SICL

Building blocks for creators of Common Lisp implementations.

Robert Strandh

2013

Contents

Chapter 1

Introduction

SICL (which doesn't mean anything in particular; pronounce it like "sickle") is a project with the purpose of creating a collection of highly-portable high-performance "modules" for developers of Common Lisp systems. Such modules include "standard libraries" such as high-level functions that operate on lists, high-level functions that operate on sequences, the `format` function, the `loop` macro, the `read` function, the *pretty printer*, etc. Other planned modules include a module that provides localized docstrings for all of Common Lisp.

However, the modules are not limited to functionality that is directly provided by the Common Lisp language specification. An existing module defines a CLOS protocol for managing first-class global environments, and another defines such a protocol for managing the contents of the compilation environment during the early phases of compilation.

Furthermore, even though these modules are part of the SICL *project*, they may not be present in the SICL *repository*. Whenever a module is sufficiently independent of the rest of the SICL code base, we try to extract it into a separate, independently maintained repository. A module for reading Common Lisp source code as a *concrete syntax tree* and a highly programmable version of the Common Lisp `read` function have already been extracted this way.

Initially, we planned to decrease the interdependence of modules as much as possible by creating a partial order between the modules, thereby enabling an

implementation of a small subset of Common Lisp to become a fully compliant implementation by adding these modules in some order. This goal turned out to be unreasonable in that many modules would have to be written in a subset of the full language, thereby making them less maintainable. Instead, we think the most reasonable strategy for creating a new Common Lisp implementation is to write it using the full language, and to bootstrap the new implementation on an existing fully compliant implementation.

We think it is important that the code of SICL be of very high quality. To that end, error messages should be as explicit as possible. Macros, for instance, do extensive syntax analysis so as to prevent error messages from being phrased in terms of expanded code.

To gain wide acceptance, SICL is distributed according to a two-clause BSD license.

We also plan to use this collection of modules, together with additional specific code, in order to produce a complete implementation of Common Lisp.

We think it is possible to improve on existing open-source Common Lisp systems in several ways, and we hope SICL will be able to accomplish that, provided that great care is taken to create code with a combination of characteristics:

- The code is layered, so that different Common Lisp implementations can choose to include SICL modules that represent gaps in their system or improvement on their existing code, without having to include parts for which they prefer their own, implementation-specific code.

  Upper layers contain code that is not performance critical. This code is written entirely in Common Lisp. To avoid circular references, we specify what lower-level Common Lisp primitives can be used to write functions in the upper layer. If done well, code in this layer could be used by all free Common Lisp implementations, which could save considerable maintenance effort. Examples of functionality in this layer would be formatted output, pretty-printing, and macros that can expand to portable Common Lisp code.

  Intermediate layers contain code that needs careful tuning to obtain performance, but where the tuning can be handled by writing different ver-

sions of the code for different cases. For instance, functions that work on all kinds of sequences might have special versions for lists and vectors. Similarly, such functions might have special versions for common values of the `:test` (such as the Common Lisp functions `#'eq`, `#'eql`, etc.) and `:key` arguments (such as `#'identity`, `#'car`, etc). These special cases should be handled by using compiler macros.

Lower layers have to rely more and more on implementation-specific details, and require the introduction of implementation-specific primitives to be used in implementations of standard Common Lisp constructs. We might provide several different versions of code in this layer for different low-level assumptions.

- The goal is for the code itself to be of very high quality. By this, we do not only mean that it should be bug-free as much as possible, but also that it should have good documentation strings and clear comments when required. We want error messages to be as explicit as possible, and to accomplish that we try to capture as many exceptional situations as is possible without performance penalties. We use very specific conditions that are subclasses of ones stipulated by the Common Lisp HyperSpec for condition signaling, so as to allow for implementations to take advantage of such specific conditions for better error reporting. Macro expansion code should do everything possible to capture as many errors as possible so that error reporting can be done in terms of code written by the programmer, as opposed to in terms of expanded code.

Part I

Portable modules

Chapter 2

Reader

This entire chapter is obsolete. We intend to use the implementation-independent reader Eclector, available at https://github.com/robert-strandh/Eclector.

Chapter 3

Printer

A large part of the printer can be written portably without performance penalty. We intend to supply standard methods for `print-object`, and code for functions such as `princ`, `prin1`, and `print`.

Initially, we were planning to use the method created by Burger and Dybvig [BD96] to print floating-point numbers so that they can be read back to the exact same number. But more recent work [Ada18] claims even better techniques. We need to compare these new techniques.

Chapter 4

Pretty printer

Richard C Waters designed the pretty printer for Common Lisp ([Wat89], [Wat92]). It may be possible to use it directly for SICL. Certainly, the existence of XP (i.e. what Waters designed) makes it low priority for SICL to have its own.

As I recall, there are some minor differences between XP and what was ultimately incorporated in the standard, but a long time has passed since I looked at it.

I also do not recall the license according to which XP is distributed. It may or may not be compatible with that of SICL. Finally, it may be that XP does not conform to the SICL coding standards.

Chapter 5

The `format` function

The directory `Code/Format` contains a near-complete implementation of the `format` function. What is missing is the printers for floating-point numbers. Also, the `formatter` function is not yet implemented.

We are planning to use the method created by Adams [Ada18] for printing floating-point numbers.

Chapter 6

The `loop` macro

Our `loop` module uses all available Common Lisp functions for its analysis of syntax and semantics. We believe this is not a problem, even though we assume the existence of `loop` for many other modules, because the code in this module will be executed during macro-expansion time, and for a new Common Lisp system, it would be executed during cross compilation by another full Common Lisp implementation.

Our `loop` module uses only standard Common Lisp code in its resulting expanded code, so that macro-expanded uses of `loop` will not require any other SICL module in order to work.

The code for the SICL `loop` macro is located in the directory `Code/Loop`.

For parsing a `loop` expression, we use a technique called *combinatory parsing*, except that we do not handle arbitrary backtracking. Luckily, arbitrary backtracking is not required to parse the fairly simple syntax of `loop` clauses.

6.1 Current state

All `loop` clauses have been tested with the test cases provided by Paul Dietz' ANSI Common Lisp test suite.

Future work includes providing an alternative parser to be used when the normal parser fails. The purpose of the alternative parser is to provide good error messages to the programmer.

6.2 Protocol

6.2.1 Package

The symbols documented in this section, and that are not in the package `common-lisp`, are defined in the package named `sicl-loop`.

6.2.2 Classes

⇒ `clause` [*Class*]

This class is the base class for all clauses.

⇒ `subclauses-mixin` [*Class*]

This class is a superclass of all classes of clauses that accept the `and loop` keyword.

⇒ `var-and-type-spec-mixin` [*Class*]

This class is a superclass of all classes of clauses and subclauses that take a `var-spec` and a `type-spec`.

⇒ `compound-forms-mixin` [*Class*]

This class is a superclass of all classes of clauses that take a list of compound forms.

⇒ `loop-return-clause-mixin` [*Class*]

This class is a superclass of all classes of clauses that can make the loop return a value.

6.2.3 Functions

⇒ `bound-variables` *clause* [*Generic Function*]

The purpose of this generic function is to generate a list of all bound variables in a clause. The same variable occurs as many times in the list as the number of times it is bound in the clause.

⇒ `accumulation-variables` *clause* [*Generic Function*]

The purpose of this generic function is to generate a list of all the accumulation variables in a clause. Each element of the list is itself a list of three elements. The first element is the name of a variable used in an `into` clause, or `nil` if the clause has no `into`. The second element determines the kind of accumulation, and can be one of the symbols `list`, `count/sum`, or `max/min`. The third element is a type specifier which can be `t`.

⇒ `declarations` *clause* [*Generic Function*]

The purpose of this generic function is to extract a list of declaration specifiers from the clause. Notice that it is a list of declaration specifiers, not a list of declarations. In other words, the symbol `declare` is omitted.

⇒ `initial-bindings` *clause* [*Generic Function*]

The purpose of this generic function is to extract the outermost level of bindings required by the clause.

⇒ `final-bindings` *clause* [*Generic Function*]

The purpose of this generic function is to extract the innermost level of bindings required by the clause.

⇒ `bindings` *clause* [*Generic Function*]

The default method of this generic function appends the result of calling `initial-bindings` and that of calling `final-bindings`.

Chapter 7

High-level functions on lists

This module is meant to be a complete implementation of portable functions and macros in the Conses dictionary (chapter 14 in the HyperSpec), except for the low-level functions such as `cons`, `car`, `cdr`, `rplaca`, and `rplacd` which can not be implemented portably. For its implementation, it uses the `loop` macro. If any other functionality is required, it will supply special implementations of such functionality, so as to avoid dependencies on other modules.

We obtain high performance by identifying important special cases such as the use of `:test` function `eq`, or `equal`, or the use of a `:key` of `identity`.

We supply compiler macros so as to avoid runtime dispatch whenever a special-case function can be determined by only looking at the call site. This ensures high performance for short lists, where argument parsing would otherwise represent a significant fraction of the cost of the call.

We are considering whether it might be worthwhile to supply a macroexpanded version of this module so that no existing implementation of the `loop` macro would be required.

This module is fairly complete, and it includes macros such as `push` and `pop` as well as compiler macros for functions that take keyword arguments such as the mapping functions.

The module also includes definitions of specific conditions that are used by this

module, together with condition reporters in English for those conditions. It also includes English-language documentation strings for some of the functions.

Chapter 8

Sequence functions

This module provides high-performance implementations of the functions in the "sequences" chapter of the HyperSpec. High performance will be obtained by identifying important special cases such as the use of `:test` function `eq`, or `equal`, or the use of a `:key` of `identity`.

8.1 Current state

In an attempt to make the functions as fast as possible, we created lots of different specialized versions for common cases of keyword arguments. Preliminary tests show that we improve on the speed compared to existing commonly used implementations of Common Lisp. However, the existence of all those special cases also makes the module hard to read and difficult to test.

For that reason, we plan to rewrite this module, using an alternative strategy.

8.2 Future work

Work is in progress using the techniques in our paper at ELS 2017 [DS17]. These techniques work well for most of the sequence functions such as `find`, `position`, etc.

Concerning the *sorting functions* (i.e., `sort` and `stable-sort`) there is an interesting challenge with respect to finding a stable sorting algorithm for vectors that uses little extra space. The naive version of mergesort uses $O(n)$ extra space, but some research ([HL90], [HL88], [KPT96]) indicates that it is possible to obtain an in-place stable version of mergesort. Since mergesort is typically significantly faster than quicksort, this would be a worthwhile direction to pursue.

Chapter 9

Hash tables

9.1 Package

The package for all symbols in this chapter is `sicl-hash-table`.

9.2 Protocol

Every class in this section has `built-in-class` as its metaclass.

⇒ `hash-table` [*Class*]

This class is the base class of all hash tables. It is a subclass of the class `t`.

⇒ `hash-table-p` *hash-table* [*Generic Function*]

This generic function returns a generalized Boolean value, where a *true* value indicates that *hash-table* is an instance of the class `hash-table`, and *false* indicates that *hash-table* is *not* an instance of the class `hash-table`.

⇒ `hash-table-count` *hash-table* [*Generic Function*]

This generic function returns a non-negative integer indicating the number of entries in *hash-table*.

⇒ `gethash` *key hash-table* `&optional` *default* [*Generic Function*]

This generic function returns two values. The first value is the value in *hash-table* associated with *key*, or `nil` if no value in *hash-table* is associated with *key*. The second value is a generalized Boolean value, where a *true* value indicates that the first value is indeed present in *hash-table*, and *false* indicates that no value is associated with *key* in *hash-table*.

⇒ `(setf gethash)` *new-value key hash-table* `&optional` *default* [*Generic Function*]

⇒ `hash-table-test` *hash-table* [*Generic Function*]

⇒ `remhash` *key hash-table* [*Generic Function*]

This generic function removes the entry associated with *key* in *hash-table*. It returns a *true* value an entry associated with *key* existed and was removed. Otherwise, it returns *false*.

9.3 Implementation

⇒ `eq-hash-table-mixin` [*Class*]

This mixin class is a superclass of every hash tables class that uses `eq` as its test function. It is a subclass of the class `t`.

⇒ `hash-table-test` *(hash-table* `eq-hash-table-mixin`*)* [*Method*]

This method returns the symbol `eq`.

⇒ `eql-hash-table-mixin` [*Class*]

This mixin class is a superclass of every hash tables class that uses `eql` as its test function. It is a subclass of the class `t`.

⇒ `hash-table-test` *(hash-table* `eql-hash-table-mixin`*)* [*Method*]

This method returns the symbol `eql`.

⇒ `equal-hash-table-mixin` [*Class*]

This mixin class is a superclass of every hash tables class that uses `equal` as its test function. It is a subclass of the class `t`.

⇒ `hash-table-test` *(hash-table* `equal-hash-table-mixin`*)* [*Method*]

This method returns the symbol `equal`.

⇒ `equalp-hash-table-mixin` [*Class*]

This mixin class is a superclass of every hash tables class that uses `equalp` as its test function. It is a subclass of the class `t`.

⇒ `hash-table-test` *(hash-table* `equalp-hash-table-mixin`*)* [*Method*]

This method returns the symbol `equalp`.

⇒ `standard-hash-table` [*Class*]

This class is a subclass of the class `hash-table`.

⇒ `:contents` [*Initarg*]

This initialization argument is accepted by all instances of `standard-hash-table`

⇒ `contents` *standard-hash-table* [*Generic Function*]

Given an instance of the class `standard-hash-table`, this generic function returns the value that was supplied as the `:contents` initialization argument when the instance was created.

9.3.1 Hash table implemented as a list of entries

⇒ `list-hash-table` [*Class*]

This class is a subclass of the class `standard-hash-table`. It provides and implementation of the protocol where the entries are stored as an association list where the key is the `car` of the element in the list and the value is the `cdr` of the element in the list.

⇒ `gethash` *key (hash-table* `list-hash-table`*)* `&optional` *default* [*Method*]

This method calls the generic function `contents` with *hash-table* as an argument to obtain a list of entries of `hash-table`. It also calls the generic function `hash-table-test` with *hash-table* as an argument to obtain a function to be used to compare the keys of the entries to `key`. It then calls the standard Common Lispfunction `assoc`, passing it *key*, the list of entries, and the test function as the value of the keyword argument `:test`. If the call returns a non-`nil` value (i.e. a valid entry), then the method returns two values, the `cdr` of that entry and `t`. Otherwise, the method return `nil` and `nil`.

Chapter 10

Type declarations of standard Common Lisp functions

This module contains portable type declarations for all standard Common Lisp functions. It could be used by implementers of Common Lisp compilers to accomplish error checking and type inferencing.

Chapter 11

Documentation strings for all Common Lisp symbols

As mentioned elsewhere, we believe that documentation strings should be separate from code, because they do not address the same audience. In addition, separating the two allows us to distribute the documentation strings as a separate module. Many implementations have substandard documentation strings, so this is an important module that can be used as a drop-in replacement for existing ones.

We will provide the infrastructure for allowing internationalization of documentation strings, but we probably will not provide different versions for different languages.

Chapter 12

Condition system

This module is located in the `Code/Conditions` directory. It it meant to contain the complete Common Lisp condition system. The module is fairly complete. There are three main items in this module:

- A complete portable implementation of the Common Lisp condition system signaling mechanism.
- Definitions of standard Common Lisp conditions.
- Definitions of additional SICL conditions.

As it turns out, the Common Lisp condition signaling mechanism can be implemented in a completely portable way, using only special variables as the underlying mechanism. In addition, this implementation method is likely to exhibit acceptable performance, because condition signaling is assumed to be fairly infrequent anyway.

The module assumes the existence of the `define-condition` macro. The full hierarchy of Common Lisp conditions is implemented, and there is a file containing condition reporters for those conditions when the English language is used.

The additional conditions are used by various SICL modules. At the moment, this module does not contain all additional conditions used by the system.

We are still undecided concerning whether all additional conditions should be concentrated in one place, or on the contrary, they should be distributed with the particular module that uses them. The first solution favors sharing of conditions that are usable by several modules, and the second solution lets the user of a module install only the conditions used by that module, without having to install a large number of conditions that will never be used.

Chapter 13

Arithmetic

The code for arithmetic functions is contained in the directory `Code/Arithmetic`. At the moment, this code is very preliminary. There is not even a package definition for the package used in the file `arithemtic.lisp`.

The code contains definitions of the functions `+`, `-`, `*`, and `/`. These functions call binary versions also defined in the same file. There are also compiler macros to turn calls to these functions with a known number of arguments into calls to the binary versions.

The binary versions of the arithmetic functions dispatch according to the exact type of the arguments to binary versions of the function with a specific combination of types, but those specific functions have not yet been written. Some of those functions can be written using portable code, but most of them will probably use machine-specific low-level instructions.

We have no specific advice for anyone who might be interested in working on code for arithmetic functions.

Chapter 14

Array

There are a few things that can be done in a portable module for arrays.

Clearly, `array-dimensions` can be defined in terms of `array-rank` and `array-dimension`. On the other hand, it can also be done the other way around, i.e., `array-rank` can be computed as the length of the list returned by `array-dimensions`, and `array-dimension` can be defined using the list returned by `array-dimensions` and `elt`, or `nth`. Whether one or the other is chosen depends on how arrays are represented. In SICL the list of dimensions is already stored in the array (See Section 15.9.2.), so the second solution is better, and it also avoids consing.

`array-total-size` can be defined in terms of `array-rank` and `array-dimension`, or in terms of and `array-dimensions` depending on what solution was chosen above.

Furthermore, `aref` and `(setf aref)` can be defined in terms of `row-major-aref` and `(setf row-major-aref)`.

It might be worthwhile defining a compiler macro for `aref`.

At the moment, there is some experimental code for arrays in the directory `Code/Array`, but it is probably best to do it over from scratch.

Part II

System-specific modules

Chapter 15

Data representation

15.1 Low-level tag bits

The three least significant bits of a machine word are used to represent four different *tags* as follows:

- `000`, `010`, `100`, `110`. These tags are used for fixnums. The bits except the last one represent integer in two's complement representation. On a 64-bit machine, a fixnum is thus in the interval $[2^{-62}, 2^{62} - 1]$.
- `001`. This tag is used for `cons` cells. A pointer to a `cons` cell is thus a pointer aligned to a double word to which the machine integer 1 has been added. See Section 15.3 for more information about the representation of `cons` cells.
- `011`. This tag is used for various *immediate* objects, and in particular for *characters*. (See Section 15.2.)
- `101`. This tag is used for all heap-allocated Common Lisp objects other than `cons` cells. A heap allocated object like this is a *standard object*. It is represented by a two-word header object with one word containing a tagged pointer to the *class object* and the other word containing a tagged pointer to the *rack*. See Section 15.4 for more information about the representation of standard objects.

- `111`. This tag is used to tag a pointer to a *rack*. Notice that a rack is not a first-class Common Lisp object, so it can not be the value of any variable, slot, or argument.

On a 64-bit machine, floats of type `short-float` and `single-float` are represented as immediate values.

15.2 Immediate objects

Immediate objects are all objects with `011` in the lower three bits. Two more bits are used to distinguish between different kinds of immediate objects, giving the following five low bits:

- `00011`. This tag is used for Unicode characters. When shifted five positions to the right, the value gives the Unicode code point.
- `01011`. This tag is used short floats.
- `10011`. This tag is used for single floats (64-bit platforms only).
- `11011`. This tag is unused.

15.2.1 Characters

As indicated above, the low five bits of a character have the value `00011`, and the corresponding Unicode code point is obtained by shifting the value of the character five positions to the right.

We currently do not plan to supply a module for Unicode support. Instead we are relying on the support available in the Unicode library by Edi Weitz.

15.2.2 Single floats

On a 64-bit platform, a single float corresponds to a single-precision IEEE floating-point value. The value is stored in the most-significant half of the word.

15.3 Representation of cons cells

A cons cell is represented as two consecutive machine words aligned on a double-word boundary.

15.4 Representation of standard objects

Recall that a *standard object* is a heap allocated object that is not a cons cell. All standard objects are represented in (at least) two parts, a *header object* and a *rack*. The header object always consists of two consecutive words aligned on a double-word boundary (just like cons cells). The first word always contains a tagged pointer to a *class* object (which is another standard object). The second word contains a tagged pointer (with tag 111) to the first word of the rack.

The first entry of the rack of every standard object is a small fixnum called the *stamp* of the standard object. The stamp is the *unique class number* of the class of the general instance as it was when the instance was created. The main purpose of this information is to be used in *generic function dispatch*. It is also used to determine whether a standard object is an obsolete instance (in this case the stamp of the standard object will not be the same as the *current* unique class number of the class of the standard object). Standard objects that can become obsolete are said to be *flexible*.

One advantage of representing standard objects this way is that the rack is *internally consistent*. To explain what we mean by this concept, let us take an *adjustable array* as an example. The implementation of aref must check that the indices are valid, compute the offset of the element and then access the element. But in the presence of threads, between the index check and the access, some other thread might have adjusted the array so that the indices are no longer valid. In most implementations, to ensure that aref is *thread safe*, it is necessary to prevent other threads from intervening between the index check and the access, for instance by using a *lock*. In SICL, adjusting the array involves creating a new rack with new dimensions, and then with a single store instruction associate the new rack with the array. The implementation of aref would then initially obtain a pointer to the rack and then do the index check,

the computation of the offset, and the access without risking any interference. No locking is therefore required. Another example is a *generic function*. When a method is added or deleted, or when a new sequence of argument classes is seen, the generic function must be destructively updated. Normally, this operation would require some locking primitive in order to prevent other threads from invoking a partially updated generic function. In SICL, to update a generic function this way, a new rack would be allocated and the modifications would be made there, leaving the original generic function intact until the final instruction to store a reference to the the new rack in the header object.

A standard object can be *rigid* or *flexible*. A rigid instance is an instance of a class that can not change, typically a system class. A flexible instance is an instance of a class that may be modified, makings its instances *obsolete*. In SICL, structure objects are flexible too.

In the following sections we give the details of the representation for all possible standard objects.

15.5 Flexible instances

A flexible instance must allow for its class to be redefined, making it necessary for the instance to be updated before being used again. The standard specifically allows for these updates to be delayed and not happen as a direct result of the class redefinition. They must happen before an attempt is made to access some slot (or some other information about the slots) of the instance. It is undesirable to make all instances directly accessible from the class, because such a solution would waste space and would have to make sure that memory leaks are avoided. We must thus take into account the presence of *obsolete instance* in the system, i.e. instances that must be *updated* at some later point in time.

