# Resolving Metastability Issues During Bootstrapping

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

## ABSTRACT

The fact that CLOS is defined as a CLOS program introduces two categories of issues that must be addressed, namely *bootstrapping issues* and *metastability issues* [2]. Of the two, the latter is the more difficult one, and also the one that has the most negative impact on the *elegance* of the code in that it requires base cases to be handled specially.

We describe *satiation*, a technique by which metastability issues can be turned into bootstrapping issues, thereby simplifying them and keeping the code elegant. Satiation consists of pre-loading the call history of a generic function with respect to a set of argument classes so that the base cases are handled without invoking the full protocol for computing the effective methods at runtime.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

## General Terms

Algorithms, Languages

## Keywords

CLOS, Common Lisp, Bootstrapping, Metastability

## 1. INTRODUCTION

While most Common Lisp implementations have their own native CLOS implementation, PCL [1] is still used in some high-performance implementations, notably SBCL. PCL was written so that CLOS could be added to a pre-CLOS Common Lisp implementation such as the one defined in CLtL [4] without too much effort. Even Common Lisp implementations that do not use PCL (such as ECL) include CLOS late in the process of building a complete system.

SICL[1] takes a different approach. With very few exceptions, SICL is written in entirely standard Common Lisp, and it is designed to be bootstrapped using a conforming Common Lisp implementation, which therefore includes a complete implementation of CLOS. SICL takes advantage of the conforming host by making extensive use of CLOS. In particular, CLOS is bootstrapped *first*, using the host CLOS implementation to break circularity in definitions.

The SICL implementation of CLOS is a truly metacircular implementation in that very few compromises are necessary because of bootstrapping or metastability issues.

Since CLOS is defined as a CLOS program, there are necessarily metacircular issues that need to be resolved. The AMOP [2] divides these issues into two families:

- Bootstrapping issues. The canonical example mentioned in the AMOP is that the class `standard-class` is its own metaclass, so it must exist before it is created.
- Metastability issues. Here, the canonical example is the function `compute-discriminating-function` being invoked on itself as a result of a method being added to it.

As mentioned in the AMOP, bootstrapping issues are relatively easy to resolve, because there is only a relatively small number of them, and they can be handled by writing special-case code. So for instance, the class `standard-class` can be defined "by hand" without using `defclass`.

Metastability issues are more difficult to handle because they may require special cases to be handled at runtime so as to avoid infinite computation.

Notice, however, that the canonical example for the metastability family issues is not a problem *per se*. Calling `compute-discriminating-function` on itself is a problem only if the *existing* discriminating function of `compute-discriminating-function` is unable to compute the discriminating function of a standard generic function[2], and whether it can do so depends on the structure of the code.

The technique described in this paper, *satiation*, guarantees

[1]https://github.com/robert-strandh/SICL

[2]`compute-discriminating-function` is a standard generic function.

that existing discriminating functions of specified[3] generic functions can handle arguments of all specified classes.

## 2. PREVIOUS WORK

The AMOP [2] contains a section titled “Living with Circularity”, which describes the essential nature of the two kinds of issues discussed here, namely *bootstrapping issues* and *metastability issues.* That section does not contain a complete list of all possible issues in any implementation of CLOS, and probably could not contain such a list, since it would depend on the exact organization of each particular implementation.

The section in the AMOP has two subsections, one for bootstrapping issues and one for metastability issues. The subsection on bootstrapping issues is more comprehensive.

### 2.1 Bootstrapping issues

The subsection in the AMOP on bootstrapping issues contains two explicit issues.

The first one involves the class `standard-class`, which is the metaclass of all standard classes, including itself. The authors simply suggest creating this class manually.

The second issue involves the fact that generic functions are required in order to create classes, but during bootstrapping, there are no generic functions, since generic functions are instances of classes. The technique used to handle such issues is to define ordinary functions to contain code for essential methods, so that such functions can be called during bootstrapping. To avoid code duplication, the methods defined later in the bootstrapping process simply call those functions.

### 2.2 Metastability issues

The subsection in the AMOP on metastability issues also contains two issues.

