Bootstrapping Common Lisp using Common Lisp

Irène Durand
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

Some Common Lisp implementations evolve through careful modifications to an existing *image*. Most of the remaining implementations are bootstrapped using some lower-level language, typically C. As far as we know, only SBCL is bootstrapped from source code written mainly in Common Lisp. But, in most cases, there is no profound reason for using a language other than Common Lisp for creating a Common Lisp system, though there are some annoying details that have to be dealt with.

We describe the bootstrapping technique used with SICL, a modern implementation of Common Lisp. Though both SICL and the bootstrapping procedure for creating it are still being worked on, they are sufficiently evolved that the big picture outlined in this paper will remain valid. Our technique uses *first-class global environments* to isolate the host environment from the environments required during the bootstrapping procedure. Contrary to SBCL, and implementations written in some other language, in SICL, we build the CLOS MOPclasses and generic functions *first*. This technique allows us to use the CLOS machinery for many other parts of the system, thereby decreasing the amount of special-purpose code, and improving maintainability of the system.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; *Multi-paradigm languages*;

KEYWORDS

CLOS, Common Lisp, Compilation, Bootstrapping

ACM Reference Format:
Irène Durand and Robert Strandh. 2021. Bootstrapping Common Lisp using Common Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.2634314

ELS'19, April 01–02 2019, Genova, Italy

ACM ISBN 978-2-9557474-3-8.
https://doi.org/10.5281/zenodo.2634314

1 INTRODUCTION

In this paper[1] , by *bootstrapping* a Common Lisp system we mean creating some *target* Common Lisp system by building it from its associated source code, using various *tools* and *language processors* to transform that source code into an *executable file* for some typical operating system such as GNU/Linux. The typical way of making such a target Common Lisp system evolve through maintenance, is to modify its source code and then restart the bootstrapping procedure to build an updated executable file.

Not all Common Lisp systems are created so that they can evolve this way. Some systems evolve by the careful modification of an existing *executing image* which is then saved as an executable file that can be executed as usual.

In this paper we will concentrate on the technique of bootstrapping used with SICL[2].

Before we can start investigating different options for bootstrapping, we must deal with an annoying but crucial detail, namely the definition of *source code*. The Free Software Foundation defines it as "the preferred form of a work for making modifications to it". We agree completely with this definition. It excludes the use of code that was automatically produced. In practice, it also excludes code written directly in machine language and most code written in assembly language, with the exception of (a) small code fragments that can not be expressed easily in some other language, and (b) code fragments that are part of a code generator written in some higher-level language.

However, for it to be possible for the source code of a Common Lisp system to be turned into an executable file, there must be *language processors* (i.e., compilers and/or interpreters) available that can handle the languages that the source code is expressed in. The main debate when it comes to bootstrapping techniques seems to be what is meant by *available* in this context. A common definition seems to be something like *whatever is available on a GNU/Linux system out of the box*.

One of the consequences of such a definition of *available* is that, in order to write a Common Lisp system, one has to use some programming language considered lower level than Common Lisp itself. Typically, C plays this role.

In this paper, we argue that one of the main reasons of the creator(s) of a target Common Lisp system wanting such

[1] In this paper, we assume that the reader is familiar with the metaobject protocol for implementing CLOS, as described in the book [1] that is dedicated to the subject.
[2] https://github.com/robert-strandh/SICL

a system in the first place, is that they are convinced of the virtues of this language for writing programs. Furthermore, Common Lisp is uniquely well adapted to writing language processors. The obvious choice for a language for writing a Common Lisp system is therefore Common Lisp itself. Since there is now a multitude of good Common Lisp implementations available and easily installable on widely-used operating systems, we think that Common Lisp should be considered to be a language for which there are language processors *available* for bootstrapping.

2 PREVIOUS WORK

2.1 Overview of existing techniques

In his excellent paper describing how SBCL is bootstrapped [2], Rhodes gives an overview of how different existing Common Lisp systems are made to evolve. Below, we summarize the contents of that paper.

We can divide Common Lisp implementations into those that are mostly written in some other language, and those that are mostly written in Common Lisp.

