# Introspective, Self-modifying Control Structures and their Implications
# Draft 0

Patrick Sparbanie Connelly
California University of Pennsylvania

December 8, 2012

## 1 Abstract

In the programming of a computer it is sometimes the case that the optimal flow of computation is unknown prior to runtime. That is to say, the programmer(s) responsible for the program's creation can not possibly predict the most efficient way to structure the program while it is being written. This paper will present introspective, self-optimizing control structures as one method for alleviating the efficiency problems created by this issue. In the process, an implementation of one such control structure will be described. Additionally, the implications of programming constructs of this nature will be discussed.

## 2 Introduction

A common idiom in computer programming involves sequentially executing a set of predicates until one that returns a true value is found. Once the truth of a predicate has been discovered, a consequent statement is executed and the flow of computation skips the testing of the remaining predicate-consequent pairs. Since these predicates are mutually exclusive (consider a chain of "if then else if then else..." statements in a C-like imperative language), then the order in which the predicates are evaluated matters. Suppose that we have such a construct and that it is composed of 1000 predicates, only one of who's consequent statement will be executed. Suppose further that the evaluation of each predicate takes a long time (say, $1s$). If it turns out that the $1000th$ predicate is true most of the time, then $999s$ will be wasted most of the time. If this pattern of computation is always true, then the programmer may be considered to be at fault. If however, the frequency of each predicate's truth/falsity cannot be known prior to runtime, or the frequency varies as a function of time (e.g. one predicate happens to be true the most for the first 10 minutes of computation, another happens to be true the most for the next 5, etc.), this process could prove to be unforeseeable

prior to runtime, and expensive as well (especially if the construct is evaluated repeatedly).

A solution to this particular problem is a similar control structure that keeps track of how often each one of it's predicates returns true and that rewrites itself based on this information. The potential for such a solution was the inspiration for the creation of supacond.

# 3 Supacond

Supacond is an introspective, self-modifying conditional control structure written in Common Lisp. The source code can be viewed here. For the less curious, and/or for those who are not lisp-savvy, A simplified, language-agnostic description of supacond's implementation follows.

Each "Supacond" is an instance of a scond object. Every scond has a slot (field) called "sc-list" that holds a list of sc objects. An sc can be though of as a predicate-consequent pair. The scond also has slots that represent the number of times it has been evaluated ("iter-count") and how often it's sc-list is to be resorted ("resort-freq"). For instance, if the scond's iter-count is at 9 and it's resort-freq is 5, it's sc-list will be resorted before it's next evaluation such that resorting occurs when (0 == (iter-count%resort-freq)) .

The criterion used to sort a scond's sc-list is the number of times that each sc's predicate has returned true (it's "truth-count"). The user can specify the resorting frequency as well as whether or not they would like each sc's truth-count to be reset just after a sort has taken place.

In short, supacond provides a solution to the problem outlined in the introduction.

# 4 Implications

Before properly discussing the implications of the existence of a tool such as supacond, a discussion of it's attributes is appropriate. This control structure is *introspective*. That is, it monitors its own behavior. The data a program accumulates regarding it's own behavior is useless to the program itself unless it has the ability to interpret the data in a meaningful way and then rewrite itself based on this interpretation (e.g. it is *self-modifying*). It is precisely these two attributes that make supacond interesting.

I believe that supacond is a particular instance of a more general phenomenon that could prove itself useful to the field of computer science. Generally, this tool is capable of observing patterns in it's flow of computation and of restructuring itself based on said patterns. This is a powerful idea because every program exhibits a pattern, even if that pattern is asymmetric (e.g. inherently sequential). If more generalized meta-programs, similar in spirit to supacond, can be created, it may be possible to write programs that solve the task with which they are charged in the best possible way. This concept becomes more interesting

when one considers a program where the data set provided as input is dynamic. This idea applied to processing the contents of a static file, for instance, is less exciting than when applied to, say, the software that dictates the next decision a robot will make when that decision is based on sensory input coming from a dynamic world. Patterns are used by humans every day to aid them in creating the mental generalizations that enable them to survive. If this sort of faculty can be instilled in computers, we may be one step closer to creating a truly intelligent agent.