université de BORDEAUX

Bootstrapping Common Lisp on Common Lisp

Irène Durand
Robert Strandh

LaBRI, University of Bordeaux

April, 2019

Context: The SICL project

https://github.com/robert-strandh/SICL
Several objectives:

- Create high-quality *modules* for implementors of Common Lisp systems.
- Improve existing techniques with respect to algorithms and data structures where possible.
- Improve readability and maintainability of code.
- Improve documentation.
- Ultimately, create a new implementation based on these modules.

How Common Lisp systems are built

Two main categories of systems:

1. Systems written mainly in a language other than Common Lisp, typically C or C++.
2. Systems written mainly in Common Lisp.

According to Rhodes (SBCL: A Sanely-Bootstrappable Common Lisp):

1. GCL, ECL, CLISP, ABCL (Java), xcl.
2. Allegro, LispWorks, CMUCL, Scieneer CL, CCL, SBCL.

How Common Lisp systems are built

Of the systems mainly written in Common Lisp there are two ways of making them evolve:

1. Image-based techniques, i.e., the system evolves by careful modifications to an in-memory image, that is then saved to secondary memory to become a new executable.
2. The system is built from a set of source files, using an existing Common Lisp implementation to create an executable for the new system.

Again, according to Rhodes, only SBCL uses the second technique, though CCL is also built from source, even though at the moment, only an old CCL version can be used for the bootstrapping process.

Complications with building Common Lisp systems

Question: Is the build process for creating a Common Lisp system intrinsically harder than the build process for creating a mere file-translating compiler, even for a language as complex as Common Lisp, and if so, why?

Complications with building Common Lisp systems

Possible answer: We think that it is, and we think that it is due to the following difference between the two:

- A file-translating compiler contains essentially only code. That code translates statements and expressions of a source language to object code in assembler or machine language.
- A Common Lisp system, on the other hand, contains not only code, but also complex data structures. The most practical way of creating those data structures is by executing some code.

Imagine, for instance, writing the C source statements that will generate a single generic function with its methods, effective method functions, etc.

Defining a generic function statically

```
struct string class_slots_name =
{
  ADD_HEAP_TAG(&string_class)),
  FIXNUM_BOX(11),
  ...
};

struct generic_function class_slots_function =
{
  ADD_HEAP_TAG(&generic_function_class),
  ...
  ADD_HEAP_TAG(&class_slots_name),
 ...
};
```

Strategies for creating the data structures

There are several possible strategies for creating these data structures:

1. Create them when the system starts. This strategy is used by ECL and Clasp.
2. Create them in an executing subset Common Lisp system, and then save the memory image to a file. This strategy is used by the image-based implementations, but also by SBCL.
3. In a host Common Lisp system, create an octet vector that holds a mirror of the contents of the target system. This strategy is used by SBCL to create an initial "core" image.
4. In a host Common Lisp system, create a graph of objects that is isomorphic to the one in the target system, then save that graph to a file. This is the strategy used by SICL.

Objectives of SICL

We want the code to look "natural". For example:

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

(defclass standard-class (class)
  (...))
```

Objectives of SICL

This is how the same information is expressed in ECL:

```
(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))))
```

Objectives of SICL

In SICL, we take advantage of the CLOS machinery for defining system classes as well as standard classes:

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

Objectives of SICL

This is how the same information is expressed in 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)
  ...)
```

Objectives of SICL

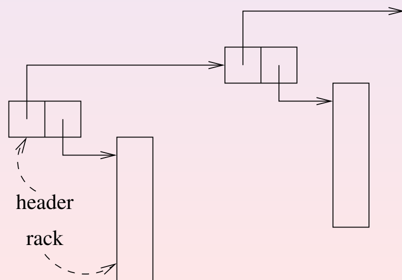
In fact, we can even write this definition in SICL:

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

SICL object representation

SICL has the following three ways of representing objects:

1. Immediate objects stored in the pointer itself, appropriately tagged, like fixnums and characters.
2. `cons` cells represented as a pair of words.
3. All other objects represented as a two-word *header* containing a pointer to the class, and a pointer to the *rack* which has some number of consecutive words.



First-class global environments

In SICL, the global environment is a first-class object, containing the following mappings:

- From names to variables, including information about type, global value, etc.
- From names to functions, including information about type, compiler macro, inlining, etc.
- From names to classes.
- From names to `setf` expanders.
- From names to type expanders.
- From names to method combinations.
- etc.

Having environments be first-class objects, allows us to have several simultaneous instances.