In the first category, there are implementations that specifically cater to applications written in that other language and that need some scripting capabilities that are supplied by the Common Lisp implementation. Whether it is advantageous or not for these implementations to be written mainly in that other language is outside the scope of this paper.

Of the implementations in the second category that are currently actively used, Rhodes claims[3] that Allegro, LispWorks, CMUCL, Scieneer, and CCL are only possible to build using older versions of the same system, and only using image-based techniques. Only SBCL can be bootstrapped using several other Common Lisp implementations.

Even a Common Lisp implementation that is largely written in Common Lisp such as SBCL has some amount of code written in other languages. In the case of SBCL, Rhodes gives the number 35 000 lines of C and assembly code "for services such as signal handling and garbage collection", of which 8 000 is for the garbage collector. The remaining lines can be summarized as around 2 000 lines per operating system supported. This is a very modest amount of code written in other languages.

2.2 Common Lisp systems in other languages

When a language such as C or C++ is used to implement a Common Lisp system, a small subset of the Common Lisp language is implemented this way. We call that subset the *base* language. The result of the initial bootstrapping procedure is typically an executable file containing the base system. Additional modules are then added to the base system to obtain a complete Common Lisp system. These additional modules must be implemented in the subset of Common Lisp defined by the base language and previously added modules.

There are several issues with this technique. For one thing, some major components that would be more easily expressed in Common Lisp must be written using the implementation language so that new modules can be added to the system, in particular a reader and an evaluator.

Another major issue has to do with maintenance. When one of the additional modules is modified, it is easy to forget exactly what subset of the Common Lisp language is allowed at that point in the bootstrapping procedure. The code for a particular module must often be expressed in some unidiomatic way and it is tempting to make the modified code more idiomatic, but doing so will then break the bootstrapping procedure.

2.3 Common Lisp systems in Common Lisp

Because of the way compilation is defined by the Common Lisp standard, there are some issues that need to be resolved in order for it to be possible for a target Common Lisp system to be bootstrapped on a host Common Lisp system. Since SBCL is very likely the only Common Lisp implementation written mostly in Common Lisp that can be built from an existing Common Lisp implementation, we describe how SBCL solves some of these issues.

2.3.1 Packages and environments. Most existing Common Lisp systems have a single global environment that is used both as the compilation environment and as the run-time environment. Compiling Common Lisp source code requires the existence of definitions of macros, types, etc. in that environment, and when source code for a target Common Lisp system is compiled using a host Common Lisp system, these definitions must be those of the target system. However, with a single global environment there can only be one definition of these entities.

SBCL solves this problem by using different package names for the code of the host system and the target system. In a final step, the packages of the target system are then renamed to conform to the standard.

2.3.2 The compiler and CLOS. Some aspects of CLOS require the presence of the compiler, at least if the resulting code is required to have some reasonable performance. In particular, the compiler is required to create a discriminating function from the effective methods[4] returned by `compute-effective-method`. For that reason, it becomes difficult to use generic functions and standard classes in the code of the compiler itself.

SBCL solves this issue by not using generic functions and standard classes in the code of the compiler. Thus, SBCL can load the compiler into a minimal running target system and then bootstrap CLOS afterwards.

However, not using generic functions and standard classes in the compiler has some of the same problems as Common

[3] For the commercial Common Lisp implementations cited in the paper by Rhodes, he includes a disclaimer that only anecdotal evidence for this information is available.

[4] Recall that the result of a call to `compute-effective-method` is a lambda expression. This lambda expression must be turned into something that is executable, hence the need for an evaluator.

Lisp systems that are written in some other language, namely that care has to be taken to make sure the proper subset of the language is used when the code of the compiler is being worked on. Furthermore, generic functions and standard classes are great tools for structuring complex code, so not being able to use these tools in such a significant and complex part of a Common Lisp implementation negatively affects the clarity and maintainability of the code.

3 THE SICL SOURCE CODE

SICL is a system that is written entirely in Common Lisp. We decided to use the full language to implement the system so as to avoid having to define and remember what subset of the language is allowed for which modules. Thus, the compiler, called Cleavir[5], makes heavy use of generic functions and classes. By using these two types of objects, we can have a compiler that is adaptable to different Common Lisp implementations. It is currently used as the main compiler of Clasp[6], and recently, a Cleavir-based compiler has been written for CLISP[7].

