Overloading

Elizabeth Labrada Deniz *[1]

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

Abstract

1 Introduction

Explicar Overloadin, ejemplos, para que sirve. Hablar de overl $\mathsf{wellformed}(\{\overline{S} \to S\})$

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 [4] 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 [4, 6, 7]. 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 [4, 7], 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 [4] 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){...}
}
...
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

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

selection occur dynamically, according to run-time types of the reciver and of the arguments [2]. 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) [3] 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 Ad-hoc Polymorphism

3.1 Type classes

3.2 Featherweight Java with dynamic and static overloading

Featherweight Multi Java (FMJ) [1, 2] is an extension of Featherweight Java (FJ) [5] with multi-methods. 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 [1]. Some important aspects to note related to overloading in these work [1, 2] 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.

$$L ::= \mathsf{class}\ C\ \mathsf{extends}\ C\ \{\overline{C}\ \overline{f};\ K;\ \overline{M}\} \quad \text{classes}$$

$$K ::= C(\overline{C}\ \overline{f})\{\mathsf{super}(\overline{f});\ \mathsf{this}.\overline{f} = \overline{f}\} \quad \text{constructors}$$

$$M ::= C\ m\ (\overline{C}\ \overline{x})\{\mathsf{return}\ e;\} \quad \text{methods}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid \mathsf{new}\ C(\overline{e}) \quad \text{expressions}$$

$$v ::= \mathsf{new}\ C(\overline{v}) \quad \text{values}$$

Figure 2: Syntax of FMJ.

- 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:* $\mathsf{MIN}(\{\overline{\overline{B} \to B}\}) \stackrel{\text{def}}{=} \{\overline{\overline{B_i} \to B_i} \in \{\overline{\overline{B} \to B}\} \mid \forall(\overline{\overline{B_j} \to B_j}) \in \{\overline{\overline{B} \to B}\}\ s.t\ \overline{B_i} \neq \overline{B_j}, \overline{B_j} \not<: \overline{B_i}\}$.

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

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.

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

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

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

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

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

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

Figure 3: Lookup functions for typing of FMJ.

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 [1] 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 [1] 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

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

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

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

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

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

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

Figure 4: Typing rules for 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.

References

[1] M. Aleksy, V. Amaral, R. Gitzel, J. Power, J. Waldron, L. Bettini, S. Capecchi, and B. Venneri. Featherweight java with dynamic and static overloading. *Science of Computer Programming*, 74(5):261 – 278, 2009.

[2] L. Bettini, S. Capecchi, and B. Venneri. Featherweight java with multi-methods. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 83–92, New York, NY, USA, 2007. ACM.

[3] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *SIGPLAN Not.*, 23(SI):1–142, Sept. 1988.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.

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

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

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

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

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

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

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

Figure 6: Operational semantics for FMJ.

[5] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[6] S. L. Kilpatrick. Ad hoc: Overloading and language design. Master's thesis, University of Texas at Austin, 2010.

[7] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 60–76, Austin, TX, USA, Jan. 1989.

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

Figure 7: Operational semantics for FMJ with static overloading.