       (normalPoly (addPolynomials (ProdPlus-p1 p1)
                                   (ProdPlus-p1 p2))
                   (ProdPlus-x p1)
```

```
                  (addPolynomials (ProdPlus-p2 p1) (ProdPlus-p2
                     p2))))))

(define mul-var (lambda [x p]
 (cond (Const? p)    (ProdPlus p x (Const 0.0))
       (ProdPlus? p) (ProdPlus (mul-var x (ProdPlus-p1 p)) (
          ProdPlus-x p) (mul-var x (ProdPlus-p2 p))))))

(define mulPolynomials (lambda [p1 p2]
 (cond (Const? p1)    (scale (Const-c p1) p2)
       (ProdPlus? p1) (addPolynomials
                       (mulPolynomials (ProdPlus-p1 p1)
                                       (mul-var (ProdPlus-x p1) p2
                                          ))
                       (mulPolynomials (ProdPlus-p2 p1) p2)))))

(typeclass-inst Num MPoly
 (+ [p1 p2] (addPolynomials p1 p2))
 (- [p1 p2] (addPolynomials p1 (scale -1.0 p2)))
 (* [p1 p2] (mulPolynomials p1 p2))
 (neg [p] (scale -1.0 p)))

(+ (ProdPlus (Const 1.0) "x" (Const 2.0)) (ProdPlus (Const 1.0) "x
   " (Const 2.0)))
;; outputs (data-inst ProdPlus {p1 (data-inst Const {c 2.0}), x x,
    p2 (data-inst Const {c 4.0})})

(* (Const 2.0) (ProdPlus (Const 1.0) "x" (Const 2.0)))
;; outputs (data-inst ProdPlus {p1 (data-inst Const {c 2.0}), x x,
    p2 (data-inst Const {c 4.0})})
```

Listing 5.6: Defining polynomials as an instance of Num

5.4.1 Impact of implementing type classes

We have seen several examples of programs using our implementation of type classes. By extending the System O semantics with type classes, we notice the following advantages:

- Type classes provide an expressive way to group together related behavior, which scales well as programs grow, and improves readability of the code. However, we lose the ability to define overloaded functions independent of type classes, which reduces some flexibility in the overloaded function definitions. To alleviate this limitation, some type class implementations allow defining functions with the same name in multiple type classes, as long as the fully qualified names of the functions are different. In our implementation, we limit defining functions with the same name in multiple type classes, as we lack a mechanism to define high level modules.

- By extending the type checker to allow constraints on type variables using type classes, we can avoid writing lengthy type constraints, and have a concise and understandable mechanism for defining type signatures.

- Utilizing the System O dynamic semantics to implement type classes has made the language easier to understand and implement. In developing the interpreter for this study, this has been of value, as the implementation remains fairly simple, making it easier to add new features as the language grows.

Chapter 6

Additional Optimizations

The function-passing transform implemented in Chapter 4 reduces the overhead of selecting an overloaded function at run-time, by transforming expressions during the type checking process. In this chapter, we describe optimizations on the function-passing transform that aim to further reduce the run-time overhead. We modify our interpreter implementation to include these optimizations, and present several examples to demonstrate their advantages.

6.1 Impact of the function-passing transform

In this section, we look at the code generated by the function-passing transform, which will help us identify areas of optimization in the transform. We take another look at the Pair example from Chapter 5, and observe the transformation of the polymorphic function addPairs. Listing 6.1 redefines the type and the function.

```
(record Pair [x y] [a b])

(type addPairs [(Pair a b) (Pair a b) (Pair a b)]
               [(Num a) (Num b)])
(define addPairs (lambda [p1 p2]
  (Pair (+ (Pair-x p1) (Pair-x p2)) (+ (Pair-y p1) (Pair-y p2)))))
```

Listing 6.1: Defining the polymorphic function addPairs

Listing 6.2 shows the transformed expression for the function addPairs. The function now accepts as arguments two implementations of a + function as the type Pair is defined on two potentially different type variables, a and b, specified in the arguments +:a and +:b. The names are generated as a part of the transformation process, based on the type variables defined by Pair. The underlying function definition is transformed to use the implementation of these functions passed as parameters. Note that Listing 6.2 is generated by commenting out parts of the implementation, as the final implementation of the interpreter generates optimized code during the transformation.

