Overloading

Elizabeth Labrada Deniz *[1]

[1] *Computer Science Department (DCC), University of Chile, Chile*

Abstract

1 Introduction

The list of programming languages that have incorporated overloading is very large. In this list can be included, as known languages like Scala, C++, Java and Haskell. Overloading is described as a kind of polymorphism, where one name is associated with several function implementations, and the choice in each use depends on the context where it is used.

In this brief survey, we present concept related to overloading and relevant works that formalize overloading with different characteristics.

2 Background

In this section we explain some concepts related to overloading, like polymorphism and the different kind of overloading.

2.1 Polymorphism

Cardelli and Wegner [**?**] present a classification of the different types of polyphormism which result relevant to understand overloading and its differences with another kind of polyphormism(Figure 1).

$$Polyphormism = \begin{cases} universal & \begin{cases} parametric \\ inclusion \end{cases} \\ ad\ hoc & \begin{cases} overloading \\ coercion \end{cases} \end{cases}$$

Figure 1: Varieties of polyphormism.

On the one hand, universally polymorphic functions typically work on an infinite number of types, where the types share a common structure. Parametric polymorphism occurs when a function defined over a range of types has a single implemetation, acting in the same way for each type [**?**, **?**, **?**]. The identity function is one of the simplest examples of parametric polymorphic function, where for any type, the behavior is the same. Another example can be the function `length`, which operates with lists of any type. Otherwise, inclusion polymorphism is used to model subtypes and inheritance. For instance, in object-oriented paradigm, an object can be viewed as polymorphic because it belongs to different classes.

On the other hand, ad-hoc polymorphism [**?**, **?**], is obtained when a function is defined over several different types and may behave in unrelated ways for each type. Overloading and coercion polymorphism are classified as the two major subcategories of ad-hoc polymorphism, according to Figure 1. In general, we are in present of overloading when the same variable name is used to denote different functions, then the context is essential to decide which function is selected by a particular instance of the name. An example of the overloading function is +, since it is applicable to both integer and real arguments. Additionally, a coercion [**?**] is a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error. The function + is also an example of coercion, if it is defined only for real addition, and integer arguments are always coerced to corresponding reals.

2.2 Static Overloading

Programming Languages like Java, C++ and C# implement static overloading, i.e., that at compile time it is selected the most appropriate function's definition for a function call, only according to the static type of the arguments. For example, if we have the Java code below, where `C` is a subclass of `B`, and `B` is a subclass of `A`, then for the invocation of the method `m`, is select the first definition. As can be observed, the selection of the implementation of the method `m`, it is based in the static type of the argument `e`, not in the dynamic type.

```
class O {
public void m (A e){...}
public void m (B e){...}
}
```

*Funded by grant CONICYT, CONICYT-PCHA/Doctorado Nacional/2015-63140148

```
...
O o = new O();
A e = new B();
o.m(e);
```

2.3 Dynamic Overloading

Multi-methods is considered a collection of overloaded methods associated to the same message, where the selection occur dynamically, according to run-time types of the reciver and of the arguments [**?**]. When selection occur dynamically, we are in presence of dynamic overloading. Thus, with this kind of overloading, in the above example of Java code, for the invocation of the method `m`, is select the second definition. Dynamic overloading seems to provide more flexibility and and accuracy selecting the most specialized implementation for a method invocation.

Common Lisp Object System(CLSO) [**?**] is an object-oriented extension to Common Lisp which implements dynamic overloading. It is worth noting that CLSO do not admit any static checking, therefore is dynamically typed. For instance, the code below in the in most languages with static type checking, results in an static error by ambiguity. However, in CLSO is considered the second implementation for method `m` more specific than the first, thus can be executed without ambiguity. In CLSO, the order of the parameters is crucial for the resolution of overloading.

```
   class A {}
class B extends A {}
def m(x:  A, y:  B): Int = 1
def m(x:  B, y:  A): Int = 2
m(new B, new B)
```

3 Overloading

In the preset section we report some different works related to the overloading field. We start with type class, a powerful mechanism which allow overloading, combined with parametric polyphormism. Then we present an interesting work that explain overloading in object-oriented programming languages, with static or dynamic overloading.