In addition to using the full language for the implementation of SICL, we want the code to be as idiomatic as possible. For example, our definition of the class `t`, looks like this:

```
(defclass t ()
  ()
  (:metaclass built-in-class))
```

This definition clearly expresses the characteristics of the class `t`. It has no superclasses because no superclasses are explicitly mentioned, and the metaclass `built-in-class` does not provide any default superclasses like `standard-class` and `funcallable-standard-class` do. While this definition of the class `t` is clear, it is not operational as is. The metaclass `built-in-class` is an indirect subclass of the class `t`, so the class `t` must exist in order for the class `built-in-class` to exist.

Our definitions of the classes `class` and `standard-class` look like this:[8]

```
(defclass class (specializer)
  ((%name :initform nil :initarg :name ...)
   ...
   (%direct-subclasses :initform '() ...)))
(defclass standard-class (class)
  (...))
```

Again, these definitions are clear. No metaclass option is given, so the metaclass defaults to `standard-class`. Like the defintion of `t`, these definitions are not operational as is, because the class `standard-class` must exist in order to be the metaclass of itself.

In a Common Lisp implementation that must bootstrap CLOS from a subset of the language that does not include CLOS, some other mechanism must be used. As an example of the consequences of the use of such a subset, consider the following definitions from ECL[9]:

```
(defparameter +class-slots+
  `(,@+specializer-slots+
    (name :initarg :name :initform nil ...)
    ...
    (direct-subclasses :initform nil ...)
    ...))
(defparameter +standard-class-slots+
  (append +class-slots+
          '((optimize-slot-access)
            (forward))))
```

Here, two special variables are defined, each one containing the specifications of the direct slots of a class. These two definitions express the exact same information as two `defclass` forms defining the classes `class` and `standard-class`, respectively. However, because the `defclass` form can not be used at this stage of the bootstrapping procedure, a different mechanism must be used.

In addition to using the CLOS machinery for defining the classes defined by the metaobject protocol, we use the same machinery for defining system classes. For example, our definition of the class `symbol` looks like this:

```
(defclass symbol (t)
  ((%name :reader symbol-name)
   (%package :reader symbol-package))
  (:metaclass built-in-class))
```

Not only is this definition clear, it is also operational. By using the CLOS machinery for definitions of system classes, we avoid having to use an additional, special, mechanism for this purpose.

In contrast, consider this definition of the system class `symbol` from SBCL:

```
(define-primitive-object
    (symbol :lowtag other-pointer-lowtag
            :widetag symbol-header-widetag
            :alloc-trans %make-symbol
            :type symbol)
  ...
  (name :ref-trans symbol-name :init :arg)
  (package :ref-trans symbol-package
           :set-trans %set-symbol-package
           :init :null)
  ...)
```

Again, a special mechanism must be used, since CLOS is not available when the type `symbol` must be defined.

The purpose of the SICL bootstrapping procedure is to make these idiomatic definitions operational in the host environment so as to create a graph of objects isomorphic to that of the target system, and then to create the target graph in an executable file.

By doing it this way, we simplify system maintenance. The bootstrapping procedure is able to work with the definitions of classes, generic functions, and methods using the standard

[5] Cleavir resides in the SICL repository on GitHub.
[6] https://github.com/clasp-developers
[7] https://clisp.sourceforge.io/
[8] In reality, there are intermediate classes between `class` and `standard-class` that are not shown here.
[9] https://common-lisp.net/project/ecl/

macros `defclass`, `defgeneric`, and `defmethod`, even though these definitions would not be operational in a system that needs to build up functionality from a language subset that does not include `CLOS`. The SICL maintainer is thus free to alter definitions of core system objects, relying on the bootstrapping procedure to make those definitions operational and ultimately turning them into an executable system.

4 OUR TECHNIQUE

4.1 SICL object representation

A SICL object is represented in one of three different ways:

- As an *immediate* object where the object is stored in the pointer itself, with the appropriate tag bits. Fixnums, characters and single floats are represented this way.
- As a two-word block. This is how `cons` cells are represented.
- As a two-word block called a *header* where the first word points to a class object, and the second word points to a sequence of words, called the *rack*, that contains the slots of the object. All objects other than immediates and `cons` cells are represented this way. We call this representation a *general instance*.