The solution is to store some kind of *version* information in the rack so that when an attempt is made to access an obsolete instance, the instance can first be updated to correspond to the current definition of its class. This version information must allow the system to determine whether any slots have been added or removed since the instance was created. Furthermore, if the garbage collector traces an obsolete instance, then it must either first update it, or the version information must allow the garbage collector to trace the obsolete

version of the instance. Our solution allows both. We simply store a reference to the *list of effective slots* that the class of the instance defined when the instance was created. This reference is stored as the *second* word of the rack (recall that the first word is taken up by the *stamp*).

This solution makes it possible to determine the layout of the rack of an obsolete instance, so that it can be traced by the garbage collector when necessary. This solution also allows the system to determine which slots have been added and which slots have been removed since the instance was created. In order to detect whether an object is obsolete, the contents of the first word of the rack (i.e. the *stamp*) is compared to the *class unique number* of the class of the object. However, this test is performed automatically in most cases, because when an obsolete object is passed as an argument to a generic function, the *automation* of the discriminating function of the generic function will fail to find an effective method, triggering an update of the object. (See Section 17.2.2.)

15.6 Funcallable standard objects

By definition, a *funcallable standard object* is an instance of a subclass of the class `funcallable-standard-object` which is itself a subclass of the class `standard-object` and of the class `function`. (See Section 15.9.7.)

To make function invocation fast, we want every subclass of the class `function` to be invoked in the same way, i.e. by loading the *static environment* into a register and then transferring control to the *entry point* of the function. The static environment and the entry point are stored in the function object and are loaded into registers as part of the function-call protocol.

When the funcallable standard object is a generic function, invoking it amounts to transferring control to the *discriminating function*. However, the discriminating function can not *be* the generic function, because the CLOS specification requires that the discriminating function of a generic function can be replaced, without changing the identity of the generic function itself. Furthermore, the discriminating function does not have to be stored in a slot of the generic function, because once it is computed and installed, it is no longer needed. In order for the generic function itself to behave in exactly the same way as its discriminating function, whenever a new discriminating function is *installed*,

the *entry point* and the *static environment* are copied from the discriminating function to the corresponding slots of the generic function itself.

15.7 Code objects

A *code object* is a standard object that represents the *code* of a function. Recall that a function consists of some code and an environment. The code object is common to all functions that share the same code.

The tables described below that are meant for the garbage collector have entries only for the values of the program counter corresponding to function calls. At every *safe point* the thread tests the flag `gc-requested` described in Section 23.3. If that flag is set, a function call is made to the local garbage collector. Therefore, every safe point corresponds to a function call, and thus the information is needed by the garbage collector only at function calls.

A code object contains the following information:

- The executable native instructions in the form of a vector of unsigned bytes.
- Tables providing support for commands of the debugger. See below for more detailed information about these tables.
- A hash table mapping a value of the program counter to a *frame map*. A frame map is a bitmap containing information that is used by the garbage collector to determine which stack locations contain Common Lisp objects that should be traced. Stack locations containing data owned by the current function invocation are present in the table. A frame map never contains a stack location used to store a callee-saves register, because whether such a location contains a Common Lisp object or some other datum is determined by the caller of the current function. The compiler may omit stack locations that are known to contain immediate Common Lisp objects that the garbage collector does not have to trace. For a typical backend, index i of the bitmap represents the stack location at address $b - i$ where b is the value of the base pointer.

- A hash table mapping a value of the program counter to a *callee-saves register map*. This map is a bitmap which has as many elements as there are callee-saves registers. A bit in the bitmap is set if the corresponding callee-saves register contains a Common Lisp object that may need to be traced by the garbage collector. The compiler may put a cleared bit in the bitmap for a register known to contain an immediate Common Lisp object that the garbage collector does not need to trace.

- A hash table mapping a value of the program counter to a *callee-saves stack map*. This map is indexed by a callee-saved register and contains a stack location in which the contents of the saved register was stored before the current function could use the register. For a typical backend such as x86-64, the entries in the table represent the registers `RBX` (index 0), `R12` (index 1), `R13` (index 2), `R14` (index 3), and `R15` (index 4) as described in Section 27.1. The value of an entry represents the value to subtract from the value of the base pointer in order to get the stack location of the saved register. A value of 0 indicates that the register is not used by this function at this program point, i.e. either current function did not save the register, or it restored the register after having finished using it,

- A vector of strings representing the source code of the compilation unit of which this code object is part. Each string corresponds to a line of code.

- Information about the file that contains the source code of the compilation unit of which this code object is part.

- The *concrete syntax tree* corresponding to the source code of the compilation unit of which this code object is part.

The following tables are used for debugger support:

- A table mapping source locations (i.e. line and column information corresponding to the location in the source code of the expression being evaluated) to values of the program counter. When the user of the debugger sets a breakpoint at a particular source location, the debugger uses this information to determine a value of the program counter at which the program should stop its execution.

- A table mapping values of the program counter to *variable liveness information.*

15.8 Rigid instances

Contrary to flexible instances, a *rigid instance* is an instance of a class that is not allowed to change after the first instance is created. Some system classes are examples of such classes. The class definition might change as long as there are no instances, but the consequences are undefined if a system class is changed after it has been instantiated.

15.9 Instances of built-in classes

The only direct instances of built-in classes are fixnums, characters, short floats, single floats, and `cons` cells.

15.9.1 Instances of `sequence`

The system class `sequence` can not be directly instantiated. Instead, it serves as a superclass for the classes `list` and `vector`.

The HyperSpec is a bit contradictory here, because in some places it says that `list` and `vector` represent an exhaustive partition of `sequence`[1] but in other places it explicitly allows for other subtypes of `sequence`.[2] The general consensus seems to be that other subtypes are allowed.

15.9.2 Arrays

Arrays are standard objects. As a consequence, the exact layout of the rack of an array is determined by the MOP machinery for computing the slots when the class is finalized. All arrays are simple.

[1] See for instance section 17.1

[2] See the definition of the system class `sequence`.

Every array class has a slot `dimensions` that contains a list of dimensions. The length of the list is the *rank* of the array.

The element-type is determined by the exact subclass of the `array` class. The elements follow the explicit slots in the rack. The size of the rack is rounded up to the nearest multiple of a word.

All arrays are *adjustable* thanks to the split representation with a header object and a rack. Adjusting the array typically requires allocating a new rack.

Specialized array classes with the following element types are provided:

- `double-float`
- `(unsigned-byte 64)`.
- `(signed-byte 64)`.
- `(unsigned-byte 32)`.
- `(signed-byte 32)`.
- `(unsigned-byte 8)`, used for code, interface with the operating system, etc.
- `character` (i.e. strings) as required by the HyperSpec.
- `bit`, as required by the HyperSpec.

Since the element type determines where an element is located and how to access it, `row-major-aref` and `(setf row-major-aref)` are *generic functions* that specialize on the type of the array.

System class `vector`

A vector is a one-dimensional array. As such, a vector has a rack where `dimensions` slot contains a proper list of a single element, namely the *length* of the vector represented as a fixnum.

All vectors have a slot for the *fill pointer*. This slot contains `nil` for vectors without a fill pointer.

System class `string`

Tentatively, we think that there is no need to optimize strings that contain only characters that could be represented in a single byte.

15.9.3 Symbols

A symbol is a standard object. It has slots containing the following data:

1. The *name* of the symbol. The value of this slot is a string.
2. The *package* of the symbol. The value of this slot is a package or `NIL` if this symbol does not have a package.

Notice that the symbol does not contain its *value* as a global variable, nor does it contain its definition as a *function* in the global environment. Instead, this information is contained in an explicit *global environment* object.

Notice also that the symbol does not contain the *property list* associated with it. This information is also kept separately in an explicit *global environment* object.

See Section 16.1 for more information on global environments.

15.9.4 Packages

A package is a standard object with the following slots:

1. The *name* of the package. The value of this slot is a string.
2. The *nicknames* of the package. The value of this slot is a list of strings.
3. The *use list* of the package. The value of this slot is a proper list of packages that are used by this package.
4. The *used-by list* of the package. The value of this slot is a proper list of packages that use this package.

5. The *external symbols* of the package. The value of this slot is a proper list of symbols that are both present in and exported from this package.

6. The *internal symbols* of the package. The value of this slot is a proper list of symbols that are present in the package but that are not exported.

7. The *shadowing symbols* of the package. The value of this slot is a proper list of symbols.

15.9.5 Hash tables

15.9.6 Streams

15.9.7 Functions

Ordinary (non-generic) SICL functions are instances of the class named `simple-function`. The class `simple-function` is a direct subclass of `funcallable-standard-object`. These two symbols both have the package `sicl-clos` as their home package.

In order to obtain reasonable performance, we represent functions in a somewhat complex way, as illustrated by Figure 15.1.

Figure 15.1 shows two functions. The two functions were created from the same compilation unit, because they share the same code object. (See Section 15.7.)

A function is represented as a two-word header (as usual) and a rack with three slots:

1. The obligatory *stamp*.

2. An *environment*, which is the lexical environment in which the function was defined. We are not giving the details of how the static environment is represented here.

3. The *entry point*. The entry point is a *raw address* of an aligned word in the vector containing the instructions of the function. Functions are always allocated in the global heap, so the code vector never moves. Therefore, this address will never need to be updated.

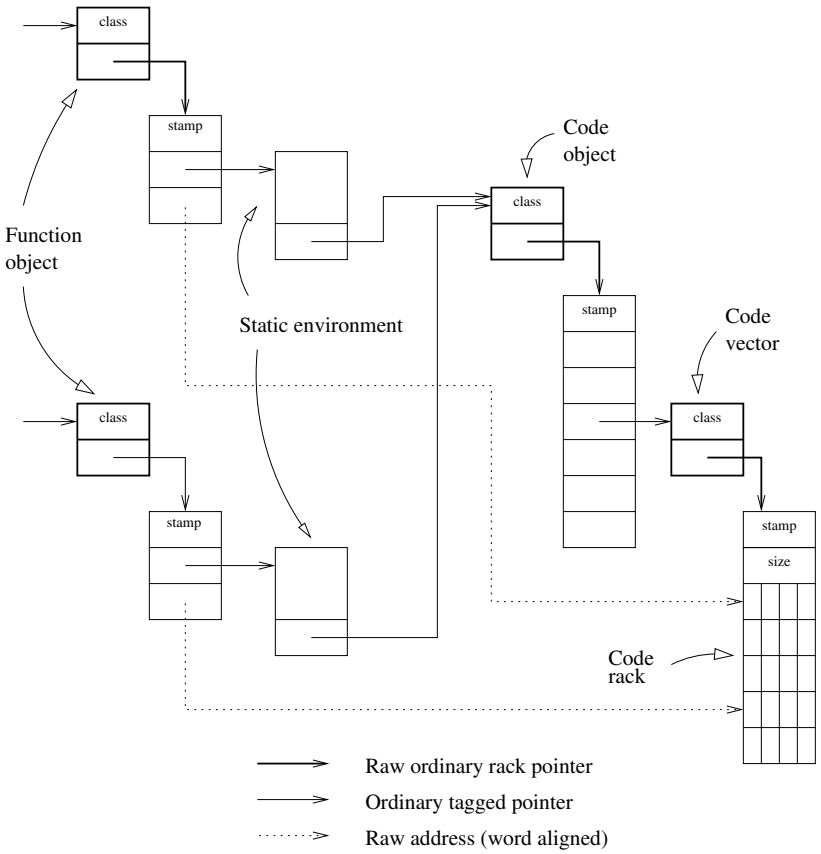

Figure 15.1: Representation of functions.

The static environment contains the *code object* as one of its elements.

Since raw addresses are word aligned, they show up as `fixnum`s when inspected by tools that are unaware of their special signification.

For a *generic function*, the description of the slots above applies both to the generic function object itself and to the *discriminating function* of the generic function. In addition to these slots, a generic function also contains other slots holding the list of its methods, and other information.

When a function is called, there are several possible situations that can occur:

- The most general case is when an object of unknown type is given as an argument to `funcall`. Then no optimization is attempted, and `funcall` is responsible for determining whether the object is a function, a name that designates a function, or an object that does not designate a function in which case an error is signaled.

- When it can be determined statically that the object called is a function (i.e. its class is a subclass of the class `function`), but nothing else is know about it, then an *external call* is made. Such a call consists of copying the contents of the *static environment* slot to the predetermined place specific to the backend, and then to issue a *call* instruction (or equivalent) to the address indicated by the *entry point* slot of the function object. When a function is called using a name known at compile time, then the object is known to be a function, though it may be a function that signals an error because the intended function is undefined.

- When it can be determined statically that the object being called is a function object in the *same compilation unit* as the caller, then we can make an *internal call*. If both the caller and the callee are *global functions* (so that the static environment contains only a pointer to the code object, then it suffices to issue a `call` instruction (or equivalent) to a relative address that can be determined statically. The relative address can be chosen so as to avoid type checking of arguments with known types. However, we may not take advantage of this possibility unless the `speed` optimize quality is high, because it makes it impossible to redefine a single function in a compilation unit.

Chapter 16

Environments

Common Lisp has a concept of *environments*, and in fact several different environments and several different *kinds* of environment are mentioned in the HyperSpec. However, Common Lisp does not mandate any particular representation of these environments, nor does it mention any particular *operations* on environments other than the implicit operations of defining functions, variables, macros, types, etc.

16.1 The global environment

In many Common Lisp systems the global environment is *spread out* in that it does not have an explicit definition as a data type. Parts of it, such as the set of packages or the set of classes, might be contained in global locations. Other parts of it may be stored in symbols such as the value or the function definition of a symbol. The standard specifically allows for this kind of spread-out representation.

In SICL, we prefer to have an explicit representation of the global environment as a data object. By doing it this way, we can allow for any number of global environments present in the system at any point in time. Different global environments can have a different set of packages, a different set of classes, a different set of types, etc. This representation can give us several interesting

advantages:

- We might ensure that there is always a *sane* environment present in case some environment gets destroyed (by a user accidentally removing some essential system function, for instance).
- We can allow for several different packages with the same name to exist in a system, as long as they are present in different environments, which would allow for simpler experimentation with different versions of packages.
- We can use different environments for bootstrapping purposes, avoiding conflicts with existing packages when a new system is built.
- By having the compiler work against such a first-class environment, we can bootstrap from any existing Common Lisp implementation. All that is needed is an implementation of the environment protocol for that implementation.
- We could even imagine a multi-user system based on different environments, and we could then allow users to do things such as defining `:after` methods on `print-object` that are private to that user.
- etc.

A global environment in SICL would then contain:

- A set of *packages*, represented either as a list or as a hash table mapping names to packages.
- A dictionary of *classes*, represented either as an association list or as a hash table mapping names to classes.
- A mapping from function names to entries representing functions, macros, compiler macros, and special operators.
- A mapping from names to entries representing type definitions.
- A mapping from names to entries representing *dynamic variables*.

- Values of *constant variables*.
- A set of *proclamations* concerning types of variables and functions, but also *autonomous* proclamations such as `optimize` and `declaration`.
- A mapping from names to *method combinations*.
- etc.

For a complete *protocol* that gives all the functionality provided by a global environment, see Section 16.2.

16.2 Global environment protocol

⇒ `environment` [*Class*]

This class is the base class for all first-class global environments.

⇒ `fboundp` *function-name environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `fboundp`.

It returns true if *function-name* has a definition in *environment* as an ordinary function, a generic function, a macro, or a special operator.

⇒ `fmakunbound` *function-name environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `fmakunbound`.

This function makes *function-name unbound* in the function namespace of *environment*.

If *function-name* already has a definition in *environment* as an ordinary function, as a generic function, as a macro, or as a special operator, then that definition is lost.

If *function-name* has a `setf` expander associated with it, then that `setf` expander is lost.

⇒ `special-operator` *function-name environment* [*Generic Function*]

If *function-name* has a definition as a special operator in *environment*, then that definition is returned. The definition is the object that was used as an argument to `(setf special-operator)`. The exact nature of this object is not specified, other than that it can not be `nil`. If *function-name* does not have a definition as a special operator in *environment*, then `nil` is returned.

⇒ `(setf special-operator)` *new function-name environment* [*Generic Function*]

Set the definition of *function-name* to be a special operator. The exact nature of *new* is not specified, except that a value of `nil` means that *function-name* no longer has a definition as a special operator in *environment*.

If a value other than `nil` is given for *new*, and *function-name* already has a definition as an ordinary function, as a generic function, or as a macro, then an error is signaled. As a consequence, if it is desirable for *function-name* to have a definition both as a special operator and as a macro, then the definition as a special operator should be set first.

⇒ `fdefinition` *function-name environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:fdefinition`.

If *function-name* has a definition in the function namespace of *environment* (i.e., if `fboundp` returns true), then a call to this function succeeds. Otherwise an error of type `undefined-function` is signaled.

If *function-name* is defined as an ordinary function or a generic function, then a call to this function returns the associated function object.

If *function-name* is defined as a macro, then a list of the form `(cl:macro-function` *function*`)` is returned, where *function* is the macro expansion function associated with the macro.

If *function-name* is defined as a special operator, then a list of the form `(cl:special` *object*`)` is returned, where the nature of *object* is currently not specified.

⇒ `(setf fdefinition)` *new-def function-name environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `(setf cl:fdefinition)`.

new-def must be an ordinary function or a generic function. If *function-name* already names a function or a macro, then the previous definition is lost. If *function-name* already names a special operator, then an error is signaled.

If *function-name* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

⇒ `macro-function` *symbol environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:macro-function`.

If *symbol* has a definition as a macro in *environment*, then the corresponding macro expansion function is returned.

If *symbol* has no definition in the function namespace of *environment*, or if the definition is not a macro, then this function returns `nil`.

⇒ `(setf macro-function)` *new-def symbol environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:macro-function)`.

new-def must be a macro expansion function or `nil`. A call to this function then always succeeds. A value of `nil` means that the *symbol* no longer has a macro function associated with it. If *symbol* already names a macro or a function, then the previous definition is lost. If *symbol* already names a special operator, that definition is kept.

If *symbol* already names a function, then any proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed type will be `t`.

If *symbol* already names a function, then any `inline` or `notinline` proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed inline information will be `nil`.

If *symbol* has an associated `setf` expander, then that `setf` expander is pre-

served.

⇒ `compiler-macro-function` *function-name environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function named `cl:compiler-macro-function`.

If *function-name* has a definition as a compiler macro in *environment*, then the corresponding compiler macro function is returned.

If *function-name* has no definition as a compiler macro in *environment*, then this function returns `nil`.

⇒ `(setf compiler-macro-function)`
new-def function-name environment [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:compiler-macro-function)`.

new-def can be a compiler macro function or `nil`. When it is a compiler macro function, then it establishes *new-def* as a compiler macro for *function-name* and any existing definition is lost. A value of `nil` means that *function-name* no longer has a compiler macro associated with it in *environment*.

⇒ `function-type` *function-name environment* [*Generic Function*]

This generic function returns the proclaimed type of the function associated with *function-name* in *environment*.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then an error is signaled.

If *function-name* is associated with an ordinary function or a generic function in *environment*, but no type proclamation for that function has been made, then this generic function returns `t`.

⇒ `(setf function-type)` *new-type function-name environment* [*Generic Function*]

This generic function is used to set the proclaimed type of the function associated with *function-name* in *environment* to *new-type*.

If *function-name* is associated with a macro or a special operator in *environ-*

ment, then an error is signaled.

⇒ `function-inline` *function-name environment* [*Generic Function*]

This generic function returns the proclaimed inline information of the function associated with *function-name* in *environment*.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then an error is signaled.

If *function-name* is associated with an ordinary function or a generic function in *environment*, then the return value of this function is either `nil`, `inline`, or `notinline`. If no inline proclamation has been made, then this generic function returns `nil`.

⇒ `(setf function-inline)`
new-inline function-name environment [*Generic Function*]

This generic function is used to set the proclaimed inline information of the function associated with *function-name* in *environment* to *new-inline*.

new-inline must have one of the values `nil`, `inline`, or `notinline`.

If *function-name* is not associated with an ordinary function or a generic function in *environment*, then an error is signaled.

⇒ `function-cell` *function-name environment* [*Generic Function*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the function named *function-name*. When *function-name* has no definition as a function, the `car` of this cell will contain a function that, when called, signals an error of type `undefined-function`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment.

⇒ `function-unbound` *function-name environment* [*Generic Function*]

A call to this function always succeeds. It returns a function that, when called, signals an error of type `undefined-function`. When *function-name* has no definition as a function, the return value of this function is the contents of the `cons` cell returned by `function-cell`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of

`equal`) function name and the same (in the sense of `eq`) environment. Client code can use the return value of this function to determine whether *function-name* is unbound and if so signal an error when an attempt is made to evaluate the form `(function` *function-name*`)`.

⇒ `function-lambda-list` *function-name environment* [*Generic Function*]

This function returns two values. The first value is an ordinary lambda list, or `nil` if no lambda list has been defined for *function-name*. The second value is true if and only if a lambda list has been defined for *function-name*.

⇒ `(setf function-lambda-list)`
new-lambda-list function-name environment [*Generic Function*]

This generic function is used to associate a new lambda list with a function name.

new-lambda-list is a new lambda list for the function named *function-name*

⇒ `function-ast` *function-name environment* [*Generic Function*]

This function returns the abstract syntax tree corresponding to the *function-name*, or `nil` if no abstract syntax tree has been associated with the function.

⇒ `boundp` *symbol environment* [*Generic Function*]

It returns true if *symbol* has a definition in *environment* as a constant variable, as a special variable, or as a symbol macro. Otherwise, it returns `nil`.

⇒ `constant-variable` *symbol environment* [*Generic Function*]

This function returns the value of the constant variable *symbol*.

If *symbol* does not have a definition as a constant variable, then an error is signaled.

⇒ `(setf constant-variable)` *value symbol environment* [*Generic Function*]

This function is used in order to define *symbol* as a constant variable in *environment*, with *value* as its value.

If *symbol* already has a definition as a special variable or as a symbol macro in *environment*, then an error is signaled.

If *symbol* already has a definition as a constant variable, and its current value

is not `eql` to *value*, then an error is signaled.

⇒ `special-variable` *symbol environment* [*Generic Function*]

This function returns two values. The first value is the value of *symbol* as a special variable in *environment*, or `nil` if *symbol* does not have a value as a special variable in *environment*. The second value is true if *symbol* does have a value as a special variable in *environment* and `nil` otherwise.