First-class global environments

The Cleavir compiler processes top-level forms relative to the information provided in a first-class global environment, passed to it as an argument.

This technique allows us to execute Cleavir in a host Common Lisp system without impacting the global environment of that system.

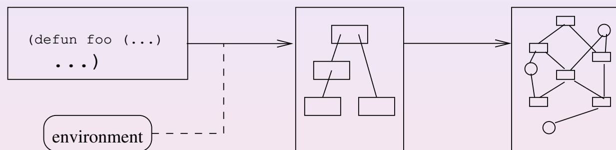
Cleavir compilation phases

The Cleavir compiler works as follows:

1. It takes Common Lisp source code and a first-class global environment and translates the source code to an *abstract syntax tree* (AST).
2. The abstract syntax tree is translated to a *high-level intermediate representation* (HIR), which resembles a traditional instruction graph, except that only Common Lisp objects are manipulated.
3. HIR is translated to implementation-specific backend code.

Here, "implementation specific" neans that the code is different for different Common Lisp implementations, like for Clasp, SICL, SBCL, etc.

Cleavir compilation phases



Executing SICL code in the host

Suppose we have some code with a call to a global function `foo` like this:

```
...
(foo <arg1> ...)
```

We translate that code to AST and then to HIR.

The HIR code then represents something like this:

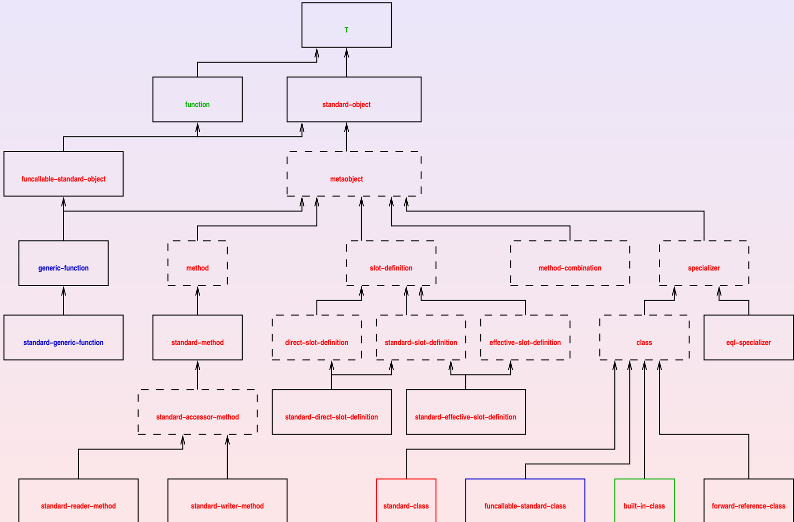
```
(lambda (env)
  (let ((#:G001 (sicl-genv:function-cell env 'foo)))
    ...
    (funcall (car #:G001) <arg1> ...)
```

Executing SICL code in the host

So, we can translate the HIR code to a small subset of Common Lisp.

The resulting code is then *tied* to a particular first-class global environment E by calling it with E as an argument.

MOP class hierarchy



Phase 0

host

cl: standard-generic-function

cl: standard-method

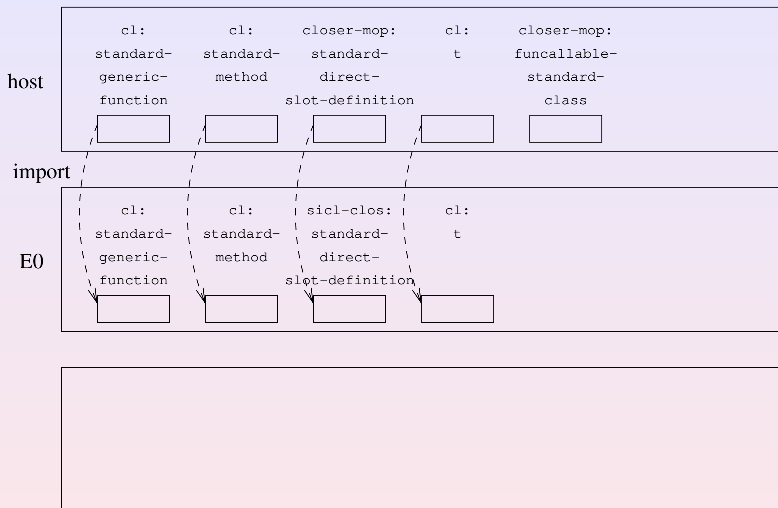
closer-mop: standard-direct-slot-definition