The first word of the rack contains a *stamp* which is a unique integer taken from the class when the instance was created. The stamps of the arguments to a generic function are used by the generic dispatch technique to determine which effective method to execute. The object representation and generic dispatch technique has been described in detail previously [3], but this short summary is sufficient to understand our bootstrapping technique.

In the description of our technique, we use the word *class* in a general way, as an object that can be used as a model for the creation of *instances*. Thus the word *class* does not imply that it is a class in the sense of the host Common Lisp implementation. While this usage of the word *class* may seem odd, recall that a class is just an ordinary Common Lisp object that is passed as an argument to `make-instance` and other functions called by it which then returns a different object. We exploit this idea by supplying our own definition of `make-instance` in different phases of the bootstrapping procedure.

Similarly, we use the word *generic function* in a general way, as an object that can be executed and that can have methods associated with it, providing partial implementations of the generic function. Again, while this usage of the word *generic function* may seem odd, recall that a generic function is simply an ordinary Common Lisp object of type `funcallable-standard-object` for which the ultimate definition (called the *discriminating function*) is computed by combining partial definitions (the *methods*) associated with it. We exploit this fact by providing different representations of generic functions in different phases of the bootstrapping procedure, and by supplying different versions of `compute-discriminating-function` adapted to each phase. Thus, a *generic function* is not a generic function in the sense of the host Common Lisp implementation. However, during the bootstrapping procedure, these objects are executable in the host system, because they are instances of the host class `funcallable-standard-object`.

4.2 Environments for bootstrapping

Our technique uses several first-class global environments [5] to create a graph of objects that is isomorphic to the graph of objects to be written to the executable file instantiating the target Common Lisp implementation. By using first-class global environments, we avoid the problems related to packages and environments cited in Section 2.3. The main feature of our technique, though, is that we create the generic functions and classes of the metaobject protocol *first*.

The environments are filled with definitions mainly as a result of loading files containing production SICL code, though some code specific to bootstrapping is required as discussed at the end of this section. This loading procedure uses the Eclector[10] reader and the Cleavir compiler to produce intermediate code in the form of a fairly conventional *flow graph* of instructions. The Cleavir compiler takes a first-class global environment as an argument, and uses this environment to search for definitions of macros, classes, types, etc. The resulting intermediate code is then translated in two different ways:

(1) Native target code is generated from it, and attached to host objects representing executable target objects such as ordinary functions, generic functions, and methods.[11]
(2) It is translated to a simple subset of Common Lisp code that accesses that same environment for definitions of functions and other objects. This Common Lisp code is then compiled using the host compiler in order to make it executable in the host.

The remainder of this section is concerned with how the host-executable code is used in order to determine the graph of target objects represented as an isomorphic graph of host objects.

4.3 Definitions

In preparation for the bootstrapping procedure, several first-class global environments are created and filled with definitions of SICL macros. The definitions of those macros reside in production SICL files. Little or no special code is required for those definitions.

A number of host object types are used during bootstrapping, in particular symbols, packages, `cons` cells, and integers. However, when such an object is used as an argument to a SICL generic function, a special version of `class-of` assigns a SICL class object as its type. Some of the host functions operating on these kinds of objects are *imported* to our environments in preparation for the bootstrapping procedure.

[10] https://github.com/robert-strandh/Eclector
[11] We do not yet have a code generator for native executable code, so currently this part of the bootstrapping procedure is omitted.

To facilitate the description of our technique, we need some definitions:

Definition 4.1. A *host class* is a class in the host system. If it is an instance of the host class `standard-class`, then it is typically created by the host macro `defclass`.

Definition 4.2. A *host instance* is an instance of a host class. If it is an instance of the host class `standard-object`, then it is typically created by a call to the host function `make-instance` using a host class or the name of a host class.

Definition 4.3. A *host generic function* is a generic function created by the host macro `defgeneric`, so it is a host instance of the host class `generic-function`. Arguments to the discriminating function of such a generic function are host instances. The host function `class-of` is called on some required arguments in order to determine what methods to call.