Notice that the symbol can have a value even though this function returns `nil` and `nil`. The first such case is when the symbol has a value as a constant variable in *environment*. The second case is when the symbol was assigned a value using `(setf symbol-value)` without declaring the variable as `special`.

⇒ `(setf special-variable)` *value symbol environment init-p* [*Generic Function*]

This function is used in order to define *symbol* as a special variable in *environment*.

If *symbol* already has a definition as a constant variable or as a symbol macro in *environment*, then an error is signaled. Otherwise, *symbol* is defined as a special variable in *environment*.

If *symbol* already has a definition as a special variable, and *init-p* is `nil`, then this function has no effect. The current value is not altered, or if *symbol* is currently unbound, then it remains unbound.

If *init-p* is true, then *value* becomes the new value of the special variable *symbol*.

⇒ `symbol-macro` *symbol environment* [*Generic Function*]

This function returns two values. The first value is a macro expansion function associated with the symbol macro named by *symbol*, or `nil` if *symbol* does not have a definition as a symbol macro. The second value is the form that *symbol* expands to as a macro, or `nil` if symbol does not have a definition as a symbol macro.

It is guaranteed that the same (in the sense of `eq`) function is returned by two consecutive calls to this function with the same (in the sense of `eq`) symbol as the first argument, as long as the definition of *symbol* does not change.

⇒ `(setf symbol-macro)` *expansion symbol environment* [*Generic Function*]

This function is used in order to define *symbol* as a symbol macro with the given *expansion* in *environment*.

If *symbol* already has a definition as a constant variable, or as a special variable, then an error of type `program-error` is signaled.

⇒ `symbol-plist` *symbol environment* [*Generic Function*]

This function is a generic version of the Common Lisp function `cl:symbol-plist`.

It returns the *property list* of *symbol* in *environment*.

⇒ `(setf symbol-plist)` *new-plist symbol environment* [*Generic Function*]

This function is a generic version of the standard Common Lisp function named (`setf cl:symbol-plist`).

Set the *property list* of *symbol* in *environment* to *new-plist*. The consequences are undefined if *new-plist* is not a property list.

⇒ `variable-type` *symbol environment* [*Generic Function*]

This generic function returns the proclaimed type of the variable associated with *symbol* in *environment*.

If *symbol* has a definition as a constant variable in *environment*, then the result of calling `type-of` on its value is returned.

If *symbol* does not have a definition as a constant variable in *environment*, and no previous type proclamation has been made for *symbol* in *environment*, then this function returns `t`.

⇒ `(setf variable-type)` *new-type symbol environment* [*Generic Function*]

This generic function is used to set the proclaimed type of the variable associated with *symbol* in *environment*.

If *symbol* has a definition as a constant variable in *environment*, then an error is signaled.

It is meaningful to set the proclaimed type even if *symbol* has not previously been defined as a special variable or as a symbol macro, because it is meaningful

to use `(setf symbol-value)` on such a symbol.

Recall that the HyperSpec defines the meaning of proclaiming the type of a symbol macro. Therefore, it is meaningful to call this function when *symbol* has a definition as a symbol macro in *environment*.

⇒ `variable-cell` *symbol environment* [*Generic Function*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the variable named *symbol*. When *symbol* has no definition as a variable, the `car` of this cell will contain an object that indicates that the variable is unbound. This object is the return value of the function `variable-unbound`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment.

⇒ `variable-unbound` *symbol environment* [*Generic Function*]

A call to this function always succeeds. It returns an object that indicates that the variable is unbound. The `cons` cell returned by the function `variable-cell` contains this object whenever the variable named *symbol* is unbound. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment (in the sense of `eq`). Client code can use the return value of this function to determine whether *symbol* is unbound.

⇒ `find-class` *symbol environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `cl:find-class`.

If *symbol* has a definition as a class in *environment*, then that class metaobject is returned. Otherwise `nil` is returned.

⇒ `(setf find-class)` *new-class symbol environment* [*Generic Function*]

This generic function is a generic version of the Common Lisp function `(setf cl:find-class)`.

This function is used in order to associate a class with a class name in *environment*.

If *new-class* is a class metaobject, then that class metaobject is associated with the name *symbol* in *environment*. If *symbol* already names a class in

environment than that association is lost.

If *new-class* is `nil`, then *symbol* is no longer associated with a class in *environment*.

If *new-class* is neither a class metaobject nor `nil`, then an error of type `type-error` is signaled.

⇒ `setf-expander` *symbol environment* [*Generic Function*]

This generic function returns the `setf` expander associated with *symbol* in *environment*. If *symbol* is not associated with any `setf` expander in *environment*, then `nil` is returned.

⇒ `(setf setf-expander)` *new-expander symbol environment* [*Generic Function*]

This generic function is used to set the `setf` expander associated with *symbol* in *environment*.

If *symbol* is not associated with an ordinary function, a generic function, or a macro in *environment*, then an error is signaled.

If there is already a `setf` expander associated with *symbol* in *environment*, then the old `setf` expander is lost.

If a value of `nil` is given for *new-expander*, then any current `setf` expander associated with *symbol* is removed. In this case, no error is signaled, even if *symbol* is not associated with any ordinary function, generic function, or macro in *environment*.

⇒ `default-setf-expander` *environment* [*Generic Function*]

This generic function returns the default `setf` expander, to be used when the function `setf-expander` returns `nil`. This function always returns a valid `setf` expander.

⇒ `(setf default-setf-expander)` *new-expander environment* [*Generic Function*]

This generic function is used to set the default `setf` expander in *environment*.

⇒ `type-expander` *symbol environment* [*Generic Function*]

This generic function returns the type expander associated with *symbol* in *environment*. If *symbol* is not associated with any type expander in *environment*, then `nil` is returned.

⇒ `(setf type-expander)` *new-expander symbol environment* [*Generic Function*]

This generic function is used to set the type expander associated with *symbol* in *environment*.

If there is already a type expander associated with *symbol* in *environment*, then the old type expander is lost.

⇒ `find-package` *name environment* [*Generic Function*]

Return the package with the name or the nickname *name* in the environment *environment*. If there is no package with that name in *environment*, then return `nil`. Contrary to the standard Common Lisp function `cl:find-package`, for this function, *name* must be a string.

⇒ `package-name` *package environment* [*Generic Function*]

Return the string that names *package* in *environment*. If *package* is not associated with any name in *environment*, then `nil` is returned. Contrary to the standard Common Lisp function `cl:package-name`, for this function, *package* must be a package object.

⇒ `(setf package-name)` *new-name package environment* [*Generic Function*]

Make the string *new-name* the new name of *package* in *environment*. If *new-name* is `nil`, then *package* no longer has a name in *environment*.

⇒ `package-nicknames` *package environment* [*Generic Function*]

Return a list of the strings that are nicknames of *package* in *environment*. Contrary to the standard Common Lisp function `cl:package-nicknames`, for this function, *package* must be a package object.

⇒ `(setf package-nicknames)` *new-names package environment* [*Generic Function*]

Associate the strings in the list *new-names* as nicknames of *package* in *environment*.

16.3 The static runtime environment

The *static runtime environment* contains runtime objects that the compiler can not prove to have *dynamic extent*, so it must assume that they have *indefinite extent*.

This situation occurs when some function captures the environment by using a `lambda` expression which contains references to local variables outside the expression itself, though such a capture in itself does not necessarily imply that the variables thus referenced have indefinite extent. It all depends on what happens to the function that is the result of the lambda expression.

If that function is just *called*, then there is no capture. This situation might occur as a result of a `let` being transformed into an application of a `lambda` expression.

If that function is passed as an argument to another function which is known not to hold on to its argument for longer than the duration of the function invocation, then there is no capture. The typical situation would be when a `lambda` expression is passed to a standard Common Lisp function such as one of the *sequence* functions that is known to have this property.

In other cases, it might be too risky for the compiler to assume dynamic extent. Even if a function is called which declares its corresponding parameter to have dynamic extent, it might be too risky to trust this, because the function might be redefined later.[1]

Even if all the conditions are present for the compiler to prove that some object has dynamic extent, it would also have to prevent the debugger to access a variable containing that object. Otherwise, the debugger or the inspector could very well hold on to that object indefinitely.

When the compiler must assume that some variable has indefinite extent, then code must be generated to store that variable in a heap-allocated environment.

It is entirely possible that allocating objects on the stack may not have any significant performance advantage. If the nursery collector allows allocation by

[1] An exception would be if the called function is in the same *compilation unit* in which case it can not be redefined without the caller being redefined at the same time.

incrementing a pointer, then allocation in the nursery is just as fast as allocation on the stack. Furthermore, if the nursery collector is a copying collector, then it will not touch dead objects. Therefore, there is also no cost in deleting objects that are no longer referenced. There are three possible additional costs associated with allocating objects in the nursery compared to allocating objects on the stack:

1. If the objects survive for a long time, then they will be traced by the garbage collector, and possibly promoted to an older generation. This situation is unlikely, however. It would mean that the program is allocating objects with dynamic extent, but that are nevertheless kept alive for a long time. Presumably, that means that these objects are being used a lot during the execution of the function that allocated them. If so, the time for the allocation should be negligible compared to the execution of the function.

2. The nursery collector will run more often since more objects are allocated from the nursery than would be the case if these objects with dynamic extent were allocated on the stack. For this aspect to be significant, such allocations must be frequent.

3. Cache performance might be better for the stack than for the nursery, but this would also be unlikely, given that the nursery is relatively small and relatively frequently accessed, so it is very likely to also be in the cache. However, it is entirely possible that objects with dynamic extent become inaccessible before the function that created them exits. Then, allocating the objects on the stack would prevent the garbage collector from reclaiming them, and they would remain allocated beyond their lifetime. If that is the case, then allocating the objects on the stack may in fact harm cache performance simply because a bigger stack may be required.

To make sure that this additional cost is significant and thus worth removing by having the compiler take into account possible stack allocations, a fairly complicated test would have to be devised:

- Two versions of the system would have to be implemented. One that allocates objects with dynamic extent on the stack and another one that

allocates those objects on the heap. For the purpose of benchmarking, the first version could be implemented by having the compiler trust `dynamic-extent` declarations, and by making sure that such declarations are only inserted where they are correct.

- The two versions of the system would need to be executed on a significant and representative application program.
- The work of the garbage collector would have to be profiled for the two cases. In particular, the number of nursery collections and the number of objects being promoted would have to be monitored.
- The maximum size of the stack needs to be monitored to determine whether allocating objects on the stack causes a significant difference in required stack space.
- Difference in cache performance should be determined as well, if possible.

16.4 Runtime information

The compiler will generate runtime information available both to the debugger and to the garbage collector. For each value of the program counter[2], all local locations in use (in registers, stack frame, or static environment) have associated type information. Maintaining this type information does not require any runtime overhead. All that is required is a mapping from a program counter value to a block of runtime information.

A location can have one of different types of values:

- Tagged Lisp value. This is the most general type. It covers every possible Lisp value. The garbage collector must trace the object contained in this location according to its type, which the garbage collector itself has to test for.
- Raw machine value. No location will be tagged with this type, but instead with any of the subtypes given below.

[2]The values of the program counter are *relative* to the beginning of the code object.

- Raw immediate machine value
  - Raw integer.
  - Raw Unicode character.
  - Raw floating-point value.
- Raw machine pointer
  - Raw machine pointer to a cons cell.
  - Raw machine pointer to the header object of a general instance.
  - Raw machine pointer that may point inside the rack of some other object. In this case, the location has to be indicated as *tied* to another location that contains either a Lisp pointer or a raw machine pointer to one of the previous types. This possibility will be used when (say) a pointer to an object is stored in some location, and a temporary pointer to one of the elements of the object is needed. The garbage collector will modify this pointer value by the same amount as that used to modify the rack.

Chapter 17

Object system

SICL will implement the full metaobject protocol (MOP) as described by the Art of the Metaobject Protocol (AMOP) [KR91], in as far it does not conflict with the HyperSpec.

17.1 Classes of class metaobjects

The AMOP stipulates the existence of four class metaclasses, namely:

- `standard-class`. This is the default metaclass for classes created by `defclass`. It is also the metaclass for all classes in the metaobject protocol except `t`, `function`, `generic-function`, and `standard-generic-function`.
- `funcallable-standard-class`. This is the metaclass for `generic-function`, and `standard-generic-function`, and of course for user-defined subclasses of those classes.
- `built-in-class`. This is the metaclass for all built-in classes. More about built-in classes in Section 17.1.2.
- `forward-referenced-class`. This is the metaclass for classes that have been referred to as superclasses, but that have not yet been created by `defclass`.

In addition, the HyperSpec requires the existence of a class metaclass named `structure-class`. Whether conditions have their own metaclass is not specified.

In SICL, every class (i.e., every instance of a metaclass) contains a *unique number* which is an integer assigned sequentially from 0 as a class is created or modified. When a class is modified, its old unique number is never reused, leaving *holes* in the sequence corresponding to numbers that no longer correspond to any classes.[1]

Every *standard-object*[2] contains a *stamp*, which is the unique number of its class as it was when the instance was created. This number is always the first element of the rack of the standard-object. Even though some system classes can not be redefined, standard-objects that are instance of system classes contain a stamp.

17.1.1 Standard classes

Instances of `standard-class` (i.e. "standard classes") are typically created by `defclass`. When no superclasses are given to `defclass`, the class `standard-object` is automatically made a superclass of the new class.

Perhaps the most interesting feature of standard classes is that they can be redefined even though there are existing instances of them. Without this feature, using Common Lisp interactively would not be as obvious as it is, so in some ways, this feature is totally essential for any interactive language.[3]

When there are existing instances of a standard class that is modified, the HyperSpec gives us very specific rules concerning how those instances are to be updated. The HyperSpec is also very clear that existing instances do not need to be updated immediately. But they must be updated no later than immediately before an access to any of the slots of the instance is attempted. The reason for that rule is so that implementations would not have to maintain

[1] It might be possible for the garbage collector to change the unique numbers of the classes, compacting the sequence, but that probably will not be necessary.

[2] Recall that a *standard-object* is an instance allocated on the heap, excluding `cons` cells. (See Chapter 15.)

[3] By "interactive language", we mean a language in which a program is built up by a sequence of interactions that augment and modify the state of some *global environment*.

a reference from a class to each of its instance. Such references would be costly in terms of space, and would have to be *weak*[4] so as to avoid memory leaks.

Instead of keeping weak references from classes to instances, implementations solve the problem of updating obsolete instance by keeping some kind of *version information* in each instance. When some operation on the instance is attempted, the version information is checked against the current version of the class. If the instance is obsolete, it is first updated according to the new definition of the class. Furthermore, the version information must contain enough information of the class as it was when the instance was created to determine whether slots have been added or removed.

In SICL the first location of the rack of each standard-object contains a *stamp*, which is the unique number of the class as it was when the instance was created. In addition, if the instance is *flexible*, it also contains a reference to the *list of effective slots* of the class as it was when the instance was created. This method makes the rack of the instance completely self contained. It allows the garbage collector to trace the obsolete instance, or update it before tracing. It allows for an inspector to inspect the obsolete instance if this should be required. The main purpose of the list of effective slots, however, is making it possible to update an obsolete instance.

Another very handy feature of standard classes, but a much simpler one, is that it allows for instances to *change class*. In other words, without changing the *identity* of the instance, the class that it is an instance of can be changed to a different class. Again, the HyperSpec gives very specific rules about how the instance must be transformed in order to be a legitimate instance of the new class. No special mechanism is required for this feature to work, other than the ability to modify all aspects of an instance except its identity. The identity is preserved by the fact that the *header object* remains the same.

17.1.2 Built-in classes

The HyperSpec contains a significant number of classes that every conforming implementation must contain. Most (all?) of these classes are referred to as

[4] A *weak* reference is a reference that is not sufficient for the garbage collector to keep the object alive. The object is kept alive only if there is at least one *strong* (i.e. normal) reference to it as well.

system classes. Some example of system classes are `symbol`, `package`, `list`, `stream`, etc. The HyperSpec tells us that by *system class* is meant "a class that may be of type built-in-class in a conforming implementation and hence cannot be inherited by classes defined by conforming programs."

Language implementers are thus given a choice as to whether a system class is really a standard class (See Section 17.1.1.), a structure class (See Section 17.1.4.), or a built-in class.

Some of the decisions are determined by the AMOP. For instance, the classes `standard-class` and `built-in-class`, labeled by the HyperSpec as system classes are required by the AMOP to be standard classes. Any implementation that wants to have an implementation of the metaobject protocol as close as possible to what the AMOP requires should take this fact into account.

However, most system classes are not mentioned at all by the AMOP, so there we have a choice. In SICL all these classes will be implemented as either built-in classes or standard classes. None of them will be structure classes.

Even though SICL will implement some of the system classes as built-in classes, this does not mean that we have to use special-purpose ways of implementing them. The SICL object system takes advantage of features of the metaobject protocol such as inheritance to define built-in classes as well as standard classes.

Even for system classes where instances do not all have the same size, notably the `array` class and its subclasses, we plan to take advantage of the metaobject protocol by allowing `make-instance` to take a *size* argument in addition to ordinary initialization arguments. We also plan to allow `defclass` to define built-in classes by passing it `built-in-class` as a metaclass. In that case, the default superclass is `t` instead of `standard-object`. This technique allows us to concentrate all important features of a built-in class and its instances in one place, which will simplify maintenance.

17.1.3 Condition classes

The HyperSpec defines an entire hierarchy of classes with the class `condition` as the root class. This hierarchy is not mentioned by the AMOP.

We plan to implement this hierarchy by defining a class named `condition-class`

analogous to `standard-class` for standard objects. The class `condition` plays a role analogous to the class `standard-object` for instances of `standard-class`.

As with built-in classes (See Section 17.1.2.), we plan to take advantage of the very complete set of tools provided by the metaobject protocol to implement condition classes. In particular, we want to allow for condition classes to be redefined even though existing instances may be present, just the way instances of `standard-object` may exist even though the class is being modified. However, we do not plan to make it possible for an instance of a condition class to have its class changed (i.e. by using `change-class`).

With respect to bootstrapping, the hierarchy of condition classes can be created fairly late in the process. The reason for this is that we plan to define a dumbed-down version of `error` during the bootstrapping process, and that version will not create any condition instances. Furthermore, all SICL code calls `error` and the other condition-signaling functions with the *name* of a condition (which is a symbol) rather than with a *condition instance*, again to allow us to create this hierarchy later in the bootstrapping process.

17.1.4 Structure classes

Just like condition classes (See Section 17.1.3.), structure classes are not mentioned at all in the AMOP. In addition, their description in the HyperSpec is limited to the dictionary entry for `defstruct`.

No part of SICL uses structure classes. The main reason is that they are difficult to work with due to the fact that conforming implementations are allowed to make it impossible to redefine existing structure classes.[5]

However, since we plan for SICL to be a conforming implementation, we naturally plan to include structure classes as well.

The main reason for using structure classes rather than standard classes is

[5] I seem to remember reading somewhere that implementers are encouraged to make it possible to modify existing structure classes in the same way that it is possible to modify standard classes even though there are existing instances, but I don't remember where I read this, and I can't seem to find the place. It might have been in CLtL2 rather than the HyperSpec, but I can't find it there either. Oh well, when I find it, I will remove this footnote.

performance. Structure classes are supposed to be implemented in the "most efficient way possible".[6] Presumably, the restrictions on structure classes exist to allow for an implementation to represent instances as a pointer directly to the vector of slots and avoid any indirection, which saves some memory accesses[7] However, in SICL *all* heap-allocated objects (other than `cons` cells) are represented as a two-word *header object* and a *rack* for reasons of simplicity and in order to allow our memory-management strategy to work.

Since SICL represents instances of structure classes this way, there is no reason to keep the restriction that structure classes can not be modified. For that reason, we plan to avoid that restriction.

Structure classes have another interesting restriction, namely that they allow only single inheritance. This restriction allows slot accessors to be non-generic, because it becomes possible for a slot to have the same physical position in all subclasses. We may or may not take advantage of this possibility. The higher priority for SICL is to make accessors for standard objects fast, rather than to work on an efficient implementation of structures.

With respect to bootstrapping, since SICL does not use structure classes at all for its implementation, implementing `defstruct` can be done fairly late. In fact, we may omit it in the initial version of the system.

17.2 Generic function dispatch

We use the generic dispatch algorithm described in our paper at ICL 2014 [Str14a]

17.2.1 Call history

Each generic function contains a *call history*. The call history is a simple list of *call history entries*. A call history entry associates a list of classes (those of the classes of the required arguments to the generic function) with a *list of applicable methods* and an *effective method*.

[6] Again, I forget where I read this, and I can't find it.

[7] No checks for outdatedness, etc.

As permitted by the AMOP, when the generic function is invoked, its discriminating function first consults the call history (though, for performance reasons, not directly) in order to see whether an existing effective method can be reused, and if so, it invokes it on the arguments received.

If the call history does not contain an entry corresponding to the classes of the required arguments, as required by the AMOP, the discriminating function then first calls `compute-applicable-methods-using-classes`, passing it the classes of the required arguments. If the second value returned by that call is *true*, then the effective method is computed by calling the generic function named `compute-effective-method`. The resulting effective method is combined with the classes of the arguments, and the list of applicable methods into a call history entry which is added to the call history, and the effective method is invoked on the arguments received. If the second value returned by the call is *false*, then the discriminating function calls `compute-applicable-methods` with the list of the arguments received, and then the effective method is computed by calling `compute-effective-method` and finally invoked.

When a method is added to the generic function, the call history is traversed to see whether there exists a call history entry such that the new method would be applicable to arguments with the classes of the entry. If so, the entry is removed. If any entry was removed, a new discriminating function is computed and installed.

When a method is removed from the generic function, the call history is traversed to see whether there exists a call history entry such that the method to be removed is in the list of applicable methods associated with the entry. If so, the entry is removed. If any entry was removed, a new discriminating function is computed and installed.

When a class metaobject is reinitialized, that class metaobject and all of its subclasses are traversed. For each class metaobject traversed, `specializer-direct-methods` is called to determine which methods contain that class as a specializer. By definition, any such method will be associated with a generic function. The call history of that generic function is traversed to determine whether there is an entry containing that method, and if so, the entry is removed from the call history. The AMOP allows the implementation to keep the entry if the *precedence list* of the class does not change as a result of being reinitialized, but for reasons explained below, we remove the entry independently of whether this is

the case. If an entry was removed, a new discriminating function is computed and installed.