cl: t

closer-mop: funcallable-standard-class

E0

Phase 0



Phase 0

host

cl:
standard-
generic-
function

cl:
standard-
method

closer-mop:
standard-
direct-
slot-definition

cl:
t

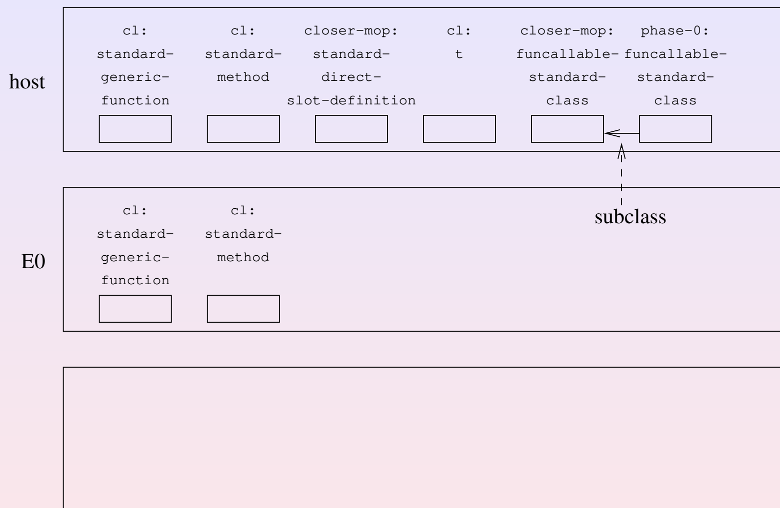
closer-mop:
funcallable-
standard-
class