Definition 4.4. A *host method* is a method created by the host macro `defmethod`, so it is a host instance of the host class `method`. The class specializers of such a method are host classes.

Definition 4.5. A *simple host instance* is a host instance that is neither a host class nor a host generic function.

Definition 4.6. An *ersatz instance* is a target general instance (as defined in Section 4.1) represented as a host data structure, using a host standard object to represent the *header* and a host simple vector to represent the *rack*. In fact, in order for the ersatz instance to be callable as a function in the host system, the header is an instance of the host class `funcallable-standard-object`.

Definition 4.7. An ersatz instance is said to be *pure* if the class slot of the header is also an ersatz instance. An ersatz instance is said to be *impure* if it is not pure. See below for more information on impure ersatz instances.

Definition 4.8. An *ersatz class* is an ersatz instance that can be instantiated to obtain another ersatz instance.

Definition 4.9. An *ersatz generic function* is an ersatz instance that is also a generic function. It is possible for an ersatz generic function to be executed in the host system because the header object is an instance of the host class `funcallable-standard-object`. The methods on an ersatz generic function are ersatz methods.

Definition 4.10. An *ersatz method* is an ersatz instance that is also a method.

Definition 4.11. A *bridge class* is a representation of a target class as a simple host instance. An impure ersatz instance has a bridge class in the class slot of its header. A bridge class can be instantiated to obtain an impure ersatz instance.

Definition 4.12. A *bridge generic function* is a representation of a target generic function as a simple host instance, though in order for it to be executed by the host, it is an instance of the host function `funcallable-standard-object`.

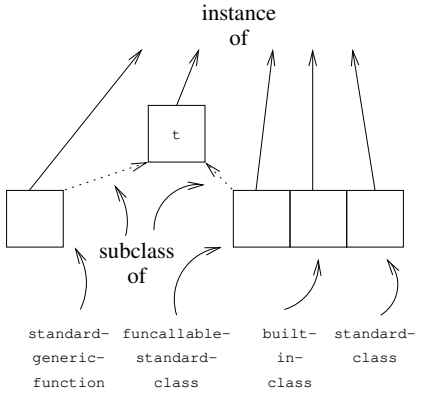

Figure 1: Simplified diagram of MOP classes.

Arguments to a bridge generic function are ersatz instances. The bridge generic function dispatches on the *stamp* (See Section 4.1.) of its required arguments.

The methods on a bridge generic function are bridge methods.

Definition 4.13. A *bridge method* is a target method represented as a simple host instance. The class specializers of such a method are bridge classes. The *method function* of a bridge method is an ordinary host function.

4.4 Bootstrapping phases

The essence of our technique consists of four phases (1 to 4), using six first-class global environments. An initial phase 0 imports host classes to environment E_0. Only classes that are required in phase 1 are imported. Classes `standard-method`, `standard-generic-function`, and the class used to represent slots `standard-direct-slot-definition` are imported with the same. Classes `standard-class`, `built-in-class`, and `funcallable-standard-class` in environment E_0 all refer to one and the same host class, namely a subclass of the host class `funcallable-standard-class`.

In each phase $i > 0$, three first-class global environments are involved, E_{i-1}, E_i, and E_{i+1}. Before phase i starts, E_{i-1} contains classes to be instantiated during phase i, and E_i contains generic functions that are not involved in phase i, but that will be used in phase $i + 1$ to operate on the instances of the classes in E_{i-1}. Some of the generic functions in E_i are accessor functions containing methods that were automatically added as a result of the classes in E_{i-1} being defined. Others are higher-level functions that call those accessors to accomplish tasks such as initialization of various metaobjects, class finalization, creation of effective methods, and creation of discriminating functions.

A phase i has two main steps:

(1) Accessor generic functions are created in E_{i+1} by loading SICL production code containing `defgeneric` forms. These generic functions are accessor functions for MOP classes and MOP generic functions. These functions

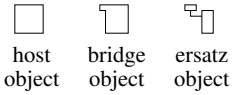

Figure 2: Objects in different phases.