The first issue involves the function `slot-value`. As described, the scenario does not correspond to the specification, because the signature of the function `slot-value-using-class` used in the scenario is different from its definition in the specification. Either way, the basis of the scenario is that `slot-value` on some instance would need to access the list of slot descriptions of the class of the instance, and that list is contained in a slot, so that a recursive use of `slot-value` would be required on the class of the instance. However, in a high-performance implementation, a slot reader would not call `slot-value`. The reason is that `slot-value` is much too general, so that unnecessary work would be done. In particular, `slot-value` must find a slot description metaobject with a particular name, whereas this name is already known in the slot reader function. Instead, in a high-performance implementation, the slot reader would access the slot directly by location.[4]

As described in the introduction of this paper, the second issue has to do with `compute-discriminating-function`. Again, the scenario described is an approximation of that of a real high-performance implementation. Their example involves adding a method to some generic function `F`, which would trigger the computation of a new discriminating function for `F`. The metastability issue occurs when `F` happens to be the function `compute-discriminating-function`. In that case, `compute-discriminating-function` would be called with itself as an argument, in which case, according to the AMOP, “the game would of course be over.” Even in a naive implementation without effective-method caching, the scenario would be more complicated than that. In such an implementation, `compute-discriminating-function` would call `compute-applicable-methods-using-classes`[5] and then the function `compute-effective-method`. In an implementation without caching, the real metastability issue occurs in these last two functions. When one of these functions is called, the discriminating function will be invoked, and therefore they will be called recursively.

In a high-performance implementation, on the other hand, what really happens depends on the contents of the cache. If the cache contains an entry that applies to instances of the class `standard-generic-function`, then no metastability issue is present. In such an implementation, the issue occurs only in the initial stages where the cache is empty, and after the cache has been flushed, should the implementation use this technique.

## 3. OUR TECHNIQUE

The technique described in this paper was developed as part of SICL.[6] The system is written entirely in Common Lisp, and it was designed to be bootstrapped from a conforming Common Lisp implementation including CLOS. According to Rhodes [3], few Common Lisp implementations are designed to be bootstrapped this way. Instead, most implementations evolve by incremental modifications to an existing binary image.

A generic function in SICL contains a *cache* (a *call history* in SICL terminology) which associates classes of required arguments with corresponding effective methods, as specifically allowed by the AMOP. The discriminating function of every standard generic function contains two parts: one part computed from the call history and a second *default* part, invoked when the first part fails. This second part of the discriminating function invokes the complete machinery defined by `compute-applicable-methods-using-classes`, `compute-applicable-methods`, and `compute-effective-method`. Crucially, the first part is a relatively simple mechanical translation from a set of call-history entries to an *automaton* that implements the dispatch. This translation requires none of the machinery for computing effective methods.

As hinted in the previous section, metastability issues can be handled automatically, provided that:

[3]The term *specified* is used as in the AMOP to mean an object (such as a class, a generic function, or a method) that is defined by the HyperSpec or the AMOP.

[4]The situation is a bit more complicated due to the fact that the location may vary according to the exact subclass of the specializer of the reader method. In fact, it can even be the case that the slot has a different allocation in different subclasses.

[5]We omit the possibility of the presence of `eql` specializers in order to keep the description manageable.

[6]The SICL repository is public, and it is currently located at https://github.com/robert-strandh/SICL.

- The call history contains entries for every combination of classes of required arguments such that:
  1. the classes are specified by the AMOP, and
  2. there is at least one primary applicable method for the combination.
- When the class hierarchy changes, an entry in the call history is removed only if it involves a modified class.
- When a method is removed from a generic function (possibly because it was replaced with a new one with the same specializers), an entry in the call history is removed only if it involves the method that was removed.

The reason that these conditions automatically handle metastability issues is that the AMOP specifically disallows modifications to specified classes. Then the first part of the discriminating function (i.e., the part computed from the call history) will always handle invocations where classes of required arguments are specified classes, including the function `compute-discriminating-function`.

Our technique, called *satiation*, makes sure that the conditions are met by trying every combination of existing classes as the required arguments of every existing generic function in order to see whether this combination corresponds to an applicable primary method of the generic function. If that is the case, then a corresponding effective method is computed, and an entry is added to the call history. Finally, the resulting discriminating function is computed. Clearly, these computations require a fully-functional machinery for precisely the functions that are being processed.

While it may seem like overkill and an excessive amount of work to compute call history entries for all possible combinations, this work is justified because:

- The satiation machinery is invoked only during bootstrapping, so it does not affect performance at runtime.
- When the machinery is invoked, only a handful of classes and generic functions exist. Therefore, the total amount of work is fairly small.

While it may be possible to invoke the satiation machinery *lazily* during bootstrapping,[7] there is not much to gain by doing that due to the limited amount of work involved. Furthermore, it would still be necessary to make sure that the call history of each specified generic function contains all entries required to avoid metastability issues at runtime, and that work is identical to what the full machinery is designed to accomplish.

Now, in a system where CLOS must be bootstrapped from a pre-CLOS implementation, satiation would require specialized code in the form of ordinary functions (as opposed to generic functions) that do the same work as the specified generic functions on arguments that are instances of specified classes, with much duplicated code as a result.