E0

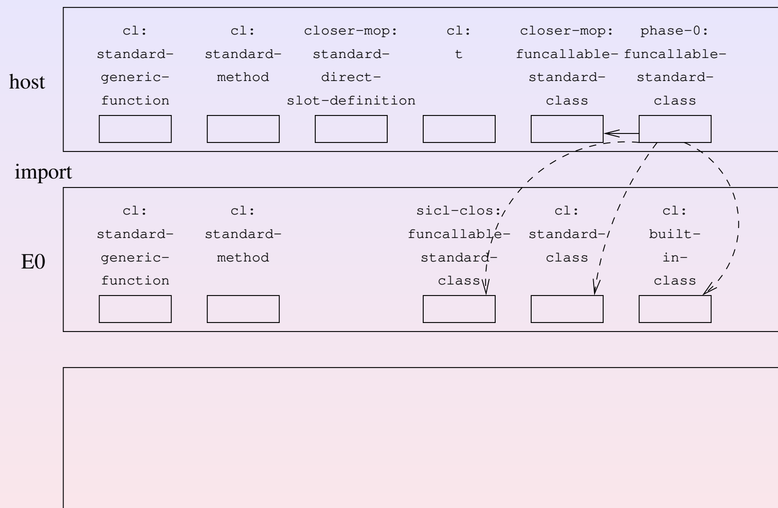
cl:
standard-
generic-
function

cl:
standard-
method

Phase 0



Phase 0



Phase 0

E0

cl: standard-generic-function

cl: standard-method

sicl-clos: funcallable-standard-class

cl: standard-class

cl: built-in-class

Phase 1

E0

cl:
standard-
generic-
function

cl:
standard-
method

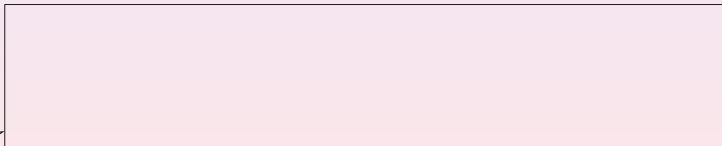
sicl-clos:
funcallable-
standard-
class

cl:
standard-
class

cl:
built-
in-
class

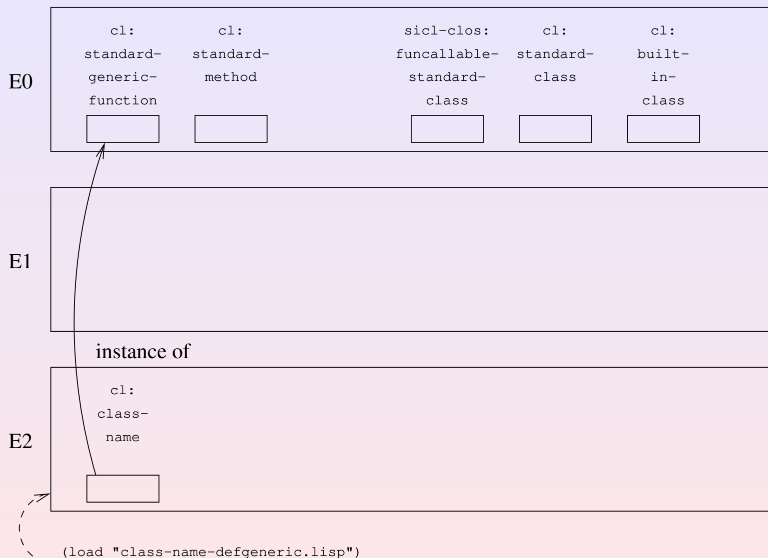
E1

E2

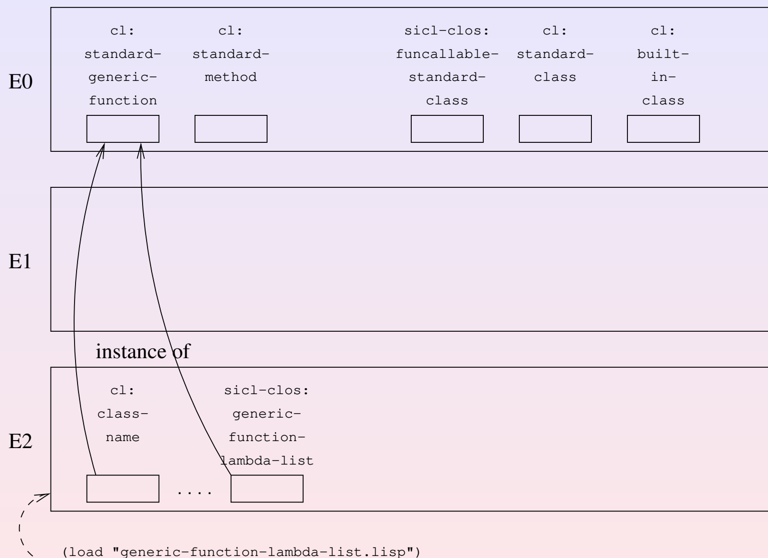


```
(defun cl:ensure-generic-function ...)
(load "defgeneric-defmacro.lisp")
```