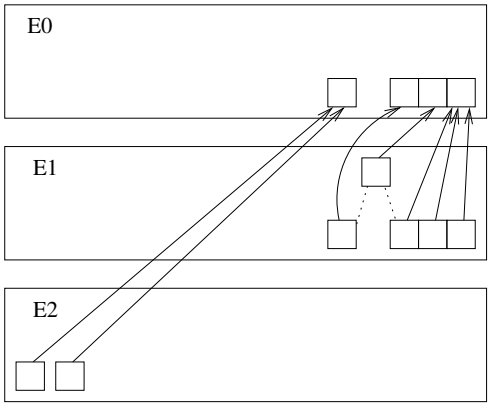

Figure 3: Phase 1.

are created in E_{i+1} rather than in E_i so as to protect the existing functions in E_i that are needed later.

(2) Classes are created in E_i by loading SICL production code containing `defclass` forms. As a result of the creation of these classes, methods are automatically added to the corresponding accessor generic functions in E_{i+1}.

Depending on the phase, SICL production code might be loaded before the first step, between the two steps, or after the last step.

Four phases accomplish the creation of a number of objects, ending with a complete set of ersatz objects. The result of each phase is illustrated by a separate figure. In these figures, the shape of each object illustrates its type as shown in Figure 2.

The four phases accomplish this following results:

(1) Host classes and host class metaclasses in E_0 are used to create host generic functions in E_2 and host classes in E_1. The result of this phase is illustrated in Figure 3.

(2) Host classes in E_1 are used to create bridge generic functions in E_3 and bridge classes in E_2. The result of this phase is illustrated in Figure 4.

(3) Bridge classes in E_2 are used to create impure ersatz generic functions in E_4 and impure ersatz classes in E_3. The result of this phase is illustrated in Figure 5.

(4) Impure ersatz classes in E_3 are used to create pure ersatz generic functions in E_5 and pure ersatz classes in E_4. The result of this phase is illustrated in Figure 6.

The result of these phases is that the impure ersatz generic functions in environment E_4 can operate on the pure ersatz generic function in environment E_5 and on the pure ersatz classes in E_4. But they can also operate on impure ersatz objects, provided their call caches contain entries for the corresponding stamps. Filling the call caches is the purpose of our *satiation* technique [4].

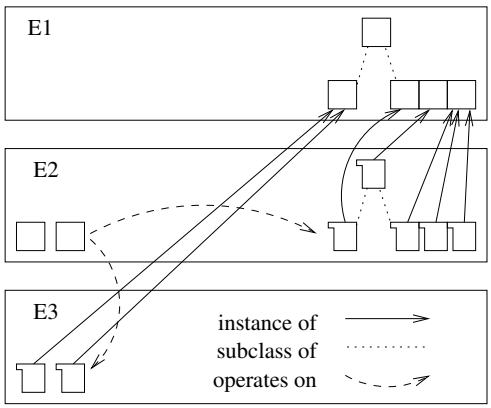

Figure 4: Phase 2.

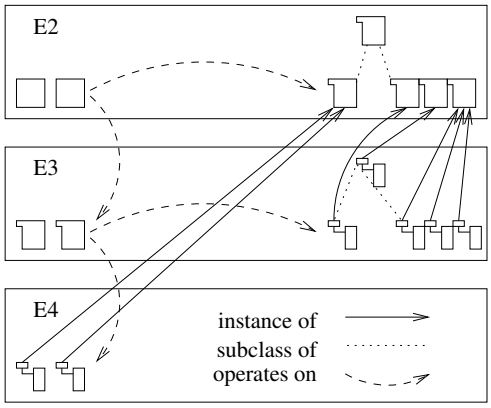

Figure 5: Phase 3.

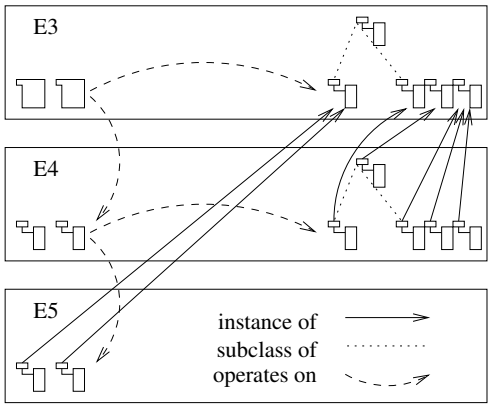

Figure 6: Phase 4.