17.2.2 The discriminating function

The discriminating function of a generic function is computed from the call history.

If the call history has relatively few entries, then the discriminating function computes the *identification*[8] of each of the required arguments. It then uses numeric comparisons in a tree-shaped computation to determine which (if any) effective method to invoke. In effect, the discriminating function becomes a very simple *automaton* where each transition is determined by a comparison between two small integers. The class numbers become constants inside the compiled code of the discriminating function, making comparison fast. Each argument identification is tested from left to right, without taking the *argument precedence order* of the generic function into account. For each argument, the set of possible effective methods is filtered by a binary search. The search is based on *intervals of class numbers* as opposed to individual class numbers. This optimization can speed up the dispatch considerably when an interval of class numbers yield the same effective method. Since it is common that the unique class numbers of the classes in an inheritance subtree cluster into contiguous intervals, this optimization is often pertinent, and in this case, only two tests (for the upper and the lower bound of the interval of class numbers) are required to determine whether that method is applicable.

The automaton of the discriminating function can not contain class numbers that were discarded as a result of classes being reinitialized, simply because whenever a class is reinitialized, the call history of every generic function specializing on that class or any of its subclasses is updated and the discriminating function is recomputed.

When a generic function is invoked on some arguments, the first step is to compute the *identification* of each required argument. The identification is computed as follows:

[8] Recall that the *identification* of an object is either the *stamp* of the object if it is a standard-object, or the *unique number* of the class of the object if it is a special instance.

- If the object is a *standard-object*, then it is the *stamp* of the instance, i.e. the unique number of the class of the instance as it was when the instance was created. The stamp is stored in the first element of the rack of the instance.

- Otherwise (i.e., if the object is a *special instance*), it is the *unique number* of the class of the object.

The identifications are then used by the automaton to find an effective method to invoke. If the automaton fails to find an effective method, the following steps are taken:

1. The identification is checked against the unique number of the class of the object. If they are not the same, then the object is a standard-object, and it is *obsolete*. The machinery for updating the instance is invoked, and then a second attempt with the automaton is made.

2. If the object identification and the unique number of the class of the object are the same, then `compute-applicable-methods-using-classes` is called. If the first return value is not the empty list and the second return value is *true*, then an effective method is computed and a new entry is added to the call history and the automaton is recomputed. Finally the effective method is invoked.

3. If the first value is the empty list and the second value is still *true*, then `no-applicable-method` is called.

4. If the second return value is *false*, then `compute-applicable-methods` is called. If the result is the empty list, then `no-applicable-method` is called. Otherwise an effective method is computed and invoked.

Notice that in most cases, no explicit test is required to determine whether an instance is obsolete. Also notice that for up-to-date standard-objects, there is no need to access the class of the instance in order to determine an effective method to call. For objects other than standard-objects, there is a small fixed number of possible classes, so determining the identification of an object can be open coded.

Notice also that many interesting optimizations are possible here when the automaton is computed from the call history.

- If there is a single entry in the call history, the automaton can be turned into a sequence of equality tests (one for each required argument). In particular, for an *accessor method*, the automaton degenerates into a single test and a call to either a method that directly accesses the slot or to `no-applicable-method`.
- In the case of a single entry in the call history, and a single applicable accessor method for that entry, the slot access can be open coded in the automaton.
- In the case above and when in addition the specializer of the accessor method is a class for a special instance (such as `fixnum`, `character`, or `cons`), determining the unique number of the class object is not required. Instead, the discriminating function can be a simple test for tag bits.

If the call history has a large number of entries, a different technique may be used. The generic function `print-object` may be such a function. A simple hashing scheme might be better in that case.

17.2.3 Accessor methods

Accessor methods are treated specially when an effective method is computed from a list of applicable methods. Rather than applying the default scheme of generating a call to the method function, when any of the methods returned by `compute-applicable-methods-using-classes` is an accessor method, `compute-discriminating-f` replaces such a method by one that makes a direct access to the slot of the instance. It does this by determining the *slot location* of the slot in instances of the class of the argument. The substitution is *not* done by `compute-applicable-methods-using-cla` itself, because the list of (sorted) methods returned by that function is used for the purpose of caching in order to avoid recomputing an effective method.

Since the location of a slot may change when the class is reinitialized, an effective method computed this way may become invalid as a result. For that reason, whenever a class is reinitialized, any call history entries with methods

specializing on that class or any of its subclasses are removed. This way, a call to `compute-applicable-methods-using-classes` will be forced, and a new location will be determined.

17.3 Dealing with metastability issues

The AMOP gives a few examples of metastability issues that need to be dealt with. It also suggests solutions to these problems, based on recognizing *special cases* such as when `class-slots` is called with the class named `standard-class`.

In SICL we use a different method, called *satiation*. Satiation, in effect, turns metastability problems into *bootstrapping problems* which are much easier to deal with. This technique is documented in our paper at ILC 2014 [Str14b].

Definition 17.1. *A generic function F is said to be* satiated *with respect to a set of classes C if and only if for every combination of classes of required arguments of F for which an effective method can be computed, if a call is made to F with such a combination, then an effective method exists in the cache of F so that no additional generic function needs to be invoked in order for the corresponding effective method to be called.*

As part of bootstrapping the object system, every specified[9] generic function is satiated with respect to the set of specified classes. Furthermore, since specified classes can not be redefined, we can make sure that every specified generic function always remains satiated with respect to the set of specified classes.

Theorem 17.1. *If every specified generic function is satiated with respect to the set of specified classes, then every call to `class-slots` will terminate.*

Proof: Base case: If `class-slots` is called with an instance of `standard-class`, then by definition of satiation, the call will terminate. If `class-slots` is called with some other class, then calls are made to `compute-applicable-methods-using-classes`, `compute-effective-method`, and `compute-discriminating-function` in order to compute a new discriminating function for `class-slots`. But these

[9] By the AMOP.

functions are satiated with respect to `standard-generic-function` and as `class-slots` is a standard-generic-function, those calls will terminate.

17.4 Implementing `slot-value` and `(setf slot-value)`

For reasons of brevity, the following discussion is about the function `slot-value`, but the case of `(setf slot-value)` is entirely analogous.

In *the Art of the Metaobject Protocol (AMOP)* [KR91], these functions are mentioned as prime examples of issues of *metastability*. The scenario that is cited is that `slot-value` of an instance calls `class-slots` on the class of the instance, and `class-slots` is an accessor which calls `slot-value`. In the AMOP, the metastability problem is resolved by recognizing that the recursion must eventually reach `standard-class`, so that treating `standard-class` as a special case resolves the problem.

However, the scenario cited above represents a simplification of the real one. The specification requires `slot-value` to call `slot-value-using-class` with the instance, the class of the instance, and an *effective slot definition metaobject*. In order for `slot-value` to find the right effective slot definition metaobject, it has to traverse the list of effective slot definition metaobjects until one is found that has the *name* of the slot given as an argument. To find the name of an effective slot definition metaobject, `slot-value` has to call `slot-definition-name` which is an accessor which calls `slot-value` which is again a metastability issue.

However, in SICL, as Section 17.2.3 explains, accessors in SICL do not call `slot-value`, so the scenario from the AMOP does not apply. Furthermore, since the accessors `class-slots` and `slot-definition-name` are *satiated* (See Section 17.3.) these functions do not have to call `slot-location` for the base case.

The standard methods on `slot-value-using-class` call the reader function `slot-definition-location` on the effective slot definition and use that location to call `standard-instance-access` on the instance.

Chapter 18

Setf expanders

The HyperSpec requires[1] the following function call forms to have a corresponding `setf` form:

- Accessors for parts of a list: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar`, `cddddr`, `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, `tenth` `rest`, `nth`.
- Array element accessors: `aref`, `row-major-aref`, `char`, `schar`, `bit`, `sbit`, `svref`.
- Other array accessors: `fill-pointer`
- Sequence element accessors: `elt`.
- Other sequence accessors: `subseq`.
- Symbol properties: `symbol-plist`.
- Environment accessors: `symbol-function`, `symbol-value`, `fdefinition`, `macro-function`, `compiler-macro-function`.

[1] See figure 5.7 in section 5.1.2.2 in the HyperSpec.

- Hash table accessors: `gethash`.
- CLOS-related accessors: `class-name`, `slot-value`, `find-class`.
- Miscellaneous: `documentation`, `logical-pathname-translations`, `get`, `readtable-case`.

The HyperSpec also gives the implementer a choice concerning the implementation of `setf` forms either as functions or as `setf` expanders. For SICL we always choose a function whenever possible. Consequently, every `setf` form in the list above is implemented as a function.

Chapter 19

Compiler

19.1 General description

The SICL compiler is a Cleavir-based compiler. As such it uses the intermediate representations defined by Cleavir, and in particular the abstract syntax trees and the instructions of the various levels of intermediate representations that it defines.

However, Cleavir allows for substantial customization. For that reason, in this chapter we describe exactly *how* SICL uses the features that Cleavir provies in order to create a complete compiler.

19.2 Different uses of the compiler

The compiler is used in several different situations. There are essentially three use cases, so it is appropriate to talk about three different compilers:

- The *file compiler*. This compiler is invoked by `compile-file`. It takes a Common Lisp source file and generates a file containing object code (a so-called *fasl* file).
- The *lambda expression compiler*. This compiler is invoked when `compile`

is called with arguments `nil` and a *lambda expression*, and by `coerce` to convert a lambda expression to a function. It compiles the lambda expression in the *null lexical environment*. It produces a *function object*.

- The *top-level expression compiler*. This compiler is invoked by `eval`. It produces a function with no parameters which is then immediately *called* by `eval`.

In addition to these use cases, we also distinguish between different compilers along an orthogonal dimension:

- A *native* compiler is a compiler that produces code for its host Common Lisp system.
- An *extrinsic* compiler is a compiler that produces code for a Common Lisp system other than its host system. An extrinsic compiler is also known as a *cross compiler*.

We now have potentially 6 different compilers. Specific issues related to cross compilation are discussed in Chapter 21.

19.3 Compilation phases

19.3.1 Reading the source code

SICL uses the Eclector[1] implementation-independent version of the standard function `read` and related functions.

While Eclector is also the default reader of SICL, for use with the compiler, Eclector is used to produce a *concrete syntax tree*[2] or CST for short. A CST is a direct mirror of the representation of the source code as ordinary S-expressions, except that each sub-expression is wrapped in a standard object that may contain other information *about* the expression. In particular, the SICL compiler

[1] https://github.com/s-expressionists/Eclector
[2] https://github.com/s-expressionists/Concrete-Syntax-Tree

includes information about *source location* in the CST, so that this information can be propagated throughout the compilation procedure.

In order to accomplish source tracking, SICL starts by reading the entire source file into memory. The internal representation of the source code is a vector of lines, where each line is a string. We use this representation, rather than a single string for the entire file, in order to avoid the issue of how newlines are represented.

The macro `with-source-tracking-stream-from-file` in the package named `sicl-source-tracking` takes a file specification and turns it into a Gray stream by reading the entire file contents and then wrapping that contents in an instance of the standard class `source-tracking-stream`. An instance of that class contains the vector of lines of the initial file, the index of the *current line*, and the index of the *current character* within the current line.

The library `trivial-gray-streams` is used to define methods on the generic functions `stream-read-char` and `stream-unread-char`. These methods modify the index of the current line and the current character as appropriate.

The system `sicl-source-tracking` also defines methods on two generic functions provided by the Eclector subsystem `eclector.parse-result`. The method on `source-position` returns an instance of the class `sicl-source-position`. Instances of this class contain the entire file contents as the vector of lines, together with the line and character index taken from the current values of the stream. The method on `make-source-range` simply constructs a `cons` of the start and the end position, provided they are both non-null.

As a result of this source tracking, every CST that corresponds to a precise location in the source file has a start and an end position associated with it. Not every CST has a location in the source file, however. For example, if the source file contains a list in the form of an opening parenthesis followed by several elements separated by spaces, then only the CSTs corresponding to the entire list, and those associated with each element, have source positions associated with them. CSTs corresponding to the `cons` cells of the list, other than the first, do not have source positions associated with them.

The source is read in a loop that reads top-level expressions until end of file. The expressions are then wrapped in a CST representing the special operator `progn` so as to produce a single CST for the entire source code in the file.

19.3.2 Conversion from CST to AST

Once the CST has been produced by Eclector, it is converted to an *abstract syntax tree*, or AST for short. This conversion involves the use of a *global environment* as defined in Section 16.1 and of lexical environments that evolve during the compilation procedure.

For the lexical environments during compilation, SICL uses a library called *Trucler*[3] which provides a modern version of the environment-related functions defined in the second edition of "Common Lisp, the Language" [Ste90].

In the AST, all macro calls have been expanded, and all other aspects of the compilation environment have been taken into account. For that reason, the AST is independent of the compilation environment.

The AST has a textual representation, so the AST can be saved to a file and a *similar* AST can be created by an application of the `read` function (using a particular read table) to the contents of the file. In fact, this textual representation is the *fasl* format that SICL uses. It fulfills the requirements for *minimal compilation* defined by the Common Lisp standard. For more information, see Chapter 20.

The AST that is generated by Cleavir is a single instance of the class `progn-ast` that contains the AST representations of each of the top-level forms in the original code, with the order preserved.

19.3.3 Conversion from AST to HIR

The acronym HIR stands for *High-level Intermediate Representation*. This representation is defined by Cleavir and documented in chapter 6 of the Cleavir documentation. The main characteristic of HIR is that the objects manipulated are all Common Lisp objects, though some of them might be *unboxed*.

[3] https://github.com/s-expressionists/Trucler

19.3.4 HIR transformations

Introducing explicit argument processing

When HIR code is created by Cleavir the outputs of an `enter-instruction` consist of lexical variables that should be initialized according to the lambda list stored in that instruction. This process is deliberately hidden in the initial HIR version, because it is highly dependent on the implementation.

In SICL, we handle the situation by introducing two new instructions, namely: `compute-argument-count-instruction` and `argument-instruction`. The HIR code that parses the arguments according to the lambda list is sufficiently complex that we documented it separately, in Chapter 25.

Converting `symbol-value-instructions`

A `symbol-value-instruction` is converted to an `fdefinition-instruction` followed by a `funcall-instruction`. The input to the `fdefinition-instruction` is the constant input with a value of `symbol-value`. The output of the `fdefinition-instruction` becomes the first input to the `funcall-instruction`. The second input to the `funcall-instruction` is the constant input (i.e. the symbol of which the value is wanted) of the original instruction.

Since the output of the `funcall-instruction` is the distinguished multiple-value location, the original instruction is converted to a `multiple-to-fixed-instruction` with its original output.

Converting `set-symbol-value-instructions`

A `set-symbol-value-instruction` is converted to an `fdefinition-instruction` followed by a `funcall-instruction`. The input to the `fdefinition-instruction` is the constant input with a value of `(setf symbol-value)`. The output of the `fdefinition-instruction` becomes the first input to the `funcall-instruction`. The second input to the `funcall-instruction` is the constant input (i.e. the symbol of which the value is wanted) of the original instruction. The third input to the is the value to assign to the variable.

Since no outputs are involved, the original instruction is converted directly to the `funcall-instruction`.

Hoisting `fdefinition-instructions`

All `fdefinition-instruction`s are *hoisted*, meaning that they are transformed and executed at the top level. The lexical location that holds the function in question then becomes a shared variable that must later be processed during closure conversion.

We assume that the input of the `fdefinition-instruction` is a constant representing the name of the desired function. For that reason, this transformation must be accomplished before constants are hoisted.

We keep an `equal` hash table of functions that have already been hoisted. The key of the hash table is the name of the function, and the value is the lexical location that holds the *function cell* of the function in the global environment.

The `fdefinition-instruction` is turned into a `car-instruction` with the input being the lexical location holding the function cell. At the top level, we insert a `funcall-instruction` that calls the function that, given a function name, returns a function cell. This function is the first-and only argument to the top-level function, so we find it by inserting an `argument-instruction` with a constant input with a value of 0. The return value of the call is obtained by a `multiple-to-fixed-instruction` immediately following the inserted `funcall-instruction`.[4]

Eliminating `fixed-to-multiple-instructions`

Recall that the `fixed-to-multiple-instruction` takes a number of inputs and stores the corresponding values as multiple values in the distinguished location for this purpose.

Eliminating a `fixed-to-multiple-instruction` involves the introduction of

[4] FIXME: Check whether we really need to insert a `multiple-to-fixed` instruction. It might be possible to insert a `return-value-instruction` since we know that there is a single return value.

two new instruction classes.

The first one is named `initialize-return-values-instruction`. It takes a single constant input value which is a fixnum that indicates the number of multiple values that the distinguished location should hold.

The second one is named `set-return-value-instruction`. It takes two inputs. The first input is a constant input containing a fixnum that indicates the index (starting at 0) of the value to store. The second input is the value to store at that index.

We generate a single `initialize-return-values-instruction` which is given the length of the input list to the original instruction. Then we generate as many `set-return-value-instruction`s as there are inputs, each one given the next input in the list of inputs of the original instruction.

Notice that if the `fixed-to-multiple-instruction` has no inputs, we still generate an `initialize-return-values-instruction` with the value 0, and a single `set-return-value-instruction` with the value `nil` in its constant input.

Eliminating `multiple-to-fixed-instructions`

Recall that the `multiple-to-fixed-instruction` fetches multiple values from the distinguished location for this purpose, and stores each one in a fixed lexical location.

Eliminating the `multiple-to-fixed-instruction` involves the introduction of two new instruction classes.

The first one is name `compute-return-value-count-instruction`. It has no inputs and a single output that will contain the number of values in the distinguished location.

The second one is named `return-value-instruction`. It has a single input which is the index of the value in the distinguished location that is wanted. It has a single output which is a lexical location that will contain the value at the given index in the distinguished location.

This transformation is more complicated than the one used for eliminating the `fixed-to-multiple-instruction`, because of the default values that are given to outputs with an index greater than or equal to the number of available values. The generated code contains two main branches, each one with as many stages as there are outputs of the original instruction. In one branch, the index of the desired value is less than the number of available values, so the corresponding value is assigned to the output. In the other branch, the index of the desired value is greater than or equal to the number of available values, so `nil` is assigned to the output instead. At each stage in the first branch, a test is emitted to see whether there are any more values. If that is not the case, control is transferred to the second branch.

As a special optimization, when there is a single output of the original instruction, we do not emit any `compute-return-value-count-instruction`. Instead a single `return-value-instruction` is generated with an index of 0. We are allowed to do that because even when no values are returned from a function, the first location must contain `nil`.

Hoisting non-trivial constants

Non-trivial constants must be hoisted to the top-level function. Whether a costant is trivial or not is determined by a call to the generic function `trivial-constant-p`.

The class `top-level-enter-instruction` is a subclass of the class `enter-instruction` and it adds a slot holding a list of all non-trivial constants in the entire code graph. When the top-level enter instruction is turned into a closure, these constants become part of the static environment of the top-level function. Initially, the list of constants is empty. During hosting, we keep a parallel list of lexical locations corresponding to each constant.

For each non-trivial constant, a check is first made to see whether it is already in the list of constants in the top-level enter instruction. If not, it is added to the end of the list, and a new lexical location is added to the end of the parallel list. Either way, we determine the position of the constant in the list of constants. The corresponding lexical location has the same position in the parallel list. When a new constant and a new lexical location must be added, a `fetch-instruction` is added at the top level. The inputs of the

`fetch-instruction` are the static-environment location of the top-level enter instruction, and a constant input corresponding to the position of the constant in the list. The output of the `fetch-instruction` is the corresponding lexical location.

The constant input is replaced by the corresponding lexical location.

Notice that this transformation will create shared variables, so it is important that it is performed before closure conversion.

Eliminating `create-cell-instructions`

A `create-cell-instruction` is turned into a `funcall-instruction` with `cons` as the function to call and `nil` as both the arguments.

The `cons` function is loaded from the static environment. To do that, we emit an `aref-instruction` with the static environment location and the offset of the `cons` function in the static environment.

Similarly, the constant `nil` is loaded from the static environment. Again, we emit an `aref-instruction`, this time with an index corresponding to the position of `nil` in the static environment.[5]

Eliminating `fetch-instructions`

A `fetch-instruction` is turned into an `aref-instruction` with a modified index input, in that we add 4 to the constant input of the `fetch-instruction` in order to get the constant input to the `aref-instruction`. The reason for this difference is that the `fetch-instruction` does not take into account the four initial elements of the static environment.

[5] FIXME: Investigate whether we could annotate the `create-cell-instructions` with inputs representing the `cons` function as output from an `fdefinition-instruction` and the constant `nil`. It may not be possible since a `create-cell` instruction is created as a result of closure conversion, and adding an `fdefinition-instruction` means that it must be hoisted, and followed by closure conversion. Perhaps it is possible if we do the closure conversion inside-out, since there are no `create-cell-instructions` in the innermost function.

Eliminating `read-cell-instructions`

A `read-cell-instruction` is simply replaced by a `car-instruction`. The `car-instruction` assumes that its argument is a `cons` cell, but we know that is the case, because we created the cell by a call to the `cons` function.

Eliminating `write-cell-instructions`

A `write-cell-instruction` is simply replaced by a `rplaca-instruction`. The `rplaca-instruction` assumes that its argument is a `cons` cell, but we know that is the case, because we created the cell by a call to the `cons` function.

19.3.5 Conversion from HIR to MIR

MIR differs from HIR in that address calculations are explicit.

The conversion from HIR to MIR starts by *expanding* `funcall-instruction`s as described below. This transformation is done first, because it introduces `read-nook-instruction`s that must be expanded by transformations that are made later.

Following the expansion of `funcall-instruction`s, conversion to MIR is done one function (i.e. starting with an `enter-instruction` at a time. For each function, conversion starts by eliminating `enclose-instruction`s in that function. Following that, the function `process-instruction` is called for each instruction in the function.

Expanding `funcall-instructions`

In HIR, the `funcall-instruction` takes a function object as its first input. During the conversion to MIR, we replace that input with three inputs:

1. A lexical location containing a fixnum that represents the absolute address of the code of the callee.

2. A lexical location containing the static environment to be passed to the callee.

3. A lexical location containing the dynamic environment to be passed to the callee.

The first two items are fetched from the rack of the function object. We use a `nook-read` instruction for each one. For this reason, `funcall-instruction`s must be expanded before `nook-read-instruction`s and `nook-write-instruction`s.

The dynamic environment is a lexical location that is kept in a slot of the `funcall-instruction`.