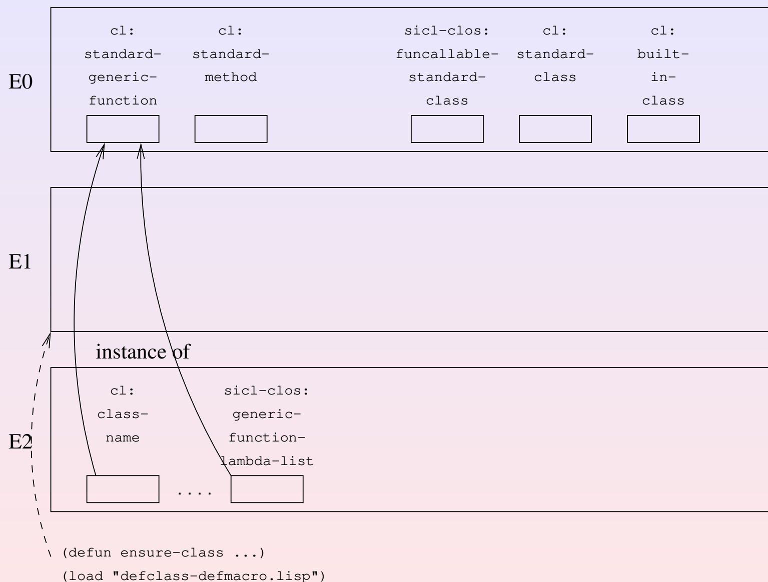
Phase 1



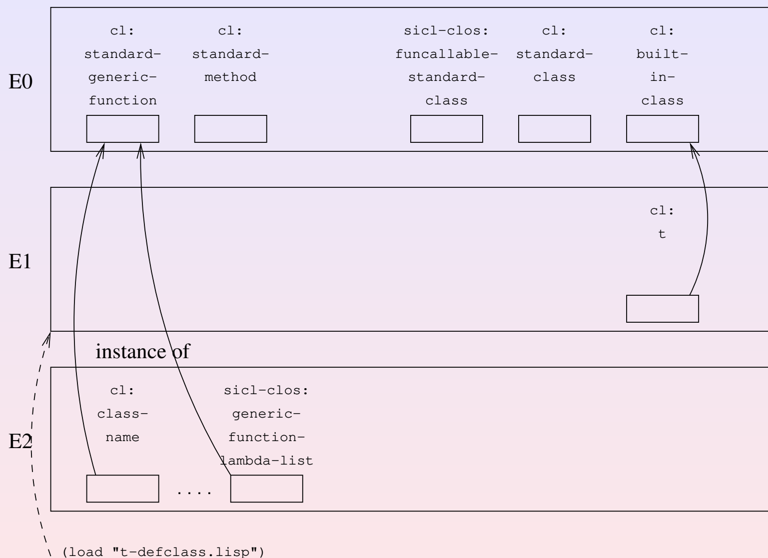
Phase 1



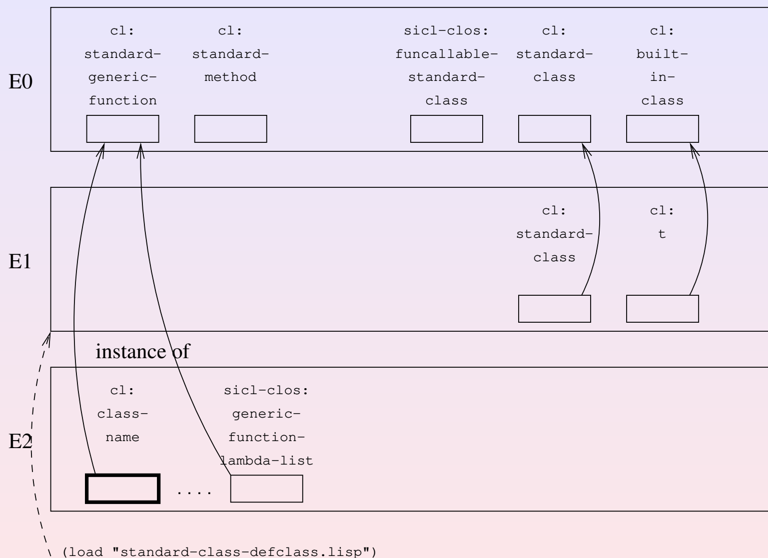
Phase 1



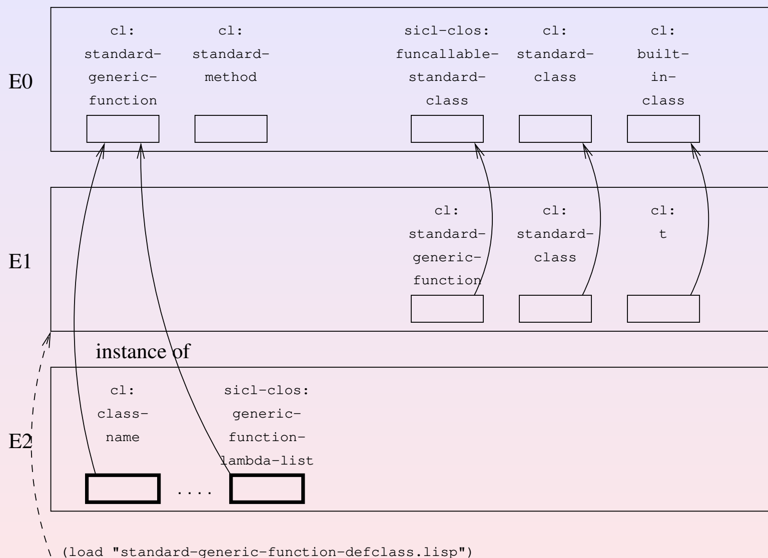
Phase 1



Phase 1



Phase 1

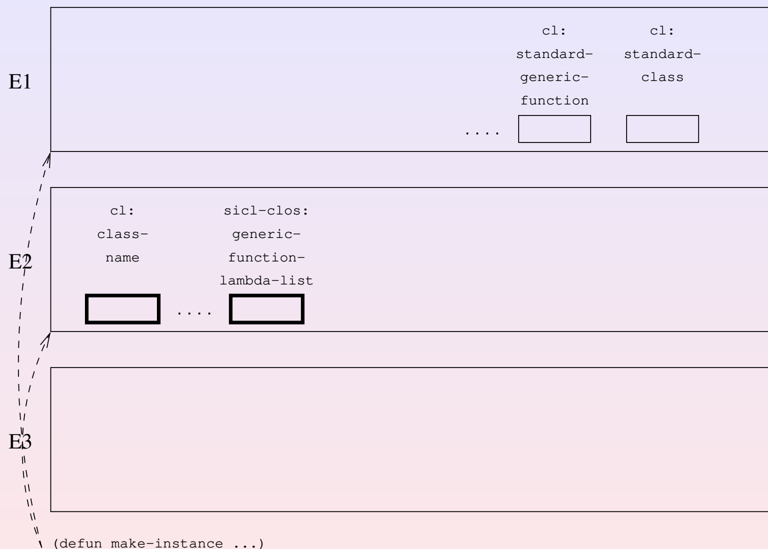


Phase 1

E1

E2

Phase 2



Phase 2

Phase 2

E1

cl: standard-generic-function

cl: standard-class

E2

cl: class-name

sicl-clos: generic-function-lambda-list