4.5 Tying the knot

At the end of these four phases, we have fully functional impure ersatz generic functions in environment E_4, and fully functional impure classes in environment E_3. But we still do not have the cyclic graph of metaobjects that a functioning CLOS system requires. Furthermore, there are still bridge generic functions that might be called in order to operate on our impure ersatz metaobjects.

To accomplish the conversion of this hierarchy of objects to a cyclic graph, we need to modify the `class` slot of the headers of each impure metaobject so that instead of referring to a bridge class, it refers to an impure ersatz class. This operation will transform every impure ersatz metaobject into a pure ersatz metaobject. However, there are a few more operations required to completely remove all references to bridge metaobjects:

- Each ersatz metaobject contains a list of the effective slot definition metaobjects of its class as the second word of the rack. In an impure ersatz metaobject, those effective slot definitions are bridge objects. Once the `class` field of the impure ersatz metaobject has been updated, this list must be updated to contain a reference to the list of the ersatz effective slot definitions from the new ersatz class.
- Each ersatz generic function contains a slot containing the method class of the methods on this generic function. In an impure ersatz generic function, this slot refers to a bridge class, so it must also be updated.

We must still find and update all impure ersatz metaobjects in the system. For classes and generic functions, this is trivial, as they are all reachable from the first-class global environment they are defined in. For other object types such as methods, slot-definitions, and method combinations, this is not the case. They must be found by a traversal of the class or generic function metaobject that they are part of. Such a traversal is straightforward.

Before the cyclic graph can be traversed and an isomorphic graph be generated in a native executable file, additional definitions must be loaded:

- Standard classes that are needed in order for the resulting native executable to be viable must be loaded. In particular, definitions of classes such as `symbol`, `package`, `cons`, `sequence`, `list`, `null`, `number`, `rational`, `integer`, and `fixnum` are needed in order for it to be possible to load compiled code into the executing image.
- Many standard functions are also needed, such as functions on packages, lists, hash tables, etc. Functions that operate on first-class global environments are needed as well.
- A simple version of the compiler must be loaded so that the resulting executable image can construct discriminating functions when definitions of generic functions and methods are loaded.

On the other hand, the garbage collector may not be needed in the initial executable image, though the data structures that the garbage collector works with must of course be present so that objects can be laid out in memory.

5 BENEFITS OF OUR TECHNIQUE

Appendix C of "The Art of the Metaobject Protocol" [1] (Living with Circularity) cites a number of ways in which their system handles circularity and avoids bootstrapping and metastability issues.

5.1 Bootstrapping benefits

The first bootstrapping problem that is mentioned is the fact that `standard-class` must exist before it can be created. Their solution is to create this class using some special-case mechanism. Our technique uses the version of `standard-class` in the preceding environment, so this problem is avoided altogether. As a result, we can freely modify the definition of `standard-class` and rerun the bootstrapping procedure. No special case has to be considered.

The second bootstrapping problem mentioned is that generic functions are used for method lookup, but these generic functions can not exist until a significant part of the protocol has been implemented. As an example, take the call to `ensure-class` made as a result of executing the expansion of a `defclass` form. By having `ensure-class` check for the special case when the argument is `standard-class` and by supplying a special function for creating instances of `standard-class` they avoid bootstrapping issues, simply because during bootstrapping, all classes created will be instances of `standard-class`. They also supply a special version of `finalize-inheritance` that checks for the metaclass `standard-class` and calls special-purpose code in this case. With our technique, no such special case is needed. All classes that are instantiated are fully operational in the preceding environment, as is the `finalize-inheritance` generic function.

5.2 Metastability benefits

The first example of a metastability problem mentioned in the book is that `slot-value` calls `slot-value-using-class` which then calls `slot-location` which in turn recursively calls `slot-value` on the class metaobject to access the slot metaobjects of the class. The authors propose to solve this problem by arranging for the function `slot-location` to check for the special argument `effective-slots` and return a predefined location. Our technique does not need this kind of special case, because the function `class-slots` does not call `slot-value` at all. It accesses the `effective-slots` slot directly, using its location. This location has been compiled in during the creation of the effective method and discriminating function for `class-slots`.