Eliminating `enclose-instruction`s

The `enclose-instruction` is turned into a `funcall-instruction`. The function being called is an element of the static environment (currently at index 1).

The arguments to the `enclose` function are:

1. A constant representing the absolute address of the entry point of the function resulting from the enclose operation.

2. An arbitrary number of inputs that become the elements of the static environment of the function resulting from the enclose operation.

The absolute address of the entry point is not known when this transformation is applied. We therefore generate a constant of 0 instead. But we must keep track of this constant so that it can be patched, once the address of the entry point is known. For that reason, we do not generate an ordinary constant input, but an instance of a subclass of `constant-input` named `entry-point-input` that, in addition to the constant value, also contains a reference to the enter instruction being enclosed.

The remaining arguments to the `funcall-instruction` are just the inputs of the `enclose-instruction` being replaced.

Therefore, the complete sequence of instructions that replaces the `enclose-instruction` is:

1. An `aref-instruction` taking as inputs the lexical location holding the static environment and a constant input holding the value 1. The output is a lexical location holding the `enclose` function to be called.

2. A `funcall-instruction` with a first input being the lexical location of the output of the `aref-instruction` and the remaining inputs being the inputs of the `enclose-intruction`.

3. A `return-value-instruction` with a constant input having the value 0, meaning we obtain the first and only value returned by the preceding `funcall` instruction.

Eliminating `car-instructions`

To eliminate a `car-instruction` we first insert an `unsigned-sub-instruction`. There are two inputs to that instruction. The first input is the input of the original instruction. The second input is an `immediate-input` with a value of 1. The output is a `raw-integer`. This instruction has a single successor, meaning that we do not care about any carry, since there can not be any.

Next, we change the class of the `car-instruction` so that it becomes a `memref1-instruction`. The input of the `memref1-instruction` is the `raw-integer` computed by the `unsigned-sub-instruction`. The output is the output of the original `car-instruction`.

Eliminating `cdr-instructions`

To eliminate a `cdr-instruction` we first insert an `unsigned-add-instruction`. There are two inputs to that instruction. The first input is the input of the original instruction. The second input is an `immediate-input` with a value of 7. The output is a `raw-integer`. This instruction has a single successor, meaning that we do not care about any carry, since there can not be any.

Next, we change the class of the `cdr-instruction` so that it becomes a `memref1-instruction`. The input of the `memref1-instruction` is the `raw-integer` computed by the `unsigned-sub-instruction`. The output is the output of the original `cdr-instruction`.

Eliminating `rplaca-instructions`

To eliminate an `rplaca-instruction` we first insert an `unsigned-sub-instruction`. There are two inputs to that instruction. The first input is the first input of the original instruction. The second input is an `immediate-input` with a value of 1. The output is a `raw-integer`. This instruction has a single successor, meaning that we do not care about any carry, since there can not be any.

Next, we change the class of the `rplaca-instruction` so that it becomes a `memset1-instruction`. The `:address` input of the `memset1-instruction` is the `raw-integer` computed by the `unsigned-sub-instruction`. The `:value` input of the `memset1-instruction` is the second input of the original instruction.

Eliminating `rplacd-instructions`

To eliminate an `rplacd-instruction` we first insert an `unsigned-add-instruction`. There are two inputs to that instruction. The first input is the first input of the original instruction. The second input is an `immediate-input` with a value of 7. The output is a `raw-integer`. This instruction has a single successor, meaning that we do not care about any carry, since there can not be any.

Next, we change the class of the `rplacd-instruction` so that it becomes a `memset1-instruction`. The `:address` input of the `memset1-instruction` is the `raw-integer` computed by the `unsigned-sub-instruction`. The `:value` input of the `memset1-instruction` is the second input of the original instruction.

Eliminating `aref-instructions`

To eliminate an `aref-instruction`, we first insert an `unsigned-add-instruction`. There are two inputs to that instruction. The first input is the first input of the original instruction. The second input is an `immediate-input` with a value of 3. The output is a `raw-integer`. This instruction has a single successor, meaning that we do not care about any carry, since there can not be any.

Next, we insert a `memref1-instruction`. The input to this instruction is the `raw-integer` computed in the first step. The output is a fresh lexical location that will hold the *rack* of the array.

The next step depends on whether the array is a bit-array or not, because if it is a bit-array we can't just use a memory reference to read the element; it has to be masked and shifted from a bigger datum.

19.3.6 Conversion from MIR to LIR

Register allocation using graph coloring

For register allocation, we use the traditional *graph coloring* method. Since this problem is intractable, we use the method described in [Muc97]. This method uses two rules:

1. The first rule consists of removing a node N in the graph that has a degree that is less than the number of available colors (registers), and solving the reduced problem. The node color chosen for N is any color not chosen by a node adjacent to N.
2. The second rule consists of removing a node N in the graph with the smallest degree that is greater than or equal to the number of available colors, and solving the reduced problem. With some luck, two or more nodes adjacent to N are assigned the same color so that the total number of colors used by the adjacent nodes is less than the number of available colors, leaving at least one color for N.

We modified the standard algorithm to allow for variables to have a *required*

register, so that either the required register gets assigned to the variable, or the variable is spilled. This technique is used for variables that are used to hold *callee-saved* registers to avoid that one such register gets assigned to another. We also allow for a variable to have a *preferred register* which is chosen by the register allocator if it is available at the time a choice has to be made. We use this technique to try as much as possible to compute a value into a register that it is required to end up in eventually, such as a register for a particular argument.[6]

Alternative strategy for register allocation

The material is this section is derived from random thoughts on an alternative strategy for register allocation. It may be removed, or it may be improved in the future.

Assume that for each program point we maintain a set of *entries*. Each entry corresponds to a lexical variable that is *live* at that program point. An entry contains the following information:

- The live variable itself.
- A dedicated stack location to save it in, should that be necessary.
- An estimated *distance* until it is going to be needed (in a register) next.
- A *location* where it is needed next. There are three possible values for this location:
  - A specific register. This possibility is used when the variable is next needed as a register argument in a function call.
  - The symbol `:callee-saves`. This possibility is used when the variable is next needed past a `funcall-instruction`.
  - The symbol `:any`. This possibility is used when the the variable is next needed to be in a register, but there are no restrictions on what kind of register it is.

[6] FIXME: Document how we indicate that an instruction might trash a register. Also, attempt to use the same method to indicate that a global function or variable value might change across a function call.

- A set of locations where it is currently available. This set is represented as a list. An element of the set can be a specific register, or the symbol `:stack` meaning that it is available in its dedicated stack location. This set has at least one element in it.

There are two aspects to this technique. The first aspect is the computation of the estimated distance. The second aspect is how decisions are made to assign a lexical variable to a register and which variable to no longer assign to a register when there are not enough registers to go around.

We first consider the second problem, and discuss the first problem later.

Now let us assume that we have some register assignment A before executing some instruction I. We want to process this instruction and determine a register assignment B after the execution if I. Processing the instruction may involve altering it, but also perhaps inserting new instructions before it and after it.

```
case register-type
  assignment-instruction v2 <- v1
    if v1 is already in a register R:
    then
      if v1 is dead after I:
      then
        . Change I to a nop-instruction.
        . Make R a member of the entry
          for v2 after I
      else
        if a register S of the type
        needed for v2 is available:
        then
          . Make S a member of the entry
            for v2 after I.
        else
          . Find all variables v such that
            a register S of the right type
            is in the set of v.
          . Between v2 and all the v,
            determine which one is
            needed furthest in the future.
          if that variable is v:
```

```
        then
          . Change I to a MOV instruction
            moving R to the privileged
            stack location for v2.
        else
          . Let v0 be the variable that
            is needed furthest in the
            future and let S0 be its
            associated register.
          if the privileged stack location
          when v0 is not in its set:
            . Emit a MOV instruction before I,
              using S0 as a source and the
              privileged stack location as its
              destination.
            . Add the privileged stack location
              to the set for v0.
          . Change I to a MOV instruction with
            R being the source and S0 being
            the destination.
          . Remove S0 from the set associated
            with v0.
  else v1 is not in a register
```

19.3.7 Code generation

19.3.8 Access to special variables and global functions

To access a special variable, the code must first search the dynamic environment in case a per-thread binding exists. If such a binding exists, a tagged pointer of type `cons` is returned, but the pointer refers to an entry on the stack; a dynamic value cell. If no such binding exists, the global value cell is returned.

In general, for every access to a special variable, the value cell must be searched for first. There are many cases, however, where the compiler can detect that multiple accesses to some special variable must refer to the same value cell. In that case, the (pointer to the) value cell is a candidate for register allocation, and computing it is loop invariant.

When it comes to the *contents* of the value cell, however, the situation is more complicated because of the possibility that multiple threads might access the (global) value

cell concurrently. In fact, this is a common situation when a global variable is used for synchronization purposes.

When some function accesses a special variable multiple times, it might seem required to read the contents of the value cell for each such access, even though the compiler can prove that the same cell is involved in each access. However, this turns out not to be the case. If none of the accesses are part of a loop and there is no externally detectable activity between accesses (no assignment to a global variable, no function call), then there is always a possible scenario according to which the same value will be obtained in all the accesses. In such cases, not only the value cell, but also the value itself is a candidate for register allocation. Even if accesses are part of a loop, in some cases the value can be cached in a register. The necessary condition for such register allocation is that the loop provably *terminates* and that there is no externally detectable activity between consecutive accesses.

The situation for global functions is similar to that of special variables, except simpler since no special binding can exist for such accesses. While it is not very probable that anyone attempts to use global functions for synchronization purposes, this can not be excluded either. An exception to the rule is when the global function is a standard Common Lisp function, in which case it can not be replaced, so it is safe to cache the function in a register.

19.3.9 Access to array elements

When an array has not been declared to be `simple` it might seem like every access to an array element would require locking to prevent a different thread from adjusting the array between the time the *length* is determined and the time the element is accessed.

However, in SICL the rack of an array is always *internally consistent* in that the *length* information accurately reflects the number of elements. When an array is adjusted, a different rack is allocated, and the new rack is put in place in a single memory store operation. Therefore, when the elements of an array are processed in some way, the compiler might access the rack only once and cache it in a register. This optimization is possible even in a loop, as long as the compiler can prove that the loop eventually terminates, and as long as there is no externally detectable activity between the accesses.

19.3.10 Access to slots of standard objects

19.4 Random thoughts

The compiler should be as portable as possible. It should use portable Common Lisp for as many of the passes as possible.

The compiler should keep information about which registers are live, and how values are represented in live registers, for all values of the program counter. This information is used by the garbage collector to determine what registers should be scanned, and how. It is also used by the debugger.

Chapter 20

Compiled files

In order to simplify SICL as much as possible, we will use the external format of the Cleavir abstract syntax tree as our so called FASL format. The Cleavir compiler already contains code that makes it possible to read and write abstract syntax trees, so with this decision, there is no need to design an additional file format.

The external format for abstract syntax trees can be read using the Common Lisp standard `read` function with a single additional reader macro, also provided by Cleavir. Since the Common Lisp reader will be present in the initial executable SICL system, there is no special code needed in order to read a FASL file.

Furthermore, the compiler will also be present in the initial initial executable SICL system, so the code for converting an abstract syntax tree into native code is also present.

The Common Lisp standard requires compiled files to be at least *minimally compiled*, and the abstract syntax tree format fulfills the requirement for minimal compilation.

The main downside of using this format for FASL files is decreased performance compared to a format containing native code. However, loading FASL files is typically only done during the development phase of some software, and almost never at run-time. The additional delay required when loading an abstract syntax tree as a result of converting it to intermediate code and then to native code is likely to be barely noticeable during development.

Chapter 21

Cross compilation

In this chapter, we discuss issues specific to *cross compilation*, i.e. when a compiler produces code for a system other than the host Common Lisp system it runs in. For general compilation topics, see Chapter 19.

21.1 General issues with cross compilation

While it may seem obvious and straightforward (though perhaps not easy) to write a cross compiler for Common Lisp, there are some minor issues that have to be considered.

Perhaps the most important extrinsic compiler is the *extrinsic file compiler*.

We exclude the use of the `read` function of the host environment because it can cause some problem.[1] Instead, we use a the Eclector[2] reader, which can be customized in many ways. In particular, it allows *source tracking*, i.e. it can associate file position with every expression in the file.

Clearly, since we are talking about the *file compiler* we face the same restrictions concerning literal objects as a native file compiler does.[3] In addition, though, there

[1] The main difference that is important to bootstrapping is that some implementations use implementation-specific functions in the result of the *backquote* reader macro. This practice is explicitly allowed by the HyperSpec (Section 2.4.6), and also encouraged (Section 2.4.6.1).

[2] See `https://github.com/robert-strandh/Eclector`

[3] See Section 3.2.4 in the HyperSpec.

are some restrictions due to differences between systems that the HyperSpec explicitly allows.

The most important such restriction has to do with floating-point numbers. If (say) the host allows for fewer types of floating-point numbers, then `read` will not accurately represent the source code as the native file compiler for the target would. Code to be compiled by the cross compiler must therefore either avoid floating-point literals altogether, or instead use some expression to create it and make sure that the expression is not evaluated until load time.

The other restriction has to do with *potential numbers* which different systems may define differently. The easy solution is to avoid potential numbers in source code. This should not be hard to do.

21.2 Environments

The HyperSpec[4] gives a list of the environments that are related to compilation. We briefly summarize them here:

- The *startup environment* is the environment of the image from which the compiler was invoked.
- The *compilation environment* is used to hold information that is required by the compiler in order to accomplish its task correctly. Such information consists of definitions and declarations that the compiler needs, for instance definitions of macros and constant variables, and declarations such as `inline` or `special`.
- The *evaluation environment* which the HyperSpec says is a run-time environment in which evaluations by the compiler takes place, typically executions of macro expanders, but also any other code that is indicated by `eval-when` to be evaluated at compile time.
- The *run-time environment* in which the program resulting from the compilation is eventually executed.

The run-time environment is clearly not relevant to cross compilation.

For the purpose of cross compilation, it is practical to think of the startup environment as containing two distinct parts, that we call the *host startup environment* and the *target startup environment*.

[4] See section 3.2.1 of the HyperSpec.

The *host startup environment* is the environment of the image from which the cross compiler was invoked.

The *target startup environment* is the initial compilation environment, in that it contains definitions and declarations that must already exist when the cross compiler is invoked. In SICL the *target startup environment* is represented explicitly as a standard object (i.e., an instance of `standard-object`). Furthermore, the compilation environment of the cross compiler is the same as the *target startup environment* so that any side effects on the compilation environment as a result of the cross compilation persist after the compilation terminates.

The relevant functions of the target startup environment are all Common Lisp functions that access or modify the environment, such as `fdefinition`, `proclaim`, `(setf macro-function)`, etc., but also implementation-specific functions such as functions for accessing and storing type expanders and `setf` expanders.

As with other bundles of related functionality, environment manipulation uses its own package, named `sicl-environment`. In the native environment, this package `use`es the `common-lisp` package so that the symbols `fdefinition`, `proclaim`, etc. are the imported symbols from the `common-lisp` package. During cross compilation, however, these symbols are `shadow`ed by the `sicl-environment` package, so that they are distinct from the analogous symbols of the host `common-lisp` package. Symbols naming macros such as `declaim` and `defun`, however, are not `shadow`ed, but the resulting expansion code contains symbols that are qualified by the `sicl-environment` package.

Let us take an example. Code fragment 21.1 shows a simplified implementation of the `defparameter` macro. It is simplified in that it does not handle the documentation.

```
(defmacro defparameter (name initial-value &optional doc)
  (declare (ignore doc))
  `(progn
     (eval-when (:compile-toplevel)
       (ensure-defined-variable ,name))
     (eval-when (:load-toplevel :execute)
       (setf (symbol-value ,name) ,initial-value))))
```

Code fragment 21.1: Simplified definition of the `defparameter` macro.

The definition of Code fragment 21.1 is established with the package `sicl-environment` as the current package. For that reason, the symbols `ensure-defined-variable` and `symbol-value` are internal to the `sicl-environment` package. When code that in-

vokes the `defparameter` macro is compiled by the cross compiler, the host compiler will evaluate the form `(ensure-defined-variable ,name)`. The result of that evaluation is that the variable is created in the target startup environment. Subsequent compilations by the cross compiler will "see" this definition and consider the variable as `special`. When the resulting code is loaded into the run-time environment, the symbol `symbol-value` in the package `sicl-environment` is imported from the package `common-lisp`.

21.3 Compile-time processing of standard macros

In Appendix A, we show a complete list of all the standard Common Lisp macros.

Most of those macros have no side effects at compile time. They simply expand to some other code to be processed instead. Some of them do, however, expand to `eval-when` forms that include the situation `:compile-toplevel`. We need to make sure that one of the following cases applies for those macros:

- The macro is not used in any top-level form in any file compiled by the cross compiler.
- The cross compiler is able to evaluate the relevant code with the analogous side effects as the native file compiler.
- The cross compiler provides alternative definitions for functions that are invoked as a result of compile-time evaluation, and those alternative definitions provide enough of the side effects to compile all files that are subsequently subject to compilation by the cross compiler.

The standard Common Lisp macros with compile-time side effects are: `declaim`, `defclass`, `defconstant`, `defgeneric`, `define-compiler-macro`, `define-condition`, `define-method-combination`, `define-modify-macro`, `define-setf-expander`, `define-symbol-macro`, `defmacro`, `defmethod`, `defpackage`, `defparameter`, `defsetf`, `defstruct`, `deftype`, `defun`, `defvar`, and `in-package`.

Of those, the following will never appear as top-level forms in any file compiled by the cross compiler:

- `define-condition`, because the condition system is not needed by the cross compiler.
- `define-method-combination`.

- `defstruct`.

The following macros are handled in an analogous way during cross compilation: `declaim`, `defconstant`, `define-compiler-macro`, `define-modify-macro`, `define-setf-expander`, `define-symbol-macro`, `defmacro`, `defparameter`, `defsetf`, `deftype`, `defun`, `defvar`, and `in-package`.

We are left with the following macros: `defclass`, `defgeneric`, and `defmethod`.

For `defclass` the HyperSpec says that the *class name* must be made available for use as a specializer in `defmethod` and as the `:metaclass` option in subsequent invocations of `defclass`. It is thus suggested that `defmethod` checks at compile-time that the specializers exist, though the HyperSpec does not mention any such checks.

The HyperSpec also says that if `find-class` is called with the relevant environment argument, then the class object should be returned. We can think of no need to invoke `find-class` at compile time.

Now, according to the specification of the metaobject protocol, `defclass` expands to a call to `ensure-class` (not a standard Common Lisp function). Our solution is thus to provide a compile-time version of `sicl-clos:ensure-class` that only makes the *class name* available.

For `defgeneric`, the HyperSpec says that the implementation is not required to perform any compile-time side effects, but that it can choose to store information about arguments and such if it so wants. As a consequence, we handle it the same way we handle `defun`.

For `defmethod`, the HyperSpec says that the implementation is not required to perform any compile-time side effects. The compile-time side effects will be in the case where no prior `defgeneric` form has been seen, in which case we store information about the derived parameters of the implicitly created generic function. As suggested by the HyperSpec in the description of `defclass`, we also verify that the specializer class names have been previously seen.

Chapter 22

Bootstrapping

22.1 General technique

SICL is bootstrapped from an existing Common Lisp implementation that, in addition to the functionality required by the standard, also contains the library `closer-mop`. This Common Lisp system is called the *host*. The result of the bootstrapping process is an *image* in the form of an executable file containing a SICL system. This system is called the *target*. The target image does not contain a complete Common Lisp system. It only contains enough functionality to load the remaining system from compiled files. (See Section 22.3.).

In general, the target image can be thought of as containing *graph* of Common Lisp objects that have been placed in memory according to the spaces managed by the memory manager. To create this graph, we first generate an *isomorphic* graph of host objects in the memory of an executing host system. To generate the target image, the isomorphic host graph is traversed, creating a target version of each object in the host graph, and placing that object on an appropriate address in the target image.

The isomorphic host graph contains objects that are *analogous* to their target counterparts as follows:

- A target `fixnum` is represented as a host integer. Whether the integer is a host fixnum or not depends on the fixnum range of the host.
- A target `character` is represented as a host character.
- A target `cons` cell is represented as a host `cons` cell.

- A target standard object is represented as a host `standard-object` for the *header* and a host `simple-vector` for the *rack*.

- Target objects such as bignums or floats are not needed at this stage of bootstrapping, so they do not have any representation as host objects.

22.2 Global environments for bootstrapping

During different stages of bootstrapping, a particular *name* (of a function, class, etc) must be associated with different objects. As a trivial example, the function `allocate-object` in the host system is used to allocate all standard objects. But `allocate-object` is also a target function for allocating objects according to the way such objects are represented by the target system. These two functions must be available simultaneously.

Most systems solve this problem by using temporary names for target packages during the bootstrapping process. For example, even though in the final target system, the name `allocate-object` must be a symbol in the `common-lisp` package, during the bootstrapping process, the name might be a symbol in a package with a different name.

In SICL we solve the problem by using multiple *first-class global environments*.

For the purpose of bootstrapping, it is convenient to think of `eval` as consisting of two distinct operations:

- Compile. A *compilation environment* is used to expand macros and for other compilation purposes. The result of compilation is code that is *untied* to any particular environment.

- Tie. The untied code produced by the first step is *tied* to a particular run-time environment. Tying is accomplished by calling the top-level function created by the compilation. This function takes a single argument, namely a "function-cell finder" function. Calling that argument function with a function name, returns a function cell in a particular environment, thereby tying the code to that particular environment.

The reason we need to separate these two operations is that for bootstrapping purposes, the two are going to use distinct global environments.

22.3 Viable image

An image I is said to be *viable* if and only if it is possible to obtain a complete Common Lisp system by starting with I and loading a sequence of ordinary compiled files.

22.4 Bootstrapping stages

22.4.1 Stage 1, bootstrapping CLOS

Definitions

Definition 22.1. *A* simple instance *is an instance of some class, but that is also neither a class nor a generic function.*

Definition 22.2. *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 22.3. *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 22.4. *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 22.5. *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 22.6. *A* simple host instance *is a host instance that is neither a host class nor a host generic function.*

Definition 22.7. *An* ersatz instance *is a target instance 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, the header is an instance of the host class* `funcallable-standard-object` *so that some ersatz instances can be used as functions in the host system.*

Definition 22.8. *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 22.9. *An* ersatz class *is an ersatz instance that can be instantiated to obtain another ersatz instance.*

Definition 22.10. *An* ersatz generic function *is an ersatz instance that is also a generic function. It is possible for an ersatz generic function 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 22.11. *An* ersatz method *is an ersatz instance that is also a method.*

Definition 22.12. *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 22.13. *A* bridge generic function *is a target generic function represented as a simple host instance, though it is an instance of the host function* `funcallable-standard-object` *so it can be executed by the host.*

Arguments to a bridge generic function are ersatz instances. The bridge generic function uses the stamp *(See Section 17.2.2.) of the required arguments to dispatch on.*

The methods on a bridge generic function are bridge methods.