E3

Load files for initialization
of classes and generic functions
as defined by AMOP.

Phase 2

E1

cl:
standard-
generic-
function

cl:
standard-
class

....

E2

cl:
class-
name

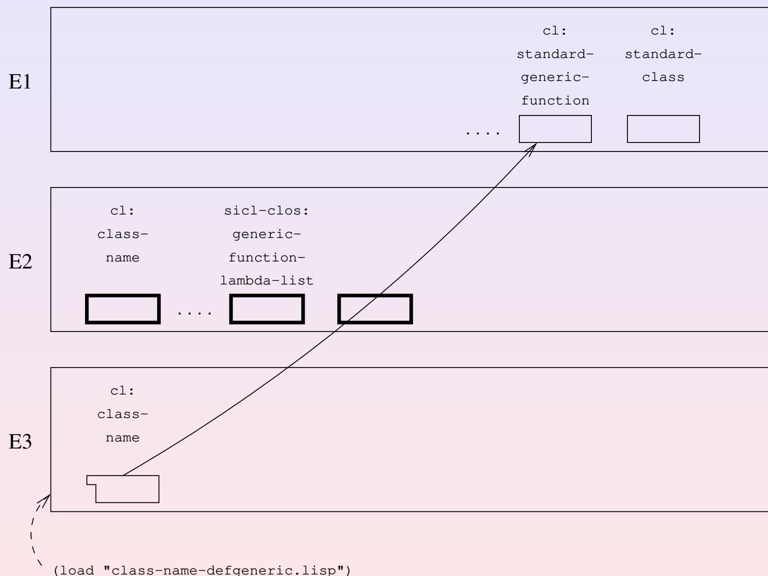
sicl-clos:
generic-
function-
lambda-list

....

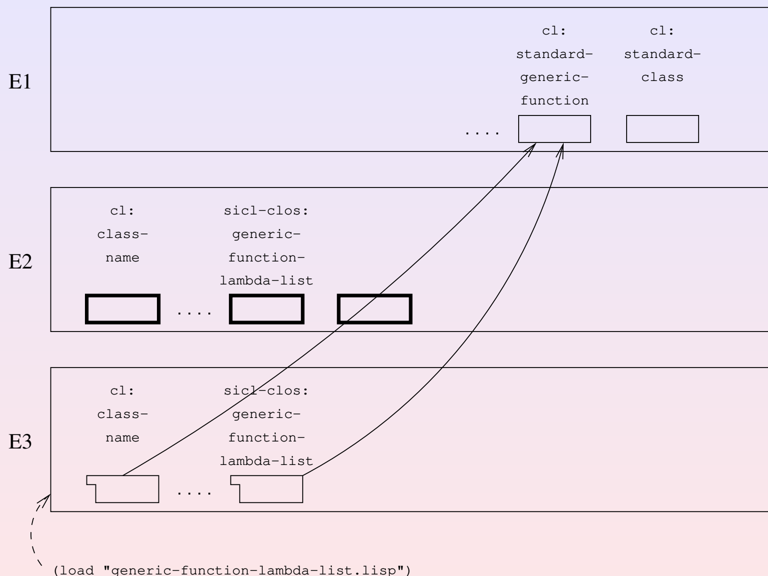
E3

```
(defun cl:ensure-generic-function ...)
(load "defgeneric-defmacro.lisp")
```