3.1 Type classes

Type class in Haskell [**?**] is a sort of interface that defines some behavior supporting overloading and parametric polymorphism. A type class is defined by Nipkow et al. [**?**] , as a set of types, which all happen to provide a certain set of functions. Haskell has a static type system, therefore at compile time, the type of every expression is known and the overloading is resolved.

The operator (==) for equality is widely used in most languages. Commonly overloaded, this polymorphic operator can be defined using type classes and instance declarations. The Haskell code showed below defines a type class that provides an interface for the (==) operator in a straightforward way. It is worth noting that this type class could have another functionality, like (/=). Once we have this type class, we can define instance for each different types, for instance, `Eq Int` and `Eq Char`, where `eqInt` and `eqChar`, are the implementation for the (==) operator per each type. The class Eq is used as a constraint for type variables that require to support equality inspection.

```
  class Eq a where
(==) ::  a -> a -> bool
instance Eq Int where
(==) = eqInt
instance Eq Char where
(==) = eqChar
```

As we mentioned previously, the overloading resolution in Haskell is static. For this, it is carried out at compile-time a transformation of the code [**?**]. A program that contain class and instance declarations it is translated to an equivalent program that does not. The overloaded variables are also eliminated, with the introduction of "dictionaries", which include the implementation for each functionality of an instance.

Haskell supports overloading with the definition of type classes, but all the overloaded instances must share a common type pattern.

Since the original type inference system is undecidable [7], some additional (syntactical) restrictions had to be imposed to obtain a type system where most general typings can efectively be computed [**?**].

The type system of ML[21], known as the Hindley-Milner type system, has two stand-out features. First, it supports type inference; type inference allows the programmer to write programs without explicit type annotations, while retaining full static type checking. Consider the following ML code fragment. The second major feature of the ML type system is parametric polymorphism. A type system supports polymorphism if it allows objects to have more than one type. The canonical example of a polymorphic function is the identity function: let id = fun x -¿ x;; Hindley-Milner infers most generic types from programs; the concept of principal types formally de

nes this feature[4]. Aside from the inherent complexity of supporting overloading, other problems arise from interactions between overloading and other programming language facilities. Especially problematic are the interactions with advanced module systems, automatic type coercions, and inheritance facilities. Consequently, languages that do support over-

loading (e.g. Ada, C++, SML) do so with restrictions.

All articles revised use Hindley/Damas/Milner system.

Type classes permit the systematic overloading of function names while retaining the advantages of the Hindley/Damas/Milner system: every expression which has a type has a most general type which can be inferred automatically [**?**]. The main purpose of this paper is to give what we believe to be the simplest algorithm published so far, a contribution for implementors. At the same time we present a correspondingly simple type inference system, a contribution aimed at users of the language. The algorithm is sound and complete with respect to the inference system, and both are very close to their ML-counterparts.

The overloading is global en type classes?? No formalize para la semantics

In the work of Camarão et al. [**?**] is summarized some of the limitations or requirements of type classes such as: the declaration of an overloaded function requires a class declarations, all the overloaded instances must share a common type pattern and definitions of overloaded symbols must occur in global instance declarations.

open world assumption where the set of declared instances is considered open to extension (from different modules, introduced at link time).