```
(define addPairs
 (lambda [+:a +:b]
  (lambda [p1 p2]
   ((lambda [+:a +:b]
    (Pair (+:a (Pair-x p1) (Pair-x p2))
          (+:b (Pair-y p1) (Pair-y p2)))) +:a +:b))))
```

Listing 6.2: Transformed definition of the polymorphic function

Finally, the calls to the function addPairs provide implementations for the + function based on the type of the actual arguments passed to the function. This is demonstrated in Listing 6.3. The functions *+:integer* and *+:double* are primitive operations defined as a part of the initial environment in Chapter 4.

```
(addPairs (Pair 1 2) (Pair 3 4))
;; transforms to ((addPairs *+:integer* *+:integer*) (Pair 1 2)
                                                     (Pair 3 4))
;; outputs (data-inst Pair {x 4, y 6})

(addPairs (Pair 1 2.0) (Pair 3 4.0))
;; transforms to ((addPairs *+:integer* *+:double*) (Pair 1 2.0)
                                                    (Pair 3 4.0))
;; outputs (data-inst Pair {x 4, y 6.0})
```

Listing 6.3: Transformed calls to the polymorphic function

6.2 Optimizing the transform

The function-passing transform significantly reduces the overhead of resolving overloading at run-time, because it is all done during the type checking process. However, it is possible to optimize the transformation further, improving the performance and readability of the resulting code. In this section, we describe the optimizations we have implemented, and their impact in improving the performance and readability of the generated code.

6.2.1 Eliminating redundant λ expressions

As mentioned in Chapter 4, and observed in Listing 6.2, the function-passing transform generates code with redundant λ expressions. This would result in the interpreter making multiple function calls with the same parameter for each expression that requires the overloaded function definitions.

Lambda lowering is the process of moving parts of expressions that do not depend on the function parameters, out of the function definition [16]. This applies to conditional expressions where some branches may not depend on the function parameters.

Similar to Lambda lowering, expression lifting is the process of moving an entire expression out of the λ expression, if it does not depend on the parameters [16]. This applies to function application. For the scope of our interpreter, we apply expression lifting in the context of λ expressions that accept overloaded function definitions as parameters. We identify expressions that do not depend on function parameters, then move them out of the function body, eliminating the need for the λ expressions entirely.

β-reduction is reduction of expressions by function application [3]. The λ expression is eliminated by substituting the value of the argument for the parameter in the function's body.

These procedures optimize the expressions generated by the rule ($\forall$I) to create λ expressions only for those expressions that use overloaded functions. This process looks at every expression generated when traversing the code tree during the transformation, and adds λ expressions only where overloaded functions are defined, or called.

Listing 6.4 demonstrates the transformation from Listing 6.2 after this optimization. Here, β-reduction is applied to substitute the λ expression by applying the arguments +:a and +:b directly. For functions with deeply nested definitions or multiple recursive calls, this optimization can greatly improve the quality of the generated code.

```
(define addPairs
 (lambda [+:a +:b]
  (lambda [p1 p2]
   (Pair (+:a (Pair-x p1) (Pair-x p2))
         (+:b (Pair-y p1) (Pair-y p2))))))
```

Listing 6.4: Transformed expressions with redundant lambdas removed

6.2.2 Reducing the overhead of function calls

The next optimization involves generating new unique variables for overloaded function instances with the concrete type parameters specified. The function definitions are recursively revised to use these parameters directly in their expressions.

When the type checker encounters an overloaded function being applied, it identifies the specific overloaded instance to be called, as discussed in Chapter 4. It then declares a new function at the top-level, whose name is generated using the function name, and the type of arguments that it is being invoked with. The function definition then applies the overloaded arguments to the new name-mangled function.

This ad-hoc monomorphization aims to reduce the overhead of redundant function calls during run-time, and additionally improves the quality of the code generated.

Adding this optimization makes the language more suited for being implemented in a fully compiled system. Defining these functions would mean that concrete versions of overloaded functions are generated at compile-time, instead of the parameters being supplied at run-time, which would considerably improve run-time performance.

6.2.3 Examples

Listing 6.5 shows two transformed functions, created for calls to the function addPairs, with arguments of type (Integer, Integer) and (Integer, Double) respectively.

```
(define *addPairs*+:integer***+:integer**
 (lambda [p1 p2]
   (Pair (*+:integer* (Pair-x p1) (Pair-x p2))
         (*+:integer* (Pair-y p1) (Pair-y p2)))))

(define *addPairs*+:integer***+:double**
 (lambda [p1 p2]
   (Pair (*+:integer* (Pair-x p1) (Pair-x p2))
         (*+:double*  (Pair-y p1) (Pair-y p2)))))
```

Listing 6.5: Concrete functions generated for addPairs

Finally, the calls to the function addPairs are transformed to use these newly defined functions directly.