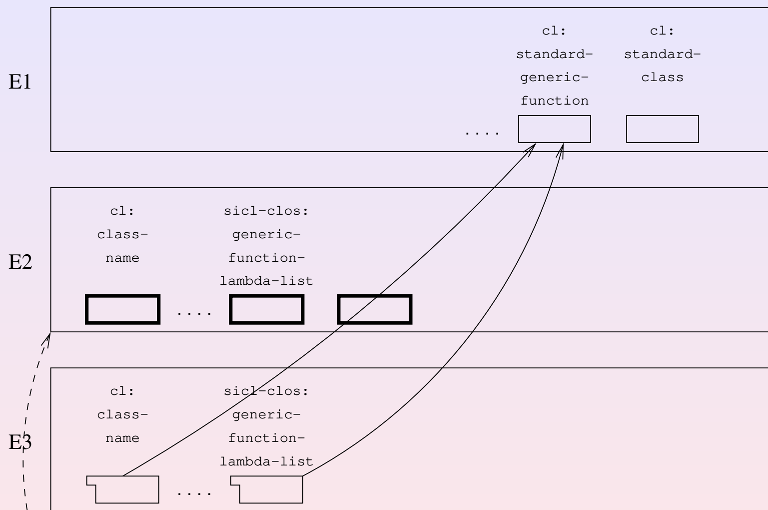
Phase 2



Phase 2

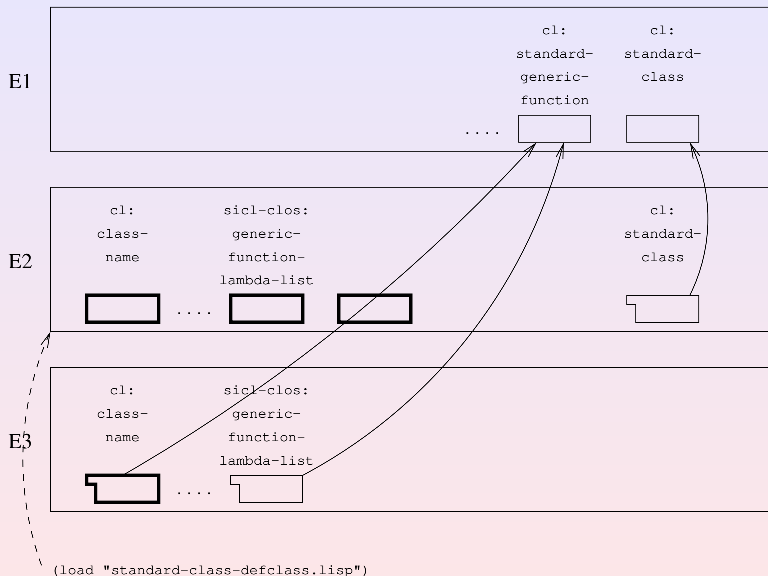


Phase 2

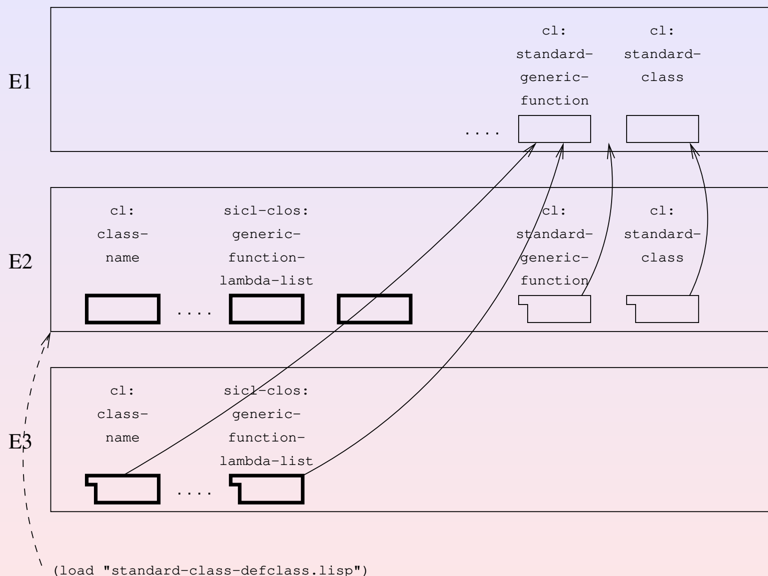



```
(defun ensure-class ...)
(load "defclass-defmacro.lisp")
```