Ribeiro and Camarão [**?**] note that a type system for Haskell allows distinct derivations for ambiguous expressions, which are usually rejected by the type inference algorithm. Also, they remark that the Haskell's open world approach affects the standard definition of ambiguity. Buscar una definicion standart

Context-dependent overloading, is a form of overloading where the selection of the definition for an application function depends not only on the types of the arguments, but also on the call context. For instance, if we have the expression $m\ e$, we use for the correct selection of the definition of m, the type of e and the information of the context in which $m\ e$ is used.

```
  class M a where m ::  Char -> a
class E a where e ::  a
instance M Char where m ...
instance M Bool where m ...
instance E Char where e ... (m e) :: Bool
```

Coherence establishes a single well-defined meaning for each expression. That is, a coherent semantics is such that, for any well-typed expression e:

3.2 Overloading in Object Oriented Paradigm

Featherweight Multi Java (FMJ) [**?**, **?**] is an extension of Featherweight Java (FJ) [**?**] with multi-methods.

$$
\begin{array}{lll}
L & ::= \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} & \text{classes} \\
K & ::= C(\overline{C}\ \overline{f})\{\mathsf{super}(\overline{f});\ \mathsf{this}.\overline{f} = \overline{f}\} & \text{constructors} \\
M & ::= C\ m\ (\overline{C}\ \overline{x})\{\mathsf{return}\ e;\} & \text{methods} \\
e & ::= x \mid e.f \mid e.m(\overline{e}) \mid \mathsf{new}\ C(\overline{e}) & \text{expressions} \\
v & ::= \mathsf{new}\ C(\overline{v}) & \text{values}
\end{array}
$$

Figure 2: Syntax of FMJ.

FJ is a basic version of Java, which focuses on the following set of features: class definitions, object creation, method invocation, field access, inheritance, subtyping and method recursion through this. Figure 2 shows the syntax of FMJ, which is minimal and simple. For more explanation, it can be consulted in [**?**]. Some important aspects to note related to overloading in these work [**?**, **?**] are:

- All the inherited overloaded methods are copied into the subclass.
- The receiver type of the method invocation has no precedence over the argument types, when the dynamic overloading selection is performed.
- All method invocations are annotated with the type selected during static type checking, in order to choose the best specialized branch during the dynamic overloading method selection. Thus, at run-time it is sound to select only a specialization of the static type.
- A procedure to select the most appropriate branch at run-time using both the dynamic type of the arguments and the annotated static type guarantees that no ambiguity can dynamically occur in well-typed programs.

In FMJ is used the concept of multi-types, which represents the types of multi-methods. Formally, a multi-types is a set of arrows types, with the following shape: $\{\overline{C_1} \to C_1, \cdots, \overline{C_n} \to C_n\}$ or $\{\overline{\overline{C} \to C}\}$, in a compact form. For example, if we have the following sequence of methods definition:
$C_1\ m\ (\overline{C_1}\ \overline{x})\{\mathsf{return}\ e_1;\}, \cdots, C_n\ m\ (\overline{C_n}\ \overline{x})\{\mathsf{return}\ e_n;\}$
or $\overline{C\ m\ (\overline{C}\ \overline{x})\{\mathsf{return}\ e;\}}$, for brevity, then the corresponding multi-type of the method m would be $\{\overline{\overline{C} \to C}\}$.

Another important point is that statically, is checked that the multi-types associated to every multi-method is well formed. For this, they define the function wellformed, formally described in Definition 1. The first condition in this definition requires that all input types are distinct. Collaterally it imposes that a muti-method has the same number of

parameters in each definition, unlike Java. The second condition guarantees that if during the dynamic overloading method selection, is chosen a specialized branch according to the static type,then it is safe.

Definition 1 (Local Well-Formedness of Multi–Types)**.** *A multi-type* $\{\overline{\overline{B} \to B}\}$ *is well-formed, denoted by* $\mathsf{wellformed}(\{\overline{\overline{B} \to B}\})$*, if* $\forall(\overline{B_i} \to B_i), (\overline{B_j} \to B_j) \in \{\overline{\overline{B} \to B}\}$ *the following conditions are verified:*

1. $\overline{B_i} \neq \overline{B_j}$
2. $\overline{B_i} <: \overline{B_j} \Rightarrow B_i <: B_j$.