Definition 22.14. *A* bridge method *is a target method represented by 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.*

Preparation

In addition to the host environment, eight different SICL first-class environments are involved in the bootstrapping procedure. We shall refer to them as E_0, E_1, E_2, E_3, E_4, E_5, E_6, and E_7.

Phase 0

In phase 0, we load enough functionality into environment $E0$ to be able to compile source files into *fasl* files. This functionality involves mainly standard macros that are required during compilation.

We then compile all the files that are ultimately used in the final system. Since *fasl* files are independent of any particular environment, the same *fasl* file can be loaded into different environments during the following phases of stage 1.

Any code can be compiled, provided that the macros that are called by the code are defined in environment E_0. In particular, code that defines macros can be compiled into `fasl`s.

Phase 1

We define a class named `sicl-boot-phase-1:funcallable-standard-class` in the host environment. It is defined as a direct subclass of the host class `closer-mop:funcallable-standar` When we evaluate `defclass` forms in phase 2, the classes created are instances of this class. We could have chosen this class only for instances that need to be executable in the host, and a subclass of the host class `standard-class` for the others, but the host class `funcallable-standard-class` can do everything that the host class named `standard-class` can, so we simplify the code by using one single class.

We define an `:around` method on `initialize-instance` in the host environment, specialized to `sicl-boot-phase-1:funcallable-standard-class`. The purpose of this `:around` method is to remove the `:reader` and `:accessor` slot options supplied in the `defclass` forms that we evaluate in phase 1. Without this `:around` method, the host function `initialize-instance` would receive keyword arguments `:readers` and `:writers` and it would then add methods to host generic functions corresponding to the names given. Instead, this `:around` method adds the readers and writers to the generic function with the corresponding name in envirnoment E_3. It assumes that this generic function exists, so we must create it explicitly before a class that defines the accessor method can be defined.

In environment E_1, we define the following classes:

- `t`. This class is the same as the host class `t`. It will be used as a specializer on method arguments that are not otherwise specialized.
- `common-lisp:standard-generic-function`. This class is the same as the class with the same name in the host environment. It will be used to create host generic functions in environment E_2.
- `common-lisp:standard-method`. This class is the same as the class with the same name in the host environment. This class will be used to create methods on the generic functions that we create in environment E_3.
- `common-lisp:standard-class`. This class is the same as the class named `sicl-boot-phase-1:funcallable-standard-class` in the host environment. It will be used to create most classes in E_2.

- `common-lisp:built-in-class`. This class is the same as the class named `sicl-boot-phase-1:funcallable-standard-class` in the host environment. It will be used to create some classes in E_2, for example `t` and `function`.
- `sicl-clos:funcallable-standard-class`. This class is the same as the class named `sicl-boot-phase-1:funcallable-standard-class` in the host environment. It will be used to create some classes in E_2, such as `generic-function` and `standard-generic-function`.
- `sicl-clos:standard-direct-slot-definition`. This class is the same as the class named `closer-mop:standard-direct-slot-definition` in the host environment. This class will be used to create slot-definition metaobjects for the classes that we create in environment E_2.

Phase 2

The purpose of phase 2 is:

- to create host generic functions in E_3 corresponding to all the accessor functions defined by SICL on standard MOP classes, and
- to create a hierarchy in E_2 of host standard classes that has the same structure as the hierarchy of MOP classes.

Three different environments are involved in phase 2:

- Environment E_1 is used to find find host classes to instantiate. The class named `standard-class` in E_1 is used to instantiate most of the classes in environment E_2, for example, `standard-class`, `built-in-class`, `slot-definition`, etc. The class named `built-in-class` in E_1 is used to create classes `t` and `function` as well as some non-MOP classes in E_2. The class named `funcallable-standard-class` in E_1 is used to create classes `generic-function` and `standard-generic-function`, also in environment E_2. The class named `standard-direct-slot-definition` in E_1 is used to define slot-definition metaobjects for the classes in environment E_2. Environment E_1 is also used to find host classes `standard-generic-function` and `standard-method` to instantiate in order to create generic functions in environment E_3 as well as methods on those generic functions.
- The run-time environment E_2 is where instances of the host classes named `standard-class`, `funcallable-standard-class`, and `built-in-class` in environment E_1 will be associated with the names of the MOP hierarchy of classes. These instances are thus host classes. The entire MOP hierarchy is created as are some built-in classes such as `cons` and some of the number classes.

- The run-time environment E_3 is where instances of the host class named `standard-generic-fun` will be associated with the names of the different accessors specialized to host classes created in E_2.

One might ask at this point why generic functions are not defined in the same environment as classes. The simple answer is that there are some generic functions that were automatically imported into E_2 from the host, that we still need in E_2, and that would have been overwritten by new ones if we had defined new generic functions in E_2.

Several adaptations are necessary in order to accomplish phase 2:

- A special version of the function `ensure-generic-function` is defined in environment E_3. It checks whether there is already a function with the name passed as an argument in E_3, and if so, it returns that function. It makes no verification that such an existing function is really a generic function; it assumes that it is. It also assumes that the parameters of that generic function correspond to the arguments of `ensure-generic-function`. If there is no generic function with the name passed as an argument in E_3, it creates an instance of the host class `standard-generic-function` and associate it with the name in E_3. To create such an instance, it calls the host function `make-instance`.
- The function `ensure-class` has a special version in E_2. Rather than checking for an existing class, it always creates a new one.

Phase 2 is divided into two steps:

1. First, the `defgeneric` forms corresponding to the accessors of the classes of the MOP hierarchy are evaluated using E_2 as both the compilation environment and run-time environment. The result of this step is a set of host generic functions in E_3, each having no methods.
2. Next, the `defclass` forms corresponding to the classes of the MOP hierarchy are evaluated using E_2 as both the compilation environment and run-time environment. The result of this step is a set of host classes in E_2 and host standard methods on the accessor generic functions created in step 1 specialized to these classes.

Phase 3

The purpose of phase 3 is to create a hierarchy of bridge classes that has the same structure as the hierarchy of MOP classes.

Three different environments are involved in phase 3:

- The run-time environment E_2 is used to look up metaclasses to instantiate in order to create the bridge classes.
- The run-time environment E_3 is the one in which bridge classes will be associated with names.
- The run-time environment E_4 is the one in which bridge generic functions will be associated with names.

We start by creating generic functions corresponding to all slot accessors that are defined in the MOP hierarchy. We then create bridge classes corresponding to the classes of the MOP hierarchy. When a bridge class is created, it will automatically create bridge methods on the bridge generic functions corresponding to slot readers and writers.

Creating bridge classes this way will also instantiate the host class `target:direct-slot-definition`.

In this phase, we also prepare for the creation of ersatz instances.

Phase 4

The purpose of this phase is to create ersatz generic functions and ersatz classes, by instantiating bridge classes.

At the end of this phase, we have a set of ersatz instances, some of which are ersatz classes, except that the `class` slot of the header object of every such instance is a bridge class. We call such ersatz instances *impure*. We also have a set of ersatz generic functions (mainly accessors) that are ersatz instances like all the others.

Phase 5

The first step of this phase is to finalize all the ersatz classes that were created in phase 4. Finalization will create ersatz instances of bridge classes corresponding to effective slot definitions.

The second step repeats the creation of MOP generic functions and MOP classes, this time in environments E_6 and E_5 respectively. As opposed to the objects created in phase 4, the objects created in this phase are *pure* ersatz objects, in that the class slot of the header object is also an ersatz object, albeit impure.

Phase 6

Phase 7

The purpose of this phase is to create ersatz instances for all objects that are needed in order to obtain a viable image, including:

- ersatz built-in classes such as `package`, `symbol`, `string`, etc.,
- ersatz instances of those classes, such as the required packages, the symbols contained in those packages, the names of those symbols, etc.
- ersatz standard classes for representing the global environment and its contents.
- ersatz instances of those classes.

Phase 8

The purpose of this phase is to replace all the host instances that have been used so far as part of the entire ersatz structure, such as symbols, lists, and integers by their ersatz counterparts.

Phase 9

The purpose of this phase is to take the simulated graph of objects used so far and transfer it to a *memory image*.

Phase 10

Finally, the memory image is written to a binary file.

Chapter 23

Garbage collector

To fully appreciate the contents of this chapter, the reader should have some basic knowledge of the usual techniques for garbage collection. We recommend "The Garbage Collection Handbook" [JHM11] to acquire such basic knowledge. We also recommend Paul Wilson's excellent survey paper [Wil92].

We use a per-thread nursery combined with a global allocator for older objects.

In Chapter 15 we describe the data representation where every heap allocated object is either a two-word `cons` cell or a *standard object* represented as a two-word *header* where the first of the two words is a tagged pointer to the class object, and the second of the two words is a pointer to the *rack* using a special tag for racks. For the purpose of garbage collection, in many ways, `cons` cells and two-word headers are treated the same way. For that reason, we refer to either one as a *dyad*.

We begin this chapter by describing the algorithms used in the global collector and the nursery collector. Finally, we describe the synchronization procedure required for the nursery collectors to collaborate with the global collector.

23.1 Global collector

23.1.1 General description

The global collector is a concurrent collector, i.e., it runs in parallel with the mutator threads. With modern processors, it is probably practical to assign at least one core

more or less permanently to the global collector. According to current thinking, the global collector will be a combination of a mark-and-sweep collector and a traditional memory allocator as implemented by `malloc()/free()` in a C environment.

We define a global heap divided into two parts called the *dyad part* and the *rack part*. The dyad part is a single vector consisting of two-word blocks. This is where dyads are allocated. The rack part of the global heap is organized as the space managed by an ordinary memory allocator. Our adaptation of Doug Lea's memory allocator that we used for this purpose can be found in Appendix C.

Since all dyads consist of two words, we can use a mark-and-sweep collector for these objects without suffering any fragmentation. The advantage of a mark-and-sweep collector is that objects will never move, which is preferable when they are used as keys in hash tables and when they are used to communicate with code in foreign languages that assume that an address of an object is fixed once and for all.

Since the rack part of the global heap is managed by an ordinary memory allocator, the racks also do not move once allocated. This fixed position is advantageous for code on some architectures. For example, the correspondence between source code location and values of the program counter does not have to be updated as a result of code being moved by the garbage collector.

Another great advantage of racks being in a permanent position is that mutator threads can cache a pointer without the necessity of this pointer having to be updated as a result of a garbage collection in the global heap. Garbage collection in the global heap can therefore be done in parallel with the execution of the mutator threads.

The global collector cycles through the following phases:

1. Idle. In this phase, the global collector is not doing any work.
2. Requesting roots. In this phase, the global collector indicates to each running application thread that it needs to transmit its part of the global root set to the global collector. For each blocked application thread, another thread (newly spawned or taken from a thread pool) does the work on its behalf.
3. Waiting for roots. In this phase, the global collector waits for each application thread or each spawned thread to finish indicating its part of the global root set. Dyads and racks are treated as independent objects.
4. Mark. In this phase, the global collector traces and marks the live objects starting with the marked roots.
5. Collecting unmarked dyads. In this phase, the global collector scans the dyads and collects the unmarked ones into a linked list.

6. Freeing unmarked racks. In this phase, the global collector iterates through the allocated racks in the rack part of the heap and frees unmarked racks.
7. Merging free lists. In this phase, the global collector merges the newly created linked list of unmarked dyads with the existing free list.
8. Clearing mark bits. In this final phase, the mark bits used for marking objects as live are cleared.

23.1.2 Idle phase

In this phase the global collector is not doing any work. Whether it has allocated threads that are blocked or no threads allocated remains to be decided. During this phase, when application threads request space from the global heap, objects are allocated *white* (see below for more information on the tri-color technique).

We maintain a free list of dyads to be used to grant requests for objects by mutator threads. We start a collection before the free list is empty so that mutator threads can continue doing useful work during a garbage collection of the global heap. If the free list of dyads ever becomes empty, then mutator threads must wait until more free dyads become available as a result of a garbage collection of the global heap.

23.1.3 Requesting roots

See Section 23.3 for more details on this phase. During this phase, when application threads request space from the global heap, objects are allocated *black* (see below for more information on the tri-color technique).

23.1.4 Waiting for roots

In this phase, we wait for each mutator thread to finish reporting its part of the root set.

23.1.5 Mark

The global collector uses the traditional tri-color marking technique. Recall that the tri-color technique works as follows:

- An object belongs to one of three sets, usually called *black*, *white*, and *gray*.

- Black objects are known to be live. White objects have not yet been traced. Remaining white objects at the end of a full collection are dead.
- Gray objects represent a frontier between black and white objects. No black object may refer to a white object.
- Initially, the root objects are colored gray and all remaining objects are colored white.
- In each iteration, a gray object is selected. The white objects it refers to are colored gray, and the object itself is then colored black.
- Collection ends when there are no more gray objects.

The root set is determined by requesting that each application thread trace live objects from their respective roots, and informing the global collector when this procedure reaches an object allocated in the global heap. For application threads that are blocked, for example waiting for input or output, the global collector spawns a thread to run the garbage collector on behalf of the application thread.

The global collector maintains two bitmaps for the purpose of marking dyads, one for black objects and one for gray objects. Each bitmap contains a bit for every dyad. In addition, the gray bitmap has a multi-level *index*, making it possible to find an arbitrary gray object in only a few cycles. For each 64-bit word in the primary bitmap, a bit in a secondary bitmap is maintained. The bit in the secondary bitmap is set if there is at least one bit set in the 64-bit word in the primary bitmap. Additional levels of index exist until the last level fits in a single 64-bit word.

To include an object in the set of gray objects, the address is used to determine a bit position in the primary bitmap. Before the corresponding bit is set, the 64-bit word that this bit is contained in is tested to see whether it has all bits cleared. If that is the case, the bit position in the secondary bitmap is determined, and recursively set in the same way. Finally the bit is set.

To exclude an object from the set of gray objects, the address is used to determine a bit position in the primary bitmap. The bit is then cleared. If this operation results in a 64-bit word that has all bits cleared, then the bit position in the secondary bitmap is determined, and recursively cleared in the same way.

To find a gray object, start at the top-level index and find an arbitrary position contining a bit that is set. This position corresponds to an index in the next-level index. Repeat the procedure until the primary bitmap is reached.

Since the operation of finding a gray object might be somewhat costly, we also keep a fixed-size cache of gray objects, organized as a stack represented as a vector. Gray objects are initially taken from the cache. Only when the cache is empty is the more

costly technique used. Whenever the objects referred to by a gray object are colored gray, they are also included in the cache (provided there is room in the cache). The cache could for instance be a vector with around a million elements. Such a vector would occupy only 8 megabytes of memory which is comparable to the space taken up by the gray bitmaps.

Every rack has a (single) mark bit as well. Since there is plenty of room in the chunk that is used for the rack, there is no need to use separate bitmaps for the mark bits of the racks. Instead, a bit in the *size* field described in Appendix C is used.

Each rack has a single mark bit, so it can be only black or white. This invariant is simply maintained by the recursive scanning of its contained object, whenever it is found to be live. The reason for not allowing racks to be gray is that we would then need to scan the entire rack zone for gray objects, or maintain additional separate mark bits with indices, just as we do for dyads. [1]

23.1.6 Collecting unmarked dyads

We collect unmarked dyads into a list that is separate from the free list used to grant request for allocations by mutator threads. We do not want requests for allocations by mutator threads to be granted from this list until we have freed the racks these dyads refer to.

If the number of dyads recovered is insufficient, i.e. a new allocation would be triggered very soon, more memory is requested from the system and used for dyads.

23.1.7 Freeing unmarked racks

During this phase, the racks that are indicated as allocated are traversed, and any rack that is unmarked is freed.

[1] We could allow for a rack to be gray, provided there is room for it in the fixed-size cache. We could even evict a dyad from the cache to make room for a rack. If the global collector never colors a rack gray, and instead always recursively scans it and colors the object it contains gray instead, then the only situation where a rack is colored gray would be when the mutator has a pointer to a rack with no pointer to the header, so there will be very few racks that are gray. The reason we would like to allow for a rack to be gray sometimes is so that the mutator will not have unbounded pauses when it is asked for roots.

23.1.8 Merging free lists

Once the racks of unmarked dyads have been freed, the two free lists are merged into a single one.

23.1.9 Clearing mark bits

Finally, we clear all the bitmaps used for marking, and we either enter the idle phase, or start a new collection, depending on the number of dyads left in the free list.

23.1.10 Write barrier

The global collector is subject to a write barrier. Let G be some object in the global heap, and let N be some object in a nursery. The write barrier must prevent the existence of a reference from G to N. Therefore, when attempt is made to create such a reference, N and the objects in its transitive closure are first moved to the global heap. As a result, there are no references from the global heap to objects in any nursery. The write barrier is implemented as a test, emitted by the compiler, to determine:

1. whether the object written to is indeed an object in the global heap, and
2. whether the datum being written is a reference to a heap-allocated object (as opposed to an immediate object).

In many cases, this test can be omitted as a result of *type inference*, for instance if the datum being written can be determined at compile time to be an immediate object.

The write barrier is tripped whether the reference to be stored is to an object in the global heap or to an object in the local heap. In the first case, the write barrier is used to make sure there is not a reference to a white object stored in a black object. In the second case, the write barrier is used to trigger a migration of local objects to the global heap.

23.1.11 Protocol

The names of these functions are exported by the package named `sicl-global-allocator`.

⇒ `copy-object` *object* [*Function*]

This function takes an object that is allocated in some thread-local heap, copies it, and returns a copy that is allocated in the global heap. All the objects referred to by *object*, including the class of a standard object, must either be immediate objects, or objects located in the global heap,

⇒ `make-array` *dimensions* `&key` *element-type initial-element initial-contents adjustable fill-pointer* [*Function*]

This function is similar to the Common Lisp function with the same name. The difference is that this function can not be used to allocate displaced arrays. This function can be used by client code to allocate arrays that are too big to be allocated in the thread-local heap. All arguments must either be immediate objects, or objects located in the global heap.

⇒ `allocate-rack` *size* [*Function*]

Allocate a rack containing *size* words and return an untagged pointer to it. Because the pointer is untagged, it will look like a fixnum. If the reference returned by this function is dropped and not passed to `allocate-header` then the corresponding memory is lost forever.

⇒ `allocate-header` *class rack* [*Function*]

This function allocates a new two-word header and returns a tagged pointer to it. The argument *class* is the class of the object to be constructed. The argument *rack* is the rack that holds the data contained in the object to be constructed. The `rack` argument must be an untagged pointer to the rack. The tag specific to racks will be added by this function.

⇒ `cons` *car cdr* [*Function*]

This function allocates a new two-word `cons` cell and returns a tagged pointer to it. The arguments have the same meaning as for the standard Common Lisp function `cons`.

23.2 Nursery collector

23.2.1 General description

The nursery should be fairly small so as to guarantee short delays, a few megabytes at most. Instead of a traditional copying collector in which objects that survive a collection are promoted, we use a *sliding collector* for the nursery. Such a collector gives a very precise idea of the age of different objects, so objects would always be

promoted in the order of oldest to youngest. This technique avoids the problem in traditional copying collectors where the allocation of some intermediate objects is immediately followed by a collection, so these objects are promoted even though they are likely to die soon after the collection. In a sliding collector, promotion will happen only when a collection leaves insufficient space in the nursery, at which point only the number of objects required to free up enough memory would be promoted, and in the strict order of oldest to youngest.

23.2.2 Allocation

The nursery allocator maintains two pointers into the nursery heap, namely the *dyad free pointer* and the *rack free pointer*. The dyad free pointer always has a smaller value than the rack free pointer. These pointers are illustrated in Figure 23.1.

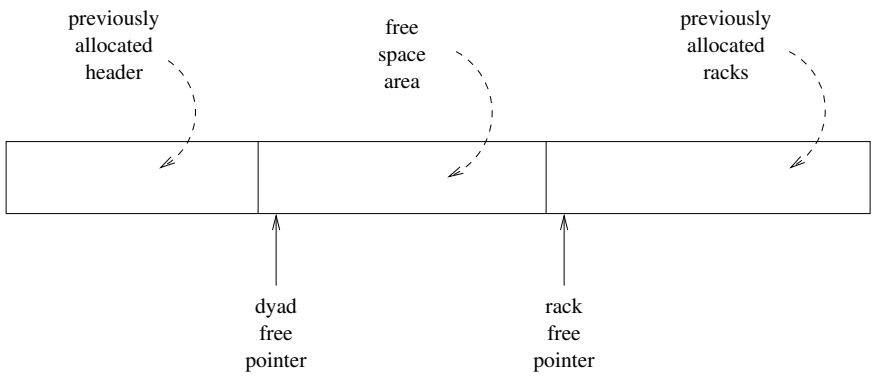

Figure 23.1: Allocation pointers in the nursery.

To allocate a dyad, the value returned is the existing value of the dyad free pointer with appropriate tag bits added. Before this value is returned, the dyad free pointer is incremented by a value corresponding to two full words.

To allocate a rack, the rack free pointer is first decremented by a value corresponding to the number of words in the rack. The new value of the rack free pointer is then returned. The tag bits (111) for a rack are added.

When a request for allocation would result in less than six free words left in the nursery, a collection is first triggered before the request is granted. The reason we need these six spare words is explained in Section 23.2.7.

23.2.3 Finding roots

Several of the phases of the nursery collector involve finding every *root*, i.e., every Common Lisp object that is currently in a processor register or on the stack. What is done to the object once it is found, depends on the phase.

Finding the roots is a fairly complicated procedure, which is probably why some implementations prefer traversing the stack *conservatively*, i.e., considering every word on the stack that *might* be a root to actually *be* one. But such tricks complicate other aspects of the garbage collector.

In SICL we use *precise* stack traversal, meaning that we know exactly when a location contains a root and when it does not.