```
(addPairs (Pair 1 2) (Pair 3 4))
;; transforms to (*addPairs*+:integer***+:integer** (Pair 1 2)
                                                    (Pair 3 4))
;; outputs (data-inst Pair {x 4, y 6})

(addPairs (Pair 1 2.0) (Pair 3 4.0))
;; transforms to (*addPairs*+:integer***+:double** (Pair 1 2.0)
                                                   (Pair 3 4.0))
;; outputs (data-inst Pair {x 4, y 6.0})
```

Listing 6.6: Optimized calls to the polymorphic function

In Listing 6.7, we look at the impact of the transformation on the recursive fact function, defined in Chapter 5.

```
(type fact [a a a a] [(Num a) (Eq a)])
(define fact
 (lambda [x zero one]
  (if (== zero x)
      one
      (* x (fact (- x one) zero one)))))
```

Listing 6.7: Recursive polymorphic function with type classes

On transformation, this function definition is transformed into a function that accepts three arguments, for the functions -, *, and ==, all constrained by the type classes Num and Eq, as shown in Listing 6.8.

```
(define fact
 (lambda [*:a ==:a -:a]
  (lambda [x zero one]
   (if (==:a zero x)
       one
       (*:a x ((fact *:a ==:a -:a) (-:a x one) zero one))))))
```

Listing 6.8: Transformed version of the function fact

Finally, calls to this function generate functions appropriate for the type of arguments passed. We observe this transformation in the recursive call as well, where the optimized fact function makes a recursive call to the same optimized version of itself.

```
;; function call:
(fact 5 0 1)

;; generates the function:
(define *fact**:integer**==:integer**-:integer**
 (lambda [x zero one]
  (if (*==:integer* zero x)
      one
      (**:integer* x (*fact**:integer**==:integer**-:integer**
         (*-:integer* x one) zero one)))))

;; resulting in the call:
(*fact**:integer**==:integer**-:integer** 5 0 1)
```

```
;; function call:
(fact 5.0 0.0 1.0)

;; generates the function:
(define *fact**:double**==:double**-:double**
 (lambda [x zero one]
  (if (*==:double* zero x)
      one
      (**:double* x (*fact**:double**==:double**-:double** (*-:
         double* x one) zero one)))))

;; resulting in the call:
(*fact**:double**==:double**-:double** 5.0 0.0 1.0)
```
Listing 6.9: Calls to the transformed function fact

6.3 Impact of the optimizations

From the examples listed in the above section, we notice that the generated code is more concise, and reduces the number of run-time operations.

By cleaning up the generated expressions to use λ expressions only where necessary, the overhead of redundant function calls is reduced, as there are fewer λ expressions to execute. The generated code also becomes much more readable, and easier to debug.

Building on the above optimization, by selectively defining functions for concrete definitions of polymorphic functions, we are further able to reduce the run-time overhead. The expressions using overloaded functions are completely converted to expressions that apply functions which have already been defined.

This optimization also makes the language more suited for implementation in a fully compiled system, where the monomorphization will generate concrete versions of overloaded functions at compile-time.

One noticeable side effect of this optimization is namespace pollution. For the scope of this interpreter, the function names generated are unique enough to prevent collision.

6.4 Comparing the System O extensions with other implementations

In this section, we implement a `Rational` type, representing rational numbers, with numerator and denominator of type `Int`. We add common math operations to the type with the goal to use it in programs like one would use any other numeric type. We use this example to contrast the differences in implementing this type and related functions in a common programming language, Java, and in System O. For brevity, we skip some boilerplate and implementation details, and focus on how type definitions and overloading are implemented in each example.

In Listing 6.10, we implement the type in Java. The private fields `numerator` and `denominator` represent the numerator and denominator parts of the rational number. The class implements the functions `add`, `mul`, and `div` on the type.

The overloaded function `meanSquare` calculates the mean square of two input numbers. The function is overloaded to work on values of type `double` or `Rational`.