A function minsel, Definition 3, is crucial for the static resolution of overloading. It is responsible for choosing the most specialized type of a multi-method invocation, given a parameter types set and a multi-type corresponding to a multi-method. If the function is defined means that it can be chosen a definition for the multi-method call, without ambiguity.

Definition 2 (Set of minimal arrow types)**.** *Given a set of arrow types* $\{\overline{\overline{B} \to B}\}$*,* $\mathsf{MIN}(\{\overline{\overline{B} \to B}\})$ *denote the set of minimal arrow types defined as follow:* $\mathsf{MIN}(\{\overline{\overline{B} \to B}\}) \stackrel{\text{def}}{=} \{\overline{B_i} \to B_i \in \{\overline{\overline{B} \to B}\} \mid \forall(\overline{B_j} \to B_j) \in \{\overline{\overline{B} \to B}\}\ s.t\ \overline{B_i} \neq \overline{B_j}, \overline{B_j} \not<: \overline{B_i}\}$.

Definition 3 (Most specialized selection)**.** *Given some parameter types* $\overline{C}$ *and a multi-type* $\{\overline{\overline{B} \to B}\}$*, then* $\mathsf{minsel}(\overline{C}, \{\overline{\overline{B} \to B}\}) \stackrel{\text{def}}{=} \overline{B_i} \to B_i$ *if and only if* $\mathsf{MIN}(\{\overline{B_j} \to B_j \in \{\overline{\overline{B} \to B}\} \mid \overline{C} <: \overline{B_j}\}) = \{\overline{B_i} \to B_i\}$.

Figure 4 present the typing rules for FMJ, which are very similar to FJ. The principal difference is in the rule (TIvnk), with the use of the function mtypesel. This function, described in Figure 3 find the appropriate type for a multi-method call, given the method, its parameter types and the receiver type of the method invocation. The auxiliary functions used in the typing rules are defined in Figure 3.

The operational semantics for FMJ, showed partially in Figure 6, works with the method invocations annotated with the static type selected. For this purpose in [**?**] is defined an annotation function. Also, is defined the function bminsel, similar to minsel, but it receives besides, the annotated type of the method invocation. The function bminsel guarantees the election of a specialized type or the same, related with annotated type, without ambiguity and in a safe way. Figure 5 presents the functions for the operational semantics.

The approach of [**?**] propose an straightforward way to obtain static overloading, changing the rule (RInvk) by the rule (RSInvk), showed in Figure 7. In the method invocation, only it is relevant the static

$$\mathsf{fields}(\mathsf{Object}) = \bullet$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad \mathsf{fields}(D) = \overline{D}\ \overline{g}}{\mathsf{fields}(C) = \overline{D}\ \overline{g},\ \overline{C}\ \overline{f}}$$

$$\mathsf{mtype}(\_, \mathsf{Object}) = \emptyset$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \quad \overline{B\ m\ (\overline{B}\ \overline{x})\{\mathsf{return}\ e;\}} \in \overline{M}}{\mathsf{mtype}(m, C) = \{\overline{\overline{B} \to B}\} \cup \mathsf{mtype}(m, D)}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad m \notin \overline{M}}{\mathsf{mtype}(m, C) = \mathsf{mtype}(m, D)}$$

$$\frac{\mathsf{mtype}(m, C) = \{\overline{\overline{B} \to B}\} \quad \mathsf{minsel}(\overline{C}, \{\overline{\overline{B} \to B}\}) = \overline{D} \to D}{\mathsf{mtypesel}(m, C, \overline{C}) = \overline{D} \to D}$$

Figure 3: Lookup functions for typing of FMJ.

annotated type. In fact, this static overloading approach has a little difference regarding Java. In Java, the second condition request by the wellformed it is not necessary.