The final issue discussed in the book arises because the function `compute-discriminating-function` is also a generic function that can not be called with itself as an argument when a method has been added or removed from it. Again they solve the issue by a special case whereby a test is made to see whether the argument is a standard generic function

(i.e. an instance of `standard-generic-function`) and if so, a special version of `compute-discriminating-function` which is not a generic function is called instead. With our technique, every generic function, `compute-discriminating-function` included, has a *call cache* that includes an effective method that is able to handle arguments that are direct instances of `standard-generic-function`. That call cache entry is not invalidated when `compute-discriminating-function` has new methods added to it, at least not when the methods added respect the restrictions of the metaobject protocol, i.e. that user code is not allowed to add methods that are applicable when given only standard objects as arguments.

5.3 Other benefits

In addition to solving the bootstrapping issues and the metastability issues given in the "The Art of the Metaobject Protocol" book, our technique has several additional benefits.

Since we begin the bootstrapping procedure by defining the classes and generic functions specified by the metaobject protocol, we are able to use the CLOS machinery to define system classes. In a system where CLOS is added late, many system classes must be defined using some other mechanism.

Furthermore, as already mentioned, our technique has great advantages to maintenance. There are no dependencies between CLOS code and other code that require duplication of information that must be kept synchronized when some code is modified.

6 CONCLUSIONS AND FUTURE WORK

We have described a technique for bootstrapping a Common Lisp system using an existing conforming Common Lisp system that is also supported by the library `closer-mop`. To our knowledge, no existing Common Lisp system is bootstrapped this way.

There are several advantages to our technique:

- The full Common Lisp language can be used in order to implement the system, including the compiler, thereby making the code more maintainable.
- By bootstrapping the MOP generic functions and the hierarchy of classes first, we eliminate the bootstrapping problems and metastability problems cited by the AMOP book [1].
- Also, by bootstrapping the MOP machinery first, we take advantage of it by using it to define all the standard system classes, thereby eliminating the need for special mechanisms for this purpose.
- The absence of special mechanisms that are needed in existing implementations for defining many aspects of the system itself, further contributes to the maintainability of our code.

Even though the technique outlined in this paper is known to work, many more aspects of the system need further work, including the bootstrapping technique itself, in order for a native executable to be generated:

- We must supply a (simple) code generator that translates intermediate code to native code. The amount of work required is fairly modest, and mainly consists of creating native code for memory operations such as `car` and `standard-instance-access`, for object allocation, and for simple arithmetic on fixnums.
- Interface code to the operating system must be supplied, in particular for input/output operations.
- We have yet to write the code that translates the host representation of the object graph into a native representation. Special care must be taken for object types that are imported from the host during bootstrapping, such as symbols, numbers, and `cons` cells.

However, we are in no hurry to create a native executable system. The moment we do, we lose a fairly good environment (namely the host Common Lisp system) for debugging our code. Instead, we plan to use the host environment for testing as many aspects of SICL as possible, and for creating support for better debugging capabilities, and only later create a native executable.

In terms of future work, there are still several optimization techniques that need to be implemented for the Cleavir compiler framework.

7 ACKNOWLEDGMENTS

We would like to thank David Murray for providing valuable feedback on early versions of this paper.

REFERENCES

[1] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[2] Christophe Rhodes. Self-sustaining systems. chapter SBCL: A Sanely-Bootstrappable Common Lisp, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8. doi: 10.1007/978-3-540-89275-5_5. URL http://dx.doi.org/10.1007/978-3-540-89275-5_5.

[3] Robert Strandh. Fast generic dispatch for common lisp. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, pages 89:89–89:96, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2931-6. doi: 10.1145/2635648.2635654. URL http://doi.acm.org/10.1145/2635648.2635654.

[4] Robert Strandh. Resolving metastability issues during bootstrapping. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, pages 103:103–103:106, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2931-6. doi: 10.1145/2635648.2635656. URL http://doi.acm.org/10.1145/2635648.2635656.

[5] Robert Strandh. First-class global environments in common lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.