A Statistical Transformation Tool for Natural Semantics

Salwa M. Ali  
Faculty of Science, Ain Shams University, Cairo, Egypt.

Mohammed W. Al-Neama  
Education College for Girls, Mosul University, Mosul – Iraq.

Zaid Ahmed Aljawary  
Faculty of Science and Technology, University of Human, Development / Sulaimaniya, Kurdistan Region, Iraq.

Kasim Abduljabbar Al-Salem  
Faculty of Computer Science, University of Cihan/ Sulaimanya, Sulaimanya – Kurdistan, Iraq.

Abstract  
Natural semantics has become a popular approach among programming language researchers. It is used for specifying many aspects of programming languages. Codes of many programming languages can be transformed into natural semantics. A simple tool is proposed to aid in the development of natural semantics. The tool involves statistics and transformations. The tool has been implemented, and since the target language is Pizza the tool has a good performance.

INTRODUCTION  
Design and specifications of the semantics of a programming language requires a number of tasks, such as statistics and transformations. Tools help to perform these tasks quickly and accurately.

There are many tools designed to aid in the development of semantics. Some of such tools are: TYPOL which is an implementation of natural semantics [1]. TYPOL may be compiled into Prolog or a functional notation based on attribute grammars.

The CENTURE system, a generic interactive environment, when provided with the description of a particular programming language it produces a language specific environment [2]. LATOS, a light weight animation tool for operational semantics. It takes as input a superset of Miranda and produce a proper declarative program which can be type-checked and executed by the appropriate language system [3].

LATOS can be compiled into Miranda and subsequently into Haskell. RML, the Relational Meta-language System which compiles natural semantics specifications into C [4].

The ASF+SDF Meta-environment, a specification environment based on the ASF+SFD formalism which combines the Algebraic Specification Formalism (SDF). This environment is an interactive environment that supports the specification of language [5].

Pattern matching is fundamental in our transformation, it is accomplished by a test of significance such as the t-test or ANOVA. Analysis of variance (ANOVA) is a statistical method used to test differences between two or more means [6].

The T-test is a common method for comparing the mean of one group to a value or the mean of one group to another. T-tests are very useful because they usually perform well in the face of minor to moderate departures from normality of the underlying group distributions [7].

The contribution of this paper is to show that it is possible to build a useful tool with simple means to transform natural semantics into Pizza programming language.

2. SPECIFICATION OF NATURAL SEMANTICS  
Natural semantics of a programming language is represented by a set of inference rules as shown in Figure 1.
Figure 1: Inference Rules of Natural Semantics

Syntax of Axioms and Rules:
The syntax of an inference rule is described as follows:

- Each rule has a number of premises written above the solid line, and one conclusion written below the solid line.
- A rule may have a side condition written to the right of the solid line. This condition must be a Boolean expression.
- Rules with no premises are called axioms and the solid line is then omitted. An axiom without a side condition has the following syntax:

\[
\frac{E_1 \rightarrow R_1 P_1, \ldots, E_n \rightarrow R_n P_n}{P_0 \rightarrow R E_0}
\]

An axiom with a side condition has the form:

\[
\frac{E_1 \rightarrow R_1 P_1, \ldots, E_n \rightarrow R_n P_n}{P_0 \rightarrow R E_0} \quad \text{if } E_b
\]

- Rules with no premises are called axioms and the solid line is then omitted. An axiom without a side condition has the following syntax:

\[
\frac{E_1 \rightarrow R_1 P_1, \ldots, E_n \rightarrow R_n P_n}{P_0 \rightarrow R E_0}
\]

About the labeled arrows on the syntax of Axioms and rules, each label on the arrow represents the current relation.

Simplifying Rules:
To make the implementation of the natural semantics easier, we introduce some simplifications to the rules.

- Each rule with n premises and a side condition \((if E_b)\) will be considered as a rule with \((n+1)\) premises and no side condition. So if we have a rule of the form:

\[
\frac{E_1 \rightarrow R_1 P_1, \ldots, E_n \rightarrow R_n P_n}{P_0 \rightarrow R E_0}
\]

Then this rule can be represented as follows:

\[
\frac{E_1 \rightarrow R_1 P_1, \ldots, E_n \rightarrow R_n P_n, E_b \rightarrow id \ True}{P_0 \rightarrow R E_0}
\]

Where \(-id\) is the equality relation.

- Since an axiom is a rule with 0 premises, so the distinction between axioms and rules will be dropped.

- Any pattern \(P\) of the form \((P_1\ldots P_n)\) will be considered as a pattern of the form \(C_n P_1\ldots P_n\). For example, the pattern \((x,y,z)\) can be written as \(\text{Triple}(x,y,z)\).

COMPILING PATTERN MATCHING
Pattern matching is an integral part of our transformation. It is the mechanism by which relations inspect arguments in order to locate relevant inference rules.