```
class Rational {
  private int numerator;
  private int denominator;

  public Rational(int n, int d) {
    // set the private fields
  }

  public Rational add(Rational c) {
    // implementation of add
  }

  public Rational mul(Rational c) {
    // implementation of mul
  }

  public Rational div(Rational c) {
    // implementation of div
  }
}

class Main {
  public static double meanSquare(double x, double y) {
    return ((x * x) + (y * y))/2;
  }

  public static Rational meanSquare(Rational x, Rational y) {
    return x.mul(x).add(y.mul(y)).div(new Rational(2, 1));
  }

  public static void main(String args[]) {
    System.out.println(meanSquare(1.0, 2.0));
    System.out.println(meanSquare(new Rational(1, 2), new Rational
        (1, 3)));
  }
}
```

Listing 6.10: Rationals in Java

The calls to the function meanSquare are resolved at compile time, with the compiler using the order, number, and type of the arguments to decide which overloaded instance to call.

We notice that operations like add and mul are defined as functions encapsulated within the class. Any new operations on the type would need to be added within the class, which may not always be possible if the class is imported from another package.

In a broader sense, this limits the scope of being able to define polymorphic behavior across classes and modules, as it must be declared with the class definition. One possible solution to this problem is an implementation like "Extension Methods" in the language C#, that allow augmenting methods to modules such that they appear to be a part of the same public interface [4].

In Listing 6.11, we rewrite the Rationals example in System O. We define the type constructor Rational, accepting two values of type Int for the numerator and denominator. The functions add, mul, and div are assumed to be overloaded and implemented for primitive types. We overload them here to accept arguments of Rational type.

The function meanSquare is defined using the overloaded definitions of add, div, and square, as denoted in the type annotation. The function accepts an additional parameter, z, as the divisor. Since overloaded functions need to vary in the type of the first argument, we cannot define an overloaded function of the form fromInteger :: Integer → a to eliminate the need for an additional parameter.

```
data Rational = Rational Int Int
```

```
inst add :: Rational -> Rational -> Rational
-- function implementation here

inst mul :: Rational -> Rational -> Rational
-- function implementation here

inst div :: Rational -> Rational -> Rational
-- function implementation here

meanSquare :: (add :: a -> a -> a,
               div :: a -> a -> a,
               mul :: a -> a -> a) -> a -> a -> a -> a
meanSquare x y z = div (add (mul x x) (mul y y)) z
```
Listing 6.11: Rationals in System O

As discussed in Chapter 3, System O allows defining overloaded functions independent of a grouping, which can impact the readability and maintainability of code as the size of the program grows. We also notice the tedious type signature for the function meanSquare arising from the need to list the overloaded functions used in the type annotation.

Finally, in Listing 6.12, we implement the function meanSquare using the extensions that we have defined on top of System O. The program reuses the definitions for the type class Num, and the functions $+$ and $*$ from Listings 5.2 and 5.6. We assume that the type class Fractional has been defined with the method /, and that the primitive type Double instantiates this class to provide an implementation for division.

This results in the function squareSum being simply restricted in its signature by the type classes Num and Fractional, as specified in the type annotation. The definition of the function uses the functions $+$ and $/$ implemented by instances of the type classes Num and Fractional. Like in the System

O implementation, an extra parameter `z` has to be supplied to the function to specify the divisor.

```
(record Rational [numerator denominator] [Int Int])

(typeclass-inst Num Rational
  (+ [x y] ;; implementation of +)
  (- [x y] ;; implementation of -)
  (* [x y] ;; implementation of *)
  (neg [x] ;; implementation of neg))

(typeclass-inst Fractional Rational
  (/ [x y] ;; implementation of /))