Phase 2

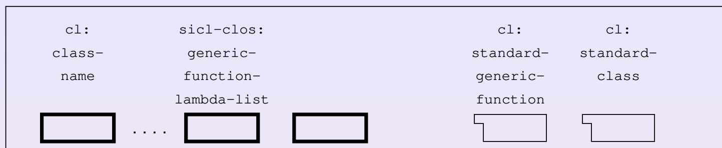


Phase 2

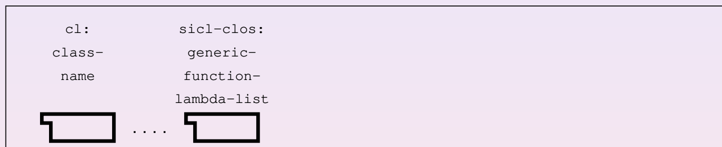


Phase 2

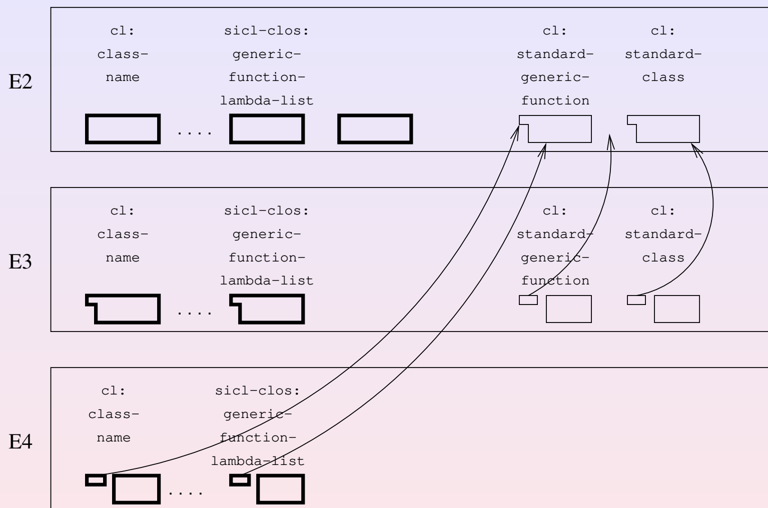
E2



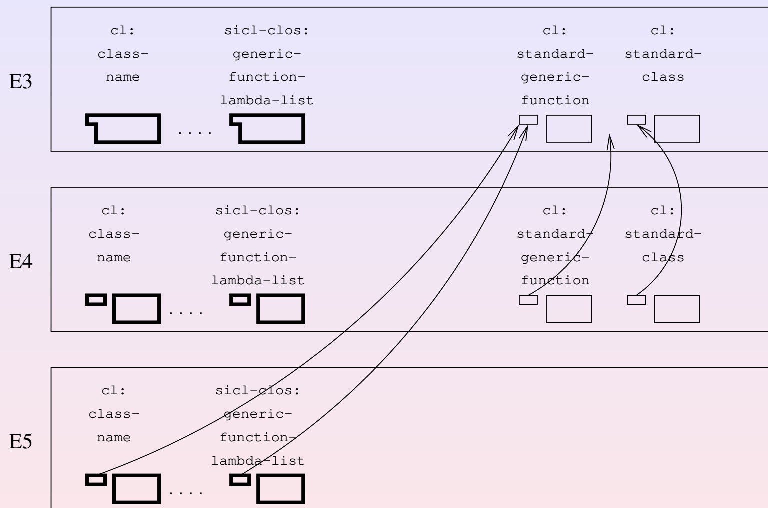
E3



Phase 3



Phase 4

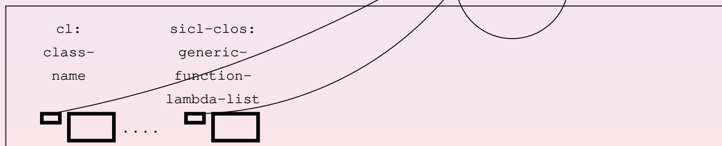


Phase 5

E3

E4

E5



Living with circularity

Appendix C of AMOP mentions two categories of circularity issues:

1. Bootstrapping issues.
2. Metastability issues.

Bootstrapping issues

Two issues are mentioned:

1. `standard-class` must exist before any metaobject can be created (including `standard-class`).
2. Classes are needed to create generic functions, and generic-functions are needed to create classes.

With our technique, these problems disappear.

Metastability issues

Two issues are mentioned:

1. `slot-value` needs to call `slot-value` of the class, to retrieve a list of slots.
2. `compute-discriminating-function` can not be used to compute a discriminating function of `compute-discriminating-function`.

We do not have the first problem, because slot accessors do not call `slot-value`. They access the slot directly using `standard-instance-access`.

We do not have the second problem, because the call cache of `compute-discriminating-function` is loaded during bootstrapping, and works for specified classes.

Future work

- Provide a code generator that produces native code.
- Produce a native object graph from the representation in the host. Special attention is needed for objects represented as host objects, such as symbols.
- Write code for the interface to the operating system, such as code for input/output, memory allocation, etc.

Acknowledgments

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

Thank you

Questions?