What is pattern matching:
Pattern matching is a general operation that is used in many different application areas. Abstractly, a pattern is a description of a set of values. Matching a value against a pattern determines whether the value is in the set denoted by the pattern. There are different matching problems depending on the different kinds of values and patterns they deal with.

In string matching, values are finite strings and patterns are regular expressions. A string matching problem is to find the occurrence of a pattern of \(m\) characters \((P_1\ldots P_m)\) in a general very much longer text string \((T_1\ldots T_n)\).

The solution has application in editors and information retrieval programs. String matching is implemented using various kinds of augmented finite automate [8]. Term matching is a built-in feature in modern functional programming languages such as Standard ML [9], and Haskell [10]. It is used to define functions by case analysis on their arguments.

Term matching is implemented by top-down methods. Tree matching is similar to term matching, it is often used in
applications based on rewriting, such as term-rewriting or code selection [11], [12].

Compiling term matching:
Pattern matching allows complex nested patterns, and also partial overlap between different equations. The purpose of compiling pattern matching is to reduce this complexity, making the resulting code more efficient.

A function definition \( f \) can be expressed using the rules in figure 1 as shown in figure 2, where \( \text{ERROR} \) means an error expression and \( \mid \) is an operation defined as:

\[
a \mid b = a \quad \text{if} \quad a \neq \text{FAIL} \\
a \mid b = b
\]

\[
f = \begin{array}{c}
E_1 \rightarrow^{R_1} P_1, \ldots, E_n \rightarrow^{R_n} P_n \\
p_0 \rightarrow^p E_0 \\
E_{11} \rightarrow^{R_{11}} P_{11}, \ldots, E_{m} \rightarrow^{R_{m}} P_{m} \\
F_1 \rightarrow^{R_1} E_1 \\
\vdots \\
E_{m1} \rightarrow^{R_{m1}} P_{m1}, \ldots, E_{mn} \rightarrow^{R_{mn}} P_{mn} \\
p_m \rightarrow^{p_m} E_m
\end{array}
\]

**Figure 2: Inference Rules with \( \mid \) operation and \( \text{ERROR} \)**

The specification in Figure 2 means that given a term to be executed we will try to match each rule in turn. First try the first rule, if the match succeed, the expression \( E_1 \) will be returned. If the match of the first rule not succeeded try the second rule if the match succeeded the expression \( E_2 \) will be returned and so on. Consider the case in which non of the rules succeeded the match in this case an error should be raised. This technique is called pattern-matching.

Pattern Matching in Pizza:
Pizza introduces algebraic data types with pattern matching, it includes support for visitor style patterns directly in the language.

Since Pizza supports pattern matching the above meaning can be expressed in Pizza by using case statements and exception handling. The Pizza code equivalent to the function definition in Figure 2 is written as shown in Figure 3.

```plaintext
try {
    switch (u) {
        case P1: switch (R11 (E11)) {
            case P11: switch (R12 (E12)) {
                switch (R1n (E1n)) {
                    case P1n: return E1;
                }
            }
        }
    }
    default: throw new Error ();
}
}
catch (Error e1) {
    try {
        try {
            switch (u) {
                case Pm: switch (Rm1)(Em1)) {
                    case Pm1:
                        switch (Rmn (Emn)) {
                            case Pmn: returned Em;
                        }
                    }
                    default: throw new Error ();
                }
            (Error em) { System.out.println ("the match is failed")
            }
        }
    }
}
```

**Figure 3: An Equivalent Pizza Code**

IMPLEMENTATION OVERVIEW
In this section we introduce a summary of the implementation strategy to transform natural semantics specifications into an executable Pizza code. The transformation is based on Wadler’s algorithm for compiling pattern matching [13]. Wadler introduced a function match which is the core if the compilation process. This function transforms an expression uses pattern matching, into an equivalent case-expression.

The transformation done by the introduced tool is composed of two main steps: match the premises, and match the conclusion. Each of these steps is described in details as follows:

1. As we have seen each rule which specifies natural semantics consists of a number of premises and one conclusion. This step takes the list of premises, and apply the function match to each of the premises. The result will be a case-expression.
2. In this step we apply the function match again to the conclusion and the case expression resulting from first step.
CONCLUSION

This paper presented a statistical transformation tool that transforms the specification of natural semantics into a highly efficient Pizza code. The tool has a strong typing discipline, it supports polymorphism and overloading. The tool provides parsing facility (based on JavaCC). It supports curried constructors and higher order functions. Since the pattern matching is an integral part of the tool, it is accomplished by a test of significance such as the t-test or ANOVA.

Furthermore since the target language of the tool is Pizza, which supports object-oriented features and offers some functional features, this makes our tool has a good performance.

REFERENCES