(type meanSquare [a a a a] [(Num a) (Fractional a)])
(define meanSquare
  (lambda [x y z]
    (/ (+ (* x x) (* x y)) z)))
```

Listing 6.12: Rationals in extended System O

From the simple examples above, we notice that the code written using the extended System O has the following advantages over the other examples:

- The ability to define type classes elegantly solves the problem with code littering and the restrictions of class inheritance seen in Listing 6.10.
- The type annotations on functions are much more concise, when compared to Listing 6.11. They make the code easier to understand and extend as requirements may grow.
- Common programming language constructs like algebraic data types allow programmers to express types with lesser boilerplate.

Further, extending System O with the type class construct allows us to utilize the underlying dynamic semantics, which makes the language easier to understand and implement. In implementing the interpreter for this study, this has been of great value, as the implementation remains fairly simple despite having many constructs used in common programming languages.

An alternative approach to implementing type classes would be to allow the ability to provide aliases to groups of functions. This would be an intermediate approach between the System O programming language and our extensions, providing more flexibility while solving the problem of lengthy type annotations.

However, by defining type classes, we lose the flexibility provided by System O in defining overloaded instances with any type, independent of a grouping. We are also restricted by the limitation that overloaded functions should differ in the type of the first argument, which limits us from defining some common type classes like Parser and Monad, which is a limitation beyond the scope of our implementation. As a workaround, we can define specific functions on types for operations that would otherwise be defined via type class instances. This approach resembles the implementation of certain common type class functions in Elm [10], a functional programming language for web applications that does not support type classes. Though suitable for one-off cases, this approach can affect the readability of programs, as we lose the mechanism to group together related functions, and a more rigid solution is desirable.

Chapter 7

Conclusion and Future Work

We have presented modest extensions to System O, that utilize dynamic semantics to implement a type class-based approach to ad-hoc polymorphism. By adding static type checking and a subsequent transform, we are able to eliminate the run-time overhead of ad-hoc polymorphism. We have developed an interpreter in Clojure, implementing all the semantics, extensions, and optimizations discussed in this study.

The programming language we have developed in this study extends System O with the goal to make the language more suited to practical applications. We show several examples using this language that demonstrate real-world programming scenarios.

Of the extensions we have added to System O, the ability to define and instantiate type classes allows programmers to write programs more expressively, with the ability to specify concise function constraints. Building the type classes on top of the dynamic semantics allows us to retain the theoretical results demonstrated by System O.

We also introduced several optimizations to the function-passing transform defined by System O. These optimizations reduce the overhead of redundant function calls at run-time and also simplify the generated code.

7.1 Future Work

A desirable feature to add to this programming language would be subtyping. It would allow for more fine-grained declaration of overloading behavior across type hierarchies, and help us utilize the dynamic semantics to its full extent.

A system like the one we have developed would also benefit from being applied in a compiled language like ALTO [2]. This would help to fully utilize the performance optimizations implemented in this study.

Bibliography

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] ALTO Programming Language. `https://cs.rit.edu/~anh/alto.html`.

[3] Henk P. Barendregt. The Lambda Calculus - Its Syntax and Semantics. In *Studies in logic and the foundations of mathematics*, 1985.

[4] Mihály Biczó, Krisztián Pócza, and Zoltán Porkoláb. Runtime Access Control in C# 3.0 Using Extension Methods. In *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, volume 30, pages 41–60, 2009.

[5] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, pages 4–7, 1985.

[6] Giuseppe Castagna. Covariance and Controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *ArXiv*, abs/1809.01427, 2020.

[7] Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.

[8] Ruy J. G. B. De Queiroz. A Proof-Theoretic Account of Programming and the Role of Reduction Rules. *Dialectica*, 42(4):265–282, 1988.

[9] L. Peter Deutsch and Edmund Berkeley. The LISP Implementafion for the PDP-1 Computer, 1964.

[10] Elm Programming Language. `https://elm-lang.org/`.

[11] Jan Heering and Paul Klint. Semantics of Programming Languages: A Tool-Oriented Approach. *SIGPLAN Not.*, 35(3):39–48, mar 2000.

[12] Eugene Kindler and Ivan Krivý. Object-oriented simulation of systems with sophisticated control. *International Journal of General Systems - INT J GEN SYSTEM*, 40:313–343, 01 2005.

[13] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, pages 348–375, 1978.

[14] Scott Milton and Heinz W. Schmidt. Dynamic Dispatch in Object-Oriented Languages. Technical report, CSIRO – Division of Information Technology, 1994.

[15] J. Garrett Morris. A Simple Semantics for Haskell Overloading. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, page 107–118, 2014.

[16] Arthur Nunes-Harwitt. From Naïve to Norvig: On Deriving a PROLOG Compiler. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, page 70–78, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Martin Odersky and Konstantin Läufer. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 54–67, New York, NY, USA, 1996. Association for Computing Machinery.

[18] Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, page 135–146, New York, NY, USA, 1995. Association for Computing Machinery.

[19] Atsushi Ohori. A Simple Semantics for ML Polymorphism. In *Proceedings of Fourth ACM/IFIP Conference on Functional Programming Languages and Computer Architecture, London, England, ACM Press*, pages 281–292, September 1989.

[20] John Peterson and Mark Jones. Implementing Type Classes. *ACM SIGPLAN Notices*, 28, 12 1995.

[21] Clojure Programming Language. `https://clojure.org/`.

[22] Mark Shields and Simon Peyton Jones. Object-Oriented Style Overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, September 2001.

[23] Christopher Strachey. Fundamental Concepts in Programming Languages. *Lecture notes for International Summer School in Computer Programming, Copenhagen*, pages 36–37, 1967.

[24] Thomas van Noort, Peter Achten, and Rinus Plasmeijer. Ad-Hoc Polymorphism and Dynamic Typing in a Statically Typed Functional Language. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, page 73–84, 2010.

[25] Axiom: The Scientific Computation System. `http://axiom-developer.org/`.

[26] Philip Wadler and Stephen Blott. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *POPL '89*, 1989.