[7]It can not be invoked lazily at runtime, because then we would be back in a situation with metastability issues.

An object in SICL is either an *immediate object*, a `cons` *cell*, or a *general instance*. A general instance is represented as a two-word header where the first word contains the *class* of the instance, and the second word contains the *rack*. When the instance is a standard object, the rack holds the values of the *slots* of the instance. But general instances are also used for instances of built-in classes, so (for example) an instance of an array would have a rack that contains storage for the elements of the array.

In SICL, we bootstrap CLOS on an existing conforming Common Lisp implementation (the host) as follows:[8]

1. First we use the definitions of the MOP classes to create ordinary host classes, albeit with names in a separate package. Slots with associated accessors will generate ordinary host generic functions.
2. Next, we create *bridge classes* and *bridge generic functions*. A bridge class is a host instance of one of the host classes created in phase 1. A bridge generic function is an instance of the host class `standard-generic-function` created in phase 1, but also of the host class named `funcallable-standard-object`.[9]
3. Third, we use bridge classes and bridge generic functions to create *ersatz classes* and *ersatz generic functions* as instances of the bridge classes previously created. These are target instances represented as a combination of a host *structure* for the header, and a host *simple vector* for the rack. At this stage the class slot of the header of an ersatz object contains a bridge class.
4. The graph of ersatz objects is then *tied* by replacing every bridge class in the class slots of every ersatz object by its equivalent ersatz class, creating a complete graph of only ersatz objects.
5. Finally, we traverse the graph of ersatz objects in order to create an isomorphic graph as a sequence of bytes to become the memory image of the target system.

In the vast majority of cases, the same definitions of functions and classes is used to create host objects in phase 1, bridge objects in phase 2, and ersatz objects in phase 3.

The discriminating function of each bridge generic function is computed using the full machinery executing as a set of host generic functions. This includes the satiation machinery so that the call history of each bridge generic function

[8]The description of the bootstrapping procedure is simplified to avoid clutter. In reality, there are several complications that need to be taken care of.

[9]The class `funcallable-standard-object` is not part of the Common Lisp standard. This is one of the few situations where bootstrapping SICL requires some functionality that is not part of the standard. We attempted to use the class `standard-generic-function` instead, but a major problem with that solution is the conflicting use of specified initialization arguments which are interpreted both by the host class and the bridge class.

is completely pre-computed. Similarly, the discriminating function of each ersatz generic function is computed using the full machinery executing as a set of bridge generic functions.

The end result is a memory image containing all the generic functions specified by the AMOP, and each of these generic functions has the following characteristics:

- The call history contains pre-computed effective methods for all combinations of specified classes that result in at least one primary applicable method.
- When a specified generic function `F` is invoked with itself or some other specified generic function as an argument, then the effective method for handling the call already exists in the call history of `F`.
- Since the AMOP specifically disallows modifications to specified classes after their initial definitions, call-history entries involving specified classes will always be valid.
- Call-history entries involving specified classes are not removed as a result of methods being added or removed in conformance with the AMOP, nor as a result of valid modifications of existing classes.
- The call history is translated into a discriminating function without invoking the machinery for computing effective methods.

This combination of characteristics guarantees the absence of metastability issues at runtime.

## 4. CONCLUSIONS AND FUTURE WORK

We have described a technique that allows us to avoid metastability issues in the implementation of CLOS in our system SICL by replacing those issues by simpler bootstrapping issues. Furthermore, our technique also simplifies bootstrapping by avoiding special cases due to the non-existence of generic functions when the system is bootstrapped. To avoid these issues, we use the *host* generic function machinery in early stages of bootstrapping.

Currently, nothing prevents a specified method on a specified generic function, specializing on specified classes to be modified or removed, and nothing prevents a specified class from being redefined. Should this happen, "the game would of course be over." We imagine a mechanism that protects the user from inadvertently invoking such operations. It should probably be possible to *toggle* the mechanism so that system code can make modifications known to be safe.

When this article was written, SICL was not yet finished, nor even in a state to be executed standalone. However, most of the difficult components (such as the compiler, the garbage collector, and of course CLOS) were in a fairly advanced stage of development. The final verdict on the technique for bootstrapping the system can not be determined until the system is able to run standalone.

## 5. REFERENCES

[1] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging lisp and object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 17–29, New York, NY, USA, 1986. ACM.

[2] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[3] C. Rhodes. Self-sustaining systems. chapter SBCL: A Sanely-Bootstrappable Common Lisp, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008.

[4] G. L. Steele. *COMMON LISP: the language.* 1984. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.