It is important to note that FMJ, and more general Java, only admit overloaded method invocation, unlike type classes. Type classes allow function calls and function arguments overloaded. The reason for this is that in Java, the functions are not first-class citizen.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \text{(TVar)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \mathsf{fields}(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i : C_i} \quad \text{(TField)}$$

$$\frac{\Gamma \vdash e : C \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \mathsf{mtypesel}(m, C, \overline{C}) = B \to \overline{B}}{\Gamma \vdash e.m(\overline{e}) : B} \quad \text{(TInvk)}$$

$$\frac{\mathsf{fields}(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash e : C \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \mathsf{new}\ C(\overline{e}) : C} \quad \text{(TNew)}$$

$$\frac{\overline{x} : \overline{B}, \mathsf{this} : C \vdash\ e : E \qquad E <: B \qquad \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\}}{B\ m\ (\overline{B}\ \overline{x})\{\mathsf{return}\ e;\}\ \mathsf{OK\ IN}\ C} \quad \text{(TMethod)}$$

$$\frac{K = C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\mathsf{super}(\overline{g});\ \mathsf{this}.\overline{f} = \overline{f}\} \quad \mathsf{fields}(D) = \overline{D}\ \overline{g} \quad \mathsf{mtype}(m, C) = \{\overline{\overline{B} \to B}\} \quad \wedge\ \mathsf{wellformed}(\{\overline{\overline{B} \to B}\})\ \mathsf{for\ all}\ m\ \mathsf{in}\ \overline{M}}{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\}\ \mathsf{OK}} \quad \text{(TClass)}$$

Figure 4: Typing rules for FMJ.

$$\frac{\mathsf{mtype}(m, C) = \{\overline{\overline{B} \to B}\} \qquad \mathsf{bminsel}(\overline{C}, \{\overline{\overline{B} \to B}\}, \overline{E}) = \overline{D} \to D}{\mathsf{bmtypesel}(m, C, \overline{C}, \overline{E}) = \overline{D} \to D}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad B\ m\ (\overline{B}\ \overline{x})\{\mathsf{return}\ e;\} \in \overline{M}}{\mathsf{mbody}(m, C, \overline{B}) = (\overline{x}, e)}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \qquad m \notin \overline{M}}{\mathsf{mbody}(m, C, \overline{B}) = \mathsf{mbody}(m, D, \overline{B})}$$

$$\frac{\mathsf{bmtypesel}(m, C, \overline{C}, \overline{E}) = \overline{D} \to D \qquad \mathsf{mbody}(m, C, \overline{D}) = (\overline{x}, e)}{\mathsf{mbodysel}(m, C, \overline{C}, \overline{E}) = (\overline{x}, e)}$$

Figure 5: Lookup functions for the operational semantic of FMJ.

$$\frac{\mathsf{fields}(C) = \overline{C}\ \overline{f}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f_i} = \mathsf{v_i}} \quad \text{(RField)}$$

$$\frac{\mathsf{mbodysel}(m, C, \overline{D}, \overline{E}) = (\overline{x}, e_0)}{\mathsf{new}\ C(\overline{v}).m(\overline{\mathsf{new\ D}(\overline{\mathsf{u}})})^{\overline{E} \to E} \longrightarrow [\overline{x} \mapsto \overline{\mathsf{new\ D}(\overline{\mathsf{u}})}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})]e_0} \quad \text{(RInvk)}$$

Figure 6: Operational semantics for FMJ.

$$\frac{\mathsf{mbody}(m, C, \overline{E}) = (\overline{x}, e_0)}{\mathsf{new}\ C(\overline{v}).m(\overline{\mathsf{new\ D}(\overline{\mathsf{u}})})^{\overline{E} \to E} \longrightarrow [\overline{x} \mapsto \overline{\mathsf{new\ D}(\overline{\mathsf{u}})}, \mathsf{this} \mapsto \mathsf{new}\ C(\overline{v})]e_0} \quad \text{(RSInvk)}$$

Figure 7: Operational semantics for FMJ with static overloading.