This precise traversal is complicated by the fact that some registers are so called *callee saves* registers, meaning that a particular function invocation does not save the register to the stack before making a function call, and instead it relies on a the first function on the call chain that needs to use the register to save it and restore it after use. A direct consequence of this scheme is that a register or a stack location close to the top of the stack may contain a datum that belongs to a function invocation arbitrarily further down the stack, and whether that register or stack location is a root can only be determined by knowing the current value of the program counter in that function invocation.

To find the roots owned by each stack frame, we use the *return address* stored in the stack frame immediately above the one we are interested in. That return address is used to access the *frame map* and the *callee-saves map* as described in Section 15.7. Recall that the *frame map* is a bitmap that contains an element for each stack location containing a Common Lisp object that might be heap allocated. For a callee-saves register that might contain a heap-allocated Common Lisp object, the frame map indicates the frame location where the previous value of the register was stored. Also recall that the callee-saves map contains an entry for each callee-saves register. The value of an entry is the stack location where the previous value of the register was stored, or 0 of the register is not used by the current invocation.

When the stack traversal starts, the top stack frame belongs to the garbage collector itself. That frame does not contain any roots. We maintain a table called *register contents table* of contents of callee-saves registers. The table is initially filled with the current contents of those real registers.

The stack is then traversed, frame by frame, starting with the second one from the top, and using the return address in the top frame. Initially, the register contents table contains the current value of those registers when the root-finding function was called.

When a stack frame is visited, the *callee-saves map* for the current stack frame is consulted. Recall that this map is indexed by a callee-saves register and contains a stack location in the current frame. The values of the register entry in the register contents table and the corresponding stack frame are swapped. After these swaps, the register contents table contains the callee-saves registers for the next frame, and the frame locations contain the value of the callee-saved registers owned by the invocation of this frame.

Next, the *frame map* is used to determine location of roots. When those roots have been processed, the next frame is visited.

When the bottom frame of the stack has been processed, we back up by applying the information in the callee-saves map again, thereby swapping back to the correct contents of the stack frame.

As an example of finding the roots, consider a situation where we have three callee-saves registers (0, 1, and 2), and three stack frames plus the one running the root-finding function. Let us call these stack frames A, B, and C, in that order of invocation. Each stack frame has three local variables. Let us call them a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, and c_3. The variables are kept by each stack frame as follows:

- For stack frame A, every variable contains a Common Lisp object. The variables a_1 and a_2 are kept in callee-saves registers 0 and 2 respectively, so the callers values of those registers have been saved in stack locations -2 and -3 respectively, but this is the bottom frame, so those saved values contain random data. The variable a_3 is kept in a caller-saves register, so it has been saved to stack location -4 before the call that created the stack frame B.
- For stack frame B, every variable contains a Common Lisp object. The variables b_1 and b_2 are kept in callee-saves registers 1 and 2 respectively, so the callers values of those registers have been saved in stack locations -2 and -3 respectively. Saved location -3 contains a_2, but saved location -2 contains random data, because no previous invocation has used it for any Common Lisp object. The variable b_3 is kept in a caller-saves register, so it has been saved to stack location -4 before that call created the stack frame C.
- Stack frame C keeps c_1 and c_2 in callee-saves registers 0 and 2, respectively, so it has saved the callers values of those registers in stack locations -2 and -3 respectively. But only c_1 is a Common Lisp object, whereas c_2 is some immediate machine value. Saved location -2 contains a_1, and saved location -3 contains b_2. It keeps c_3 in a caller-saves register, so it has been saved to stack location -4 before the root-finding function is invoked.

This initial situation is illustrated in Figure 23.2.

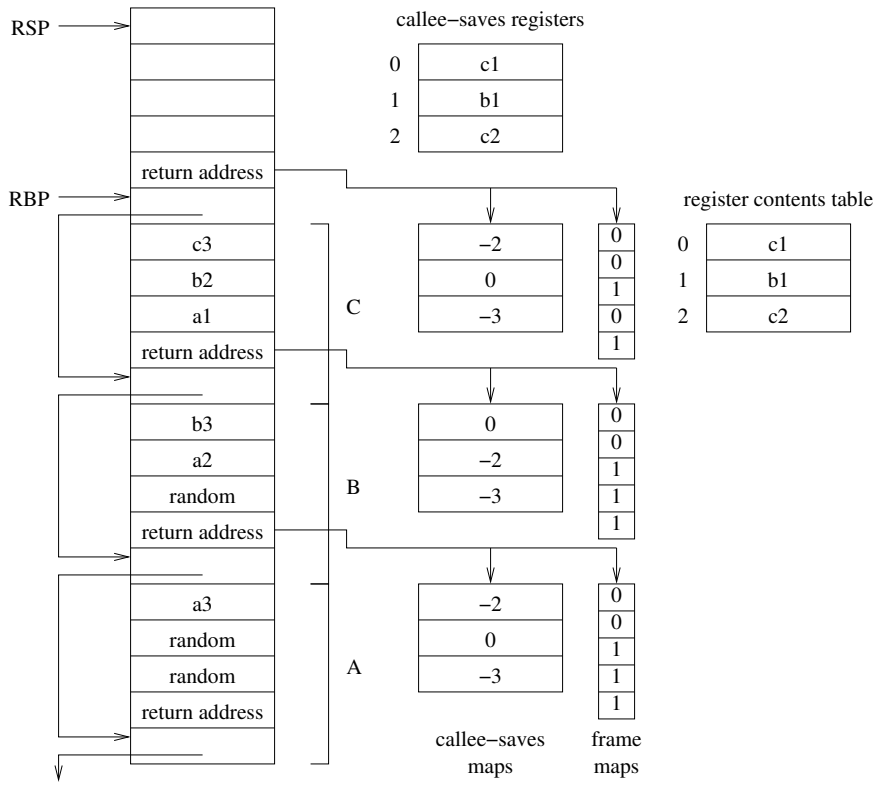

Figure 23.2: Finding roots, situation before first iteration.

In the first iteration, we start by consulting the callee-saves map. Entries that are not 0 are used to determine a stack location. The contents of that stack location is swapped with the corresponding element in the register contents table. The result of this operation is shown in Figure 23.3.

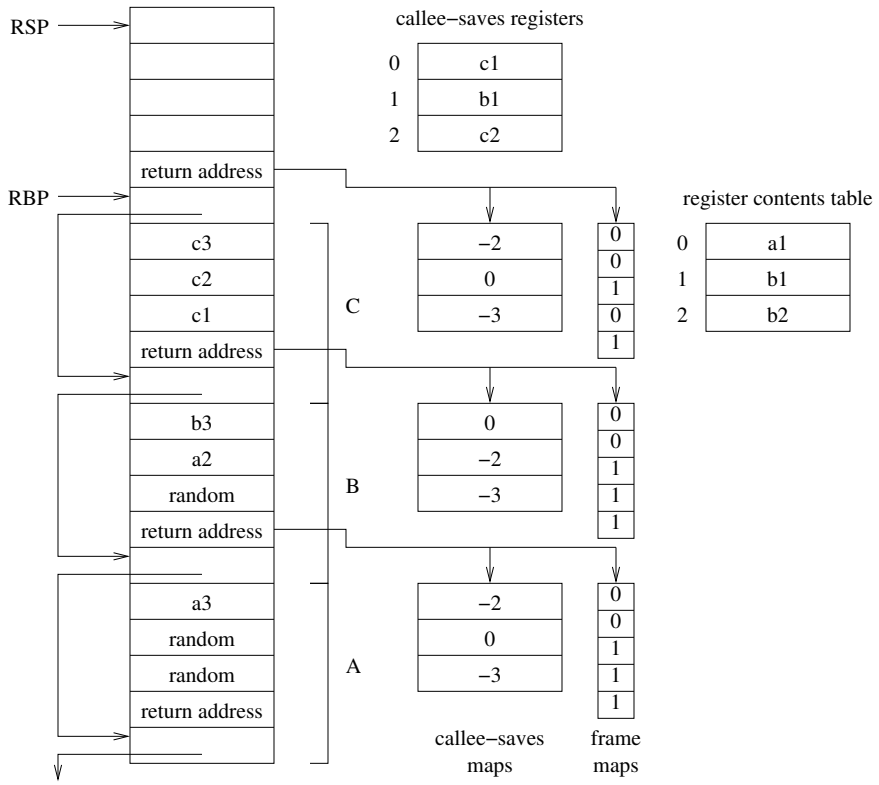

Figure 23.3: Finding roots, first iteration.

As Figure 23.3 shows, stack frame C now contains only data belonging to this invocation. The `frame map` is now consulted. It indicates that stack locations -2 and -4 contain Common Lisp objects, so c_1 and c_3 are identified as roots.

The initial situation of the second iteration is shown in Figure 23.4.

In the second iteration, we start by consulting the callee-saves map. Entries that are not 0 are used to determine a stack location. The contents of that stack location is swapped with the corresponding element in the register contents table. The result of this operation is shown in Figure 23.5.

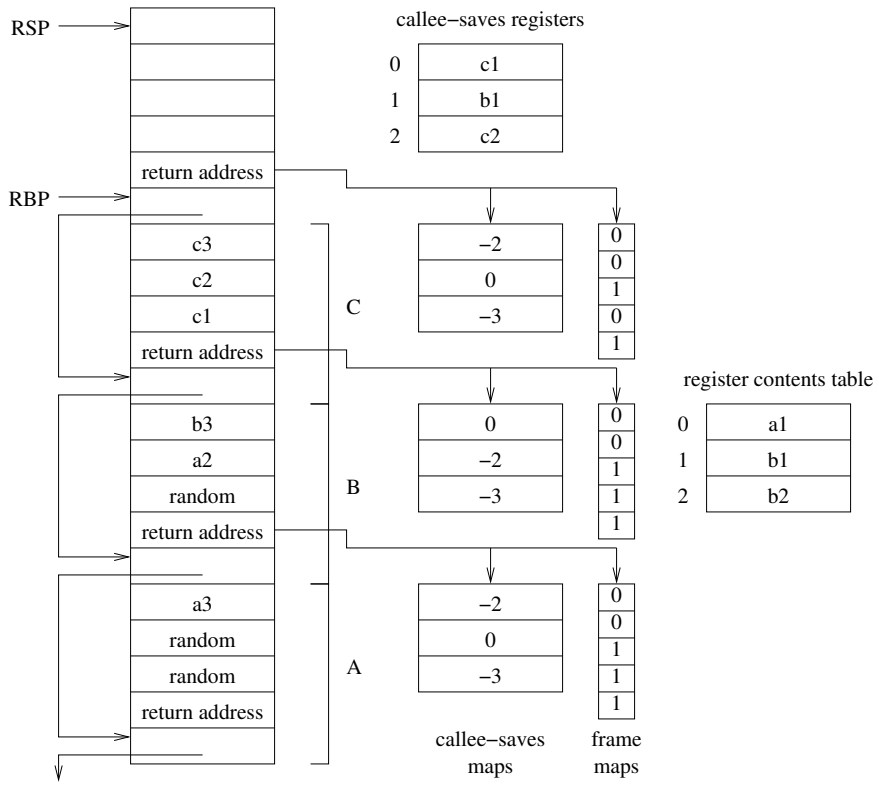

Figure 23.4: Finding roots, situation before second iteration.

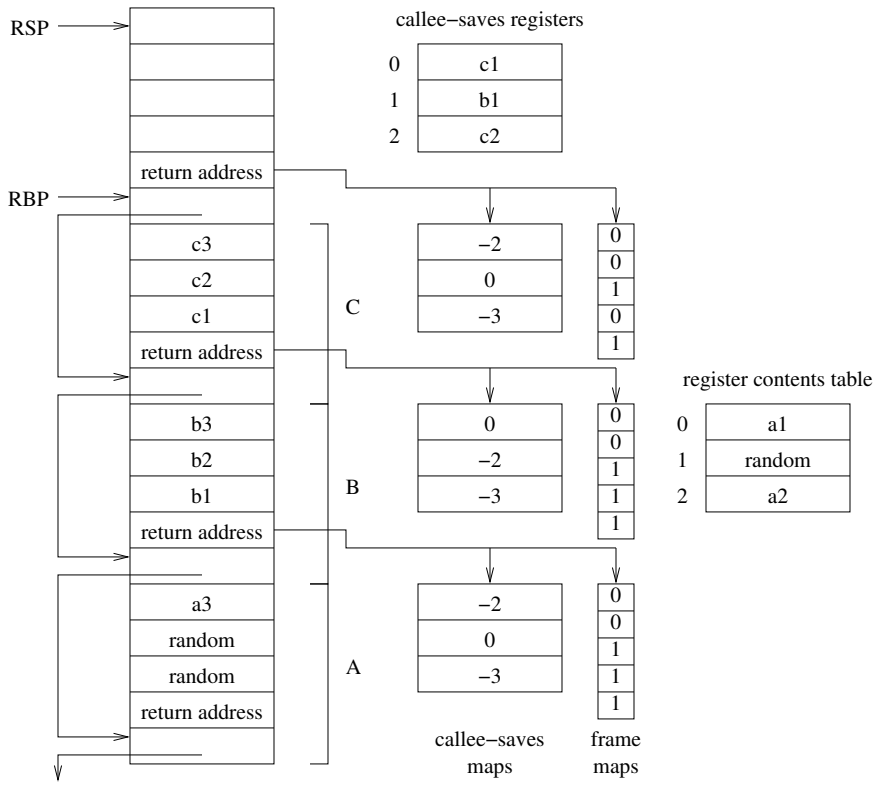

Figure 23.5: Finding roots, second iteration.

As Figure 23.5 shows, stack frame B now contains only data belonging to this invocation. The `frame map` is now consulted. It indicates that stack locations -2, -3, and -4 contain Common Lisp objects, so b_1, b_2, and b_3 are identified as roots.

The initial situation of the third iteration is shown in Figure 23.6.

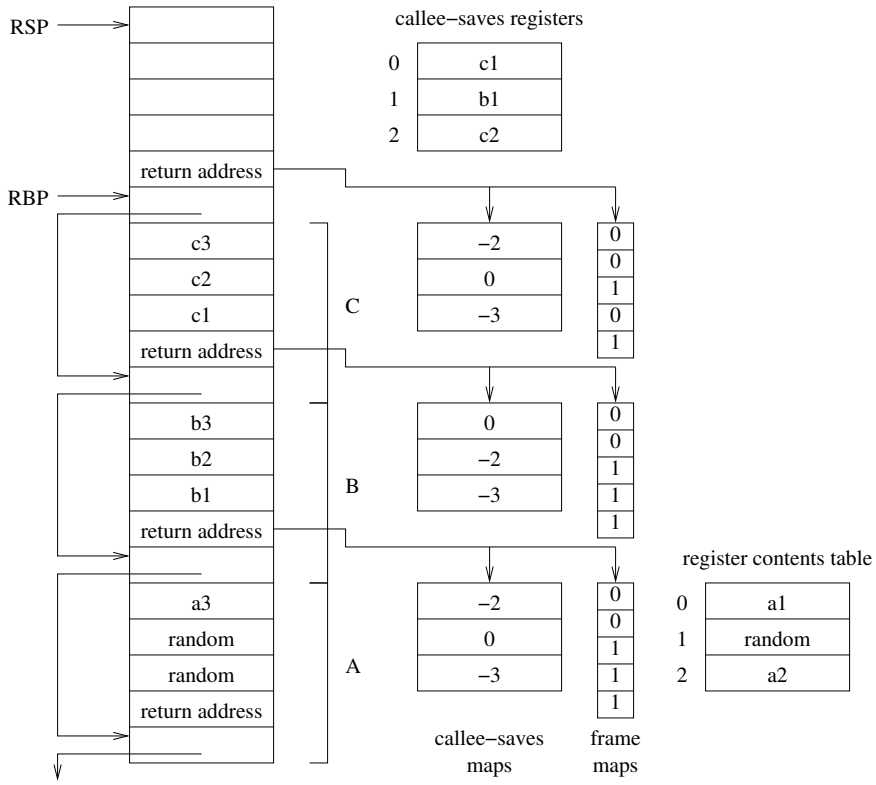

Figure 23.6: Finding roots, situation before third iteration.

In the third iteration, we start by consulting the callee-saves map. Entries that are not 0 are used to determine a stack location. The contents of that stack location is swapped with the corresponding element in the register contents table. The result of this operation is shown in Figure 23.7.

As Figure 23.7 shows, stack frame C now contains only data belonging to this invocation. The `frame map` is now consulted. It indicates that stack locations -2, -3, and -4 contain Common Lisp objects, so a_1, a_2, and a_3 are identified as roots.

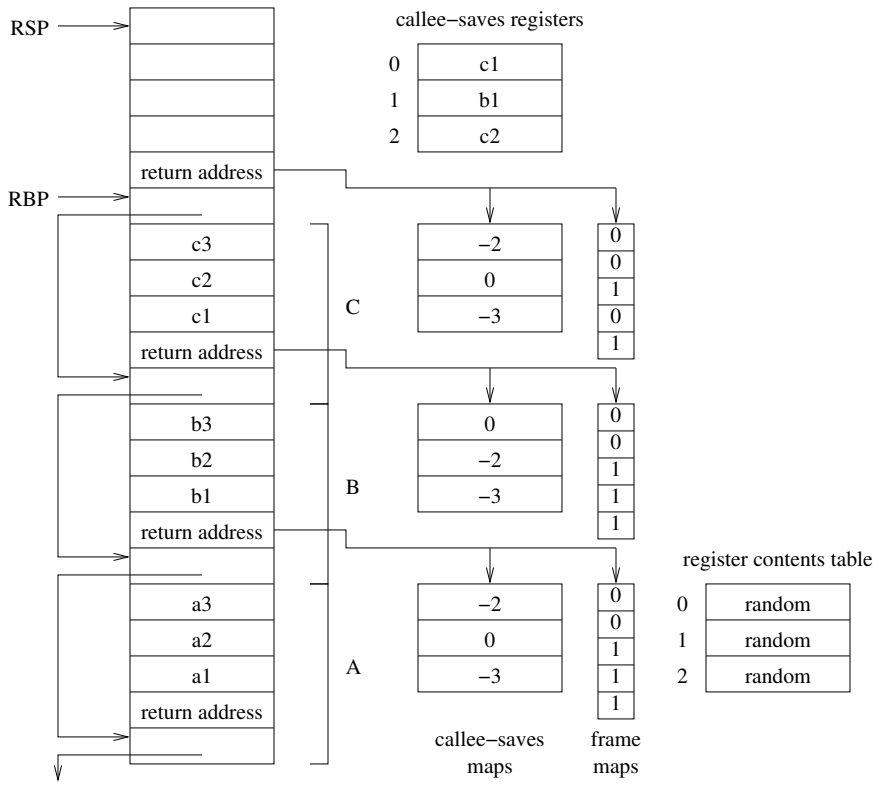

Figure 23.7: Finding roots, third iteration.

We have now reached the bottom stack frame, so we have correctly identified all the roots. To restore the stack to its initial state, this procedure must be executed in reverse order. However, the stack can be kept in this state until roots need to be found a second time.

While it is possible for the garbage collector to handle pointers into arbitrary positions of a rack,[2] in such a situation, there must also be an identifiable root that refers to the corresponding header. Otherwise, it may be the case that the object is incorrectly determined to be dead.

23.2.4 Mark phase

The mark phase uses a separate bitmap called the *live bitmap*. It has a bit for every word in the nursery. After the mark phase has finished, if a bit is set in the bitmap, it means that the corresponding word is part of a live object. The live bitmap is initialized so that all bits are cleared before the mark phase begins.

The mark phase starts by finding the roots as described in Section 23.2.3. Only pointers to dyads are reported in this phase. For every dyad found, the two bits in the live bitmap corresponding to the two words in the dyad are set to 1. The stack is scanned from top to bottom and then left in that state.

A flag determines whether a reference to an object allocated in the global heap should be marked or not. This flag is set when the local garbage collection was requested by the global garbage collector, and cleared when the local garbage collection was performed as a result of the local heap being full. If a traced object is located in the heap and the flag is set, then the global collector is informed that it is live. If the object is located in the nursery, the corresponding bits in the live bitmap are set.

Once the roots are found, the dyads marked as live in the nursery are traced according to the class of the object. If the object is a standard object (as indicated by the fact that the second word contains a rack with the tag reserved for racks (111), then the class is found in the first word of the header. The class object is recursively visited, and also consulted to determine the number of words in the rack and what words of the rack may contain pointers to heap-allocated objects. The bits in the live bitmap corresponding to the words of the rack are marked, and the words that may contain live object of the rack are recursively visited. If the object is a `cons` cell, the `car` and the `cdr` are recursively visited. Tracing stops when an object is reached that has already been traced, or when an object is reached that is located in the global heap.

[2] Recall that a pointer to a rack is tagged. The same tag is used for a pointer to any element of a rack.

The mark phase maintains an integer variable named *live space*. It is initialized to 0 at the beginning of the mark phase. For every unmarked object in the nursery that is encountered, the value of the variable is incremented by the size of that object. If the object is a standard object, then not only is the rack traced, but the size of the rack is added to the value of the variable as well. When the mark phase is finished, this variable contains the total amount of live space in the nursery.

At the end of the mark phase, if the flag is set, a *signal* operation is performed on a semaphore used for synchronization between the local and the global garbage collector.

23.2.5 Promotion phase

After the mark phase has finished, the value of the variable *live space* is used to determine whether some of the objects in the nursery should be *promoted*. Good threshold values are yet to be determined, but we think that if more than half the nursery contains live objects, then some objects should be promoted. The higher the threshold, the more likely it is that a collection will be triggered soon after the current one. The lower the threshold, the more likely it is for young objects to be promoted even though they are likely to die soon.

The other threshold value to be determined is how many objects should be promoted. We think that, after promotion, around one fourth of the nursery should contain live objects. The higher the threshold, the more likely it is for another promotion to be triggered during the next invocation of the collector. The lower the threshold, the more likely it is for young objects to be promoted even though they are likely to die soon.

The promotion phase uses a bitmap called the *promotion bitmap* again containing one bit per word in the nursery. This bitmap is initialized so that all its bits are cleared. When an object has been promoted, the first word of the dyad contains a *forwarding pointer* pointing to the promoted object in the global heap, and the bit in the promotion bitmap corresponding to the first word of the dyad is set. Similarly, when a standard object has been promoted, *every word* of the rack contains a forwarding pointer, pointing to the new word in the global heap, and the bit in the promotion bitmap corresponding to that word is set. The reason for treating the rack this way is that a stack frame may contain local variables that are rack pointers, and those variables must be updated when the object owning the rack has been promoted. We do this by replacing the contents of the variable with the forwarding pointer when the corresponding bit in the promotion bitmap is set.

The promotion phase executes a loop over the live objects that have not yet been promoted in the nursery, starting from the one with the smallest address. An object

is skipped if it not live, as indicated by the bit in the live bitmap, and an object is skipped if it has already been promoted, as indicated by the bit in the promotion bitmap. If the object is not to be skipped, it is promoted. This means that a *copy* of the object is allocated in the global heap. A forward reference is stored in the first word of the dyad of the original object, and a bit is set in the promotion bitmap. Similarly, the contents of every word of the rack is replaced by a forwarding pointer, and the bit in the promotion bitmap corresponding to each such word is set. Notice that, during this phase, there will be references from the global heap to the nursery, but these references will never be followed by the global collector, and they will disappear at the end of the promotion phase.

The newly promoted *copy* is then traced in much the same way as objects are traced during the mark phase. An object is traced by recursively visiting its contained objects. When a visited object reference indicates that it has already been promoted, as indicated by the corresponding bit being set in the promotion bitmap, the reference is replaced by the forward reference stored in the first word of the dyad in the nursery, and tracing stops. If the visited object has not yet been promoted, then a copy is allocated in the global heap as described above. The corresponding bits in the promotion bitmap are set and forwarding pointers are stored, again as above, and the copy is recursively traced. Notice that if a functional programming style is used, all objects referred to by the initial copy will have already been promoted. Only side effects can result in an object containing a reference to an object that was allocated later on in the nursery. Therefore, if a functional style is used, the recursion will be very shallow. For every object that is being promoted, we keep a tally of the amount of total space that has been promoted this way. When the total amount of space is greater than or equal to the threshold that has been determined, the outer loop stops. We can not, however, stop the tracing in each iteration, because stopping it prematurely means that there will be references from the global heap to the nursery remaining after this phase.

When a sufficient number of objects have been promoted, the stack is traversed to find roots that need to be modified as a result of object having been promoted. However, since the stack is now in a state where every local variable is stored in the stack frame that owns it as Section 23.2.3 explains, we only need to consult the frame map of each frame to determine the roots. Whether the root contains a pointer to a dyad or to some element of a rack, the promotion bitmap for that pointer is consulted. If it is set, then the root is replaced by the forwarding pointer stored in the dyad or the rack.

Finally, every live object in the nursery that has not been promoted is scanned to determine any reference to a promoted object. Such a reference is replaced by the forward reference as before.

When every relevant reference to a promoted object has been updated this way, the

promotion bitmap is subtracted from the live bitmap. This way, the live bitmap has a bit set only for live objects that are still in the nursery.

23.2.6 Compaction phase

In the *compaction phase*, the live bitmaps is used to slide dyads to the beginning of the nursery heap and to slide racks to the end of the nursery heap. A *source* pointer follows live words to be copied, and a *target* pointer follows available words.

23.2.7 Break table build phase

In the *table build phase*, the live bitmap is used to construct two *break tables* between the new locations of the dyad free pointer and the rack free pointer after compaction. Each table has the format $a_0, d_0, a_1, d_1, \ldots, a_n, d_n, a_{n+1}$ where a_i is an address and d_i is a *delta* or an *offset*. The meaning of the elements of a table is that an address that was originally between a_i and a_{i+1} should be updated by having d_i added to it. One break table is built for dyads and another break table is built for racks. In the worst case, each of these tables may contain three more words than there are free words available for it. This is the reason why a collection is triggered when granting a request for allocation would result in fewer than six words left in the nursery heap.

We use *binary search* to find an entry in a break table.

23.2.8 Pointer adjustment phase

Once the break tables are built, the stack is scanned to find roots, and it is also restored to its initial state as described in Section 23.2.3. For each root that contains a pointer to the nursery, we determine whether it is pointer to a dyad or to a rack, and consult the corresponding break table to determine a value to be added to the current one. Notice that we can handle pointers to any arbitrary position in a rack. The corresponding break table will indicate the correct amount for such internal pointers as well.

Next, dyads in the nursery are traversed. For a `cons` cell, the first break table is consulted in order to adjust both the `car` and the `cdr`. For a standard object, the first break table is consulted in order to adjust the `class` slot and the second break table is consulted in order to adjust the `rack` slot. Once the rack slot has been adjusted, the rack is traversed according to the class in order to determine what words of the rack

contain Common Lisp objects. For each such word, if it is a heap-allocated object, the first break table is consulted in order to adjust its value.

In the *adjust phase*, dyads and racks are scanned, and fields containing pointers are adjusted according to the offset table. The offset table is searched using binary search, except that a simple caching scheme is used to avoid a full binary search in nearly all cases.

23.3 Synchronization between collectors

Since each thread is responsible for collecting its own heap and since the global collector can not run until every application thread has run its own collector, we need to find a way of dealing with threads that are stopped for any reason, for instance waiting for input/output.

We think that in cases like that, one of the threads of the global collector would run the garbage collector on behalf of the stopped thread. During the execution of the garbage collector, if the stopped thread becomes unstopped, it must then be prevented from running application code until the collection has run to completion.

We do this by introducing a variable and two semaphores for each application thread. The variable and the semaphores are shared between that thread and the global collector threads. The variable has 4 bit positions with the following meanings:

- `application-blocked` meaning that the application thread might be blocked when the bit is set, and is therefore not capable of executing the local garbage collector. This bit is both set and cleared by the application thread.
- `gc-requested` meaning that the global collector has requested that the application thread run the garbage collector. This bit is set by the global collector and cleared by the application thread.
- `gc-in-progress` meaning that the global collector is running the nursery garbage collector on behalf of the thread. This bit is both set and cleared by the global collector.
- `application-waiting` meaning that the application is waiting for the global collector to finish the garbage collection on behalf of the application thread.

The two semaphores work as follows:

- `global-gc-may-proceed`. This semaphore is initialized to 0. When the global collector requests that an application thread run the local garbage collector

it waits on this semaphore. The application thread signals this semaphore, indicating that all objects in the global heap reachable from the local heap have been marked.

- `application-may-execute`. This semaphore is initialized to 0. When an application thread awakes after having been blocked, it checks whether the global collector is currently executing on behalf of this application thread. If so, the application thread waits on this semaphore. When the global collector has finished executing on behalf of this thread, it signals this semaphore, allowing the application thread to continue its execution.

23.3.1 Running application thread

At every safe point (function call, function return, and a branch to an inferior address[3]), the application thread consults the shared variable. If the `gc-requested` bit is set, then it clears that bit and runs the nursery garbage collector. At the end of the collection, it signals the `global-gc-may-proceed` semaphore indicating to the global collector that it has finished, and finally continues the application thread. No synchronization is required to read the shared variable, because once the `gc-request` bit is set, no other bit is going to move as the result of any action on the part of the global collector.

23.3.2 Application thread about to block

When there is a possibility that an application thread is about to block, for example when it is about to execute some input or output operation, it must inform the global collector that it might be unable to run the local garbage collector itself, and that the global collector may have to run it on behalf of the application thread.

The application starts by reading the shared variable into an ordinary lexical variable. If the `gc-request` bit is set, it clears the bit and runs the nursery garbage collector. At the end of the collection, it signals the `global-gc-may-proceed` semaphore indicating to the global collector that it has finished, and then starts over by reading the variable again. This action is repeated until the `gc-request` bit is cleared.

When the `gc-request` bit is cleared, it sets the `application-blocked` bit in a second local copy of the shared variable. It then performs a CAS operation to set the `application-blocked` bit in the shared variable. Should this operation fail, then it

[3] To minimize the overhead in case of a very tight loop, we will use loop unrolling so that a branch to an inferior address will be less frequent.

means that the `gc-request` bit has been set since the shared variable was read the first time. The application thread then starts the entire operation over again.

If the CAS operation succeeds, it means that the global collector has been informed that, if it needs for this nursery heap to be garbage collected, it has to do it on behalf of the thread. The application thread can now perform the operation that might block the thread.

23.3.3 Application thread waking up after block

When the application thread wakes up after having been blocked, there is a possibility that the global garbage collector is in the process of running the nursery collector on behalf of the application thread. If that is the case, then the application thread must wait until the garbage collection is finished.

The application thread starts by reading the shared variable into an ordinary lexical variable, and it also makes a second copy. If the `gc-in-progress` bit is cleared, it then clears the `application-blocked` bit in the second copy and performs a CAS operation. If the CAS operation succeeds, this means that the global collector is not running on behalf of the application thread, and will not do so because it has been informed that the application thread is not blocked. The application thread then returns to its normal operation without any further action.

If the CAS operation fails, then that means that the global collector has started a collection on behalf of the application thread since the shared variable was read. Then the application thread reads the shared variable again. If the `gc-in-progress` bit is set, then the application thread clears the `application-blocked` bit and sets the `application-waiting` bit in the second local copy of the shared variable. It then performs a CAS operation. If the CAS operation succeeds, it means that the global collector is still running the nursery collector on behalf of the application thread, and that the global collector has been informed that, when the nursery collection is finished, it should signal the `application-may-execute` semaphore.

23.3.4 Preparing for a global collection

Before a global collection can take place, each nursery must first be collected, and references from the nurseries to the global heap must be marked so that the global collector will keep the referenced objects.

For each application thread, the following actions are performed:

The contents of the shared variable is read into an ordinary lexical variable, and a second copy of it is made. The `application-blocked` bit is examined.

1. If it is cleared, then the `gc-request` bit is set in the second copy, and a CAS operation is attempted.

   (a) If the operation succeeds, then that means that the application thread is still not blocked and it has been properly informed that it is expected to run a nursery collection. The `global-gc-may-proceed` semaphore for this application thread is added to a set of such semaphores.

   (b) If the operation fails, then that means that the application has been blocked in the meantime. The entire operation is then restarted by reading the shared variable again.

2. If it is set, then the application is blocked. Then the `gc-in-progress` bit of the second copy is set, and a CAS operation is attempted.

   (a) If the operation succeeds, then that means that the application thread is still blocked, and that it has been properly informed that a nursery collection on its behalf is pending. A new thread (or an existing one from a pool) is assigned to do a nursery collection on behalf of the application thread, and the thread is added to a set of such threads.

   (b) If the operation fails, then that means that the application is no longer blocked. The entire operation is then restarted by reading the shared variable again.

When all the application threads have been processed this way, the global garbage collector executes a *wait* operation sequentially on each semaphore in the saved set.

Then the global collector waits for each the threads in the set of threads doing a nursery collection on behalf of an application thread to finish. When such a thread finishes, the shared variable is read into an ordinary lexical variable and a second copy is made. The `application-waiting` bit is examined.

1. If it is set, the global collector clears the bit, clears the `gc-in-progress` bit, and executes a *signal* operation on the `application-may-execute` semaphore. No further synchronization is required.

2. If it is cleared, then that means that the application is still blocked. Then the global collector clears the `gc-in-progress` bit in the second copy and attempts a CAS operation.

(a) If the CAS operation succeeds, then it means that the operation is still blocked, and it has been informed that it can safely execute application code when it wakes up.

(b) If the CAS operation fails, then the application may no longer be blocked. The entire operation is then restarted by reading the shared variable again.

23.4 Implementation

In most systems, the garbage collector is implemented in some language other than Common Lisp. However, we imagine using Common Lisp together with some additional low-level primitives for accessing memory by address instead.

Chapter 24

Debugger

Part of the reason for SICL is to have a system that provides excellent debugging facilities for the programmer. The kind of debugger we plan to support is described in a separate repository.[1] In this chapter, we describe only the support that SICL contains in order to make such a debugger possible.

The execution of every function starts by testing a *flag* passed in a register.[2] This flag indicates whether the current thread is being debugged. The function contains two versions of the code, called the *normal* version and the *debugging* version.[3] The flag determines which version is chosen. Thus, when the thread is not being debugged, the only overhead is this single test at the beginning of the function. Furthermore, once the test is done, the register is no longer needed for this purpose, and is at the disposal of the register allocator for the remainder of the function body.

The normal version is used when the thread is not run under the control of the debugger, so this version does not contain any code for communicating with the debugger. Furthermore, this version is highly optimized. In particular, variables that occur in the source code may have been eliminated by various optimization passes. A function call in the normal version clears the flag register, so that the callee can choose its normal version as well.

The debugging version starts by examining a special variable that contains information about the current thread. In particular, this information includes a table in the form of a bit vector containing summary information about breakpoints. In this version

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

[2] For the x86-64 platform, it is register RAX.

[3] This idea was suggested by Michael Raskin.

of the function body the compiler inserts a call to a small subroutine before and after every form to be evaluated. The subroutine does not use the full Common Lisp function-call protocol. Instead, it is just a very fast call that can be done with a `jsr` instruction (or equivalent) on must architectures.

The subroutine does a test in two steps. In the first step, the value of the program counter is taken modulo some reasonably large value such as 256, and a the bit vector is queried to see whether the corresponding entry is a 1. If it is 0, the subroutine simply returns. This first step will slow down every debugged thread a little bit, but most of the time, the value will be 0, and then, normal function execution is resumed.

If the entry in the bit vector turns out to be 1, then the final test is made. The program counter is checked against a hash table in the thread instance to see whether some action must be taken. If so, the thread gives up control to the debugger.

A function call in the debugging version sets the flag register to 1, so that the callee can choose its debugging version as well. The debugging version does not have optimizations applied to it that may make debugging harder. Lexical variables that appear in source code may be kept, or code may be included that can compute their values from the lexical variables that *are* kept, for the duration of their scope.

Chapter 25

Processing arguments

In this chapter, we describe how processing arguments is accomplished by inserting HIR instructions immediately after HIR code is generated from an abstract syntax tree. By doing it this way, we obtain several advantages:

- We simplify the translation of HIR code to LIR later on the translation process.
- HIR transformations such as constant hoisting and `fdefinition` hoisting can be applied to the argument-processing code, thereby simplifying this code.
- The HIR instructions introduced are subject to various HIR transformations such as value numbering, constant propagation, etc.

Each type of parameter is is handled by a different module. In addition, because of the complexity of initializing keyword parameters, that module is is further subdivided.

Two new HIR instructions are used in order to accomplish the argument processing:

- The `compute-argument-count-instruction` has no inputs, and a single output. It is responsible for computing the total number of arguments passed to the function.
- The `argument-instruction` has one input and one output. The input is a datum that must be a non-negative fixnum. The output is the argument with an index represented by the value of the input, starting at 0 for the first argument.

The overall organization of the modules for initializing parameters is shown in Figure 25.1.

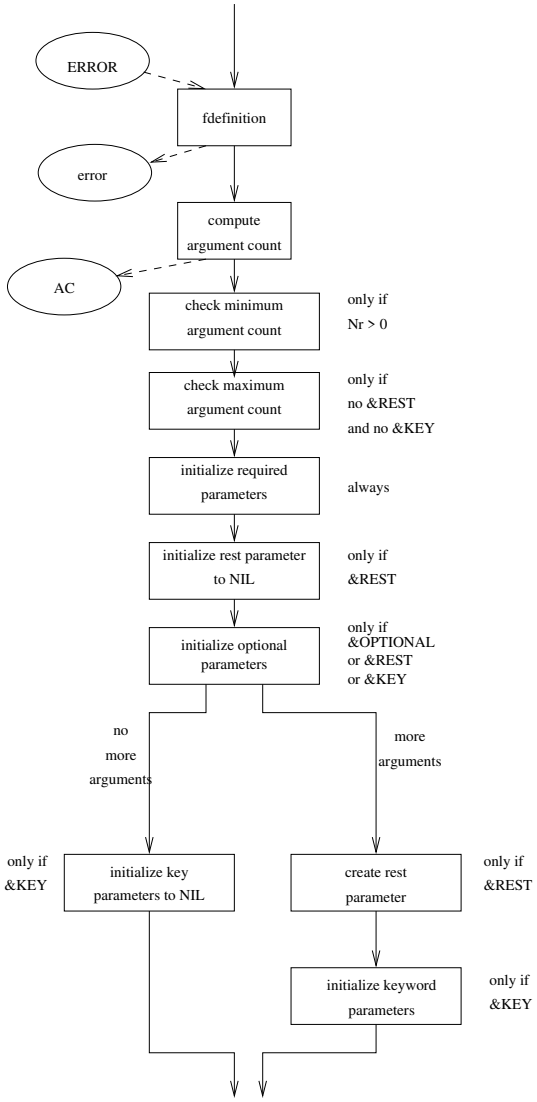

Figure 25.1: Processing all arguments.

25.1 Calling `error`

To minimize the clutter in many of the figures in this chapter, when an error situation is detected, a box labeled "error" is shown. But this box is slightly more complicated than a single instruction. The complete sequence of instructions is shown in Figure 25.2.

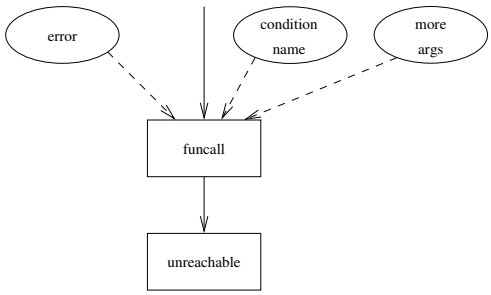

Figure 25.2: Calling `error`.

As Figure 25.2 shows, the sequences consists of two instructions.

The first instruction is a `funcall-instruction` that calls the function `error` as computed by the first instruction in Figure 25.1. This instruction takes more arguments to be passed to `error`. The first additional argument is a constant input containing the name of the condition class to signal. Remaining additional arguments are initialization arguments for the particular condition class.

The second instruction is an `unreachable-instruction`. An additional instruction is required, because the `funcall-instruction` has a single successor. The `unreachable-instruction` indicates that control can not reach this point. This instruction has no successors.

25.2 Checking the minimum argument count

Unless there are no required parameters, there is a minimum allowed value for the argument count, and it is equal to the number of required parameters. The HIR code for this check is simple and is illustrated by Figure 25.3.

As figure Figure 25.1 shows, this test is only inserted when the number of required parameters is strictly greater than 0. The input labeled "Nr" in Figure 25.3 is a constant

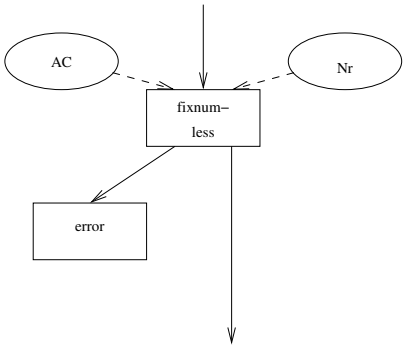

Figure 25.3: Checking minimum argument count.

input (known at compile time) representing the number of required parameters, and the input labeled "AC" is the lexical location computed by the `compute-argument-count` instruction in Figure 25.1.

25.3 Checking the maximum argument count

Unless there is a `&rest` parameter or there are `&key` parameters, there is a maximum allowed value for the argument count, and it is equal to the sum of the number of required parameters and the number of optional parameters. The HIR code for this check is also simple, and is illustrated by Figure 25.4.

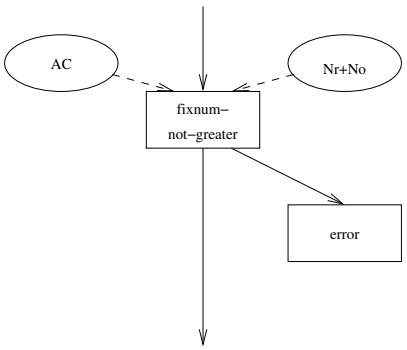

Figure 25.4: Checking maximum argument count.

As figure Figure 25.1 shows, this test is only inserted when there is no `&rest` parameter and no `&key` parameters. The input labeled "Nr+No" in Figure 25.4 is a constant input (known at compile time) representing the number of required parameters plus the number of optional parameters, and the input labeled "AC" is again the lexical location computed by the `compute-argument-count` instruction in Figure 25.1.

25.4 Initializing required parameters

Figure 25.5 illustrates the HIR code for initializing required parameters.

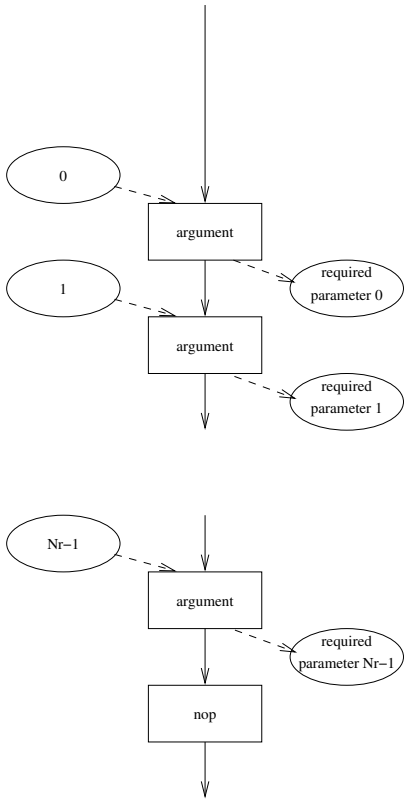

Figure 25.5: Initializing required parameters.

The number of stages is determined at compile time and is equal to the number of

required parameters. For each parameter, the next argument (starting with 0) is assigned to that parameter.

As Figure 25.1 suggests, this procedure can be applied even when there are no required parameters. In this case, the result consists only of the final `nop-instruction`.

25.5 Initializing optional parameters

Figure 25.6 illustrates the HIR code for initializing optional parameters.

As Figure 25.6 shows, there are two main branches in this code, the *left* branch and the *right* branch. The control flow starts in the *right* branch, and continues in that branch if until there are no more arguments. At that point, control is transferred to the *left* branch where the remaining optional parameters are initialized to `nil`.

As Figure 25.6 also shows, the code ends with a test to check whether there are no more argument, even though this test is not required in order to decide how any more optional parameters should be initialized (because there are no more at that point). The reason for the existence of this test is so that the left branch (no more arguments) can be used to determine whether all keyword parameters should be initialized to `nil`, as illustrated by Figure 25.1.

As with the required parameters, this procedure can be applied even when there are no optional parameters. In this case, it degenerates into the last test, i.e. it determines whether there are more arguments than required parameters.

25.6 Initializing keyword parameters to `nil`

When there are no more arguments after all the required and all the optional parameters have been initialized, then, if there are keyword parameters, then they can all be initialized to `nil`. This procedure is shown in Figure 25.7.

As Figure 25.1 shows, this HIR code is added only if the lambda list contains `&key` parameters.

25.7 Creating the `&rest` parameter

Figure 25.8 illustrates how the `&rest` parameter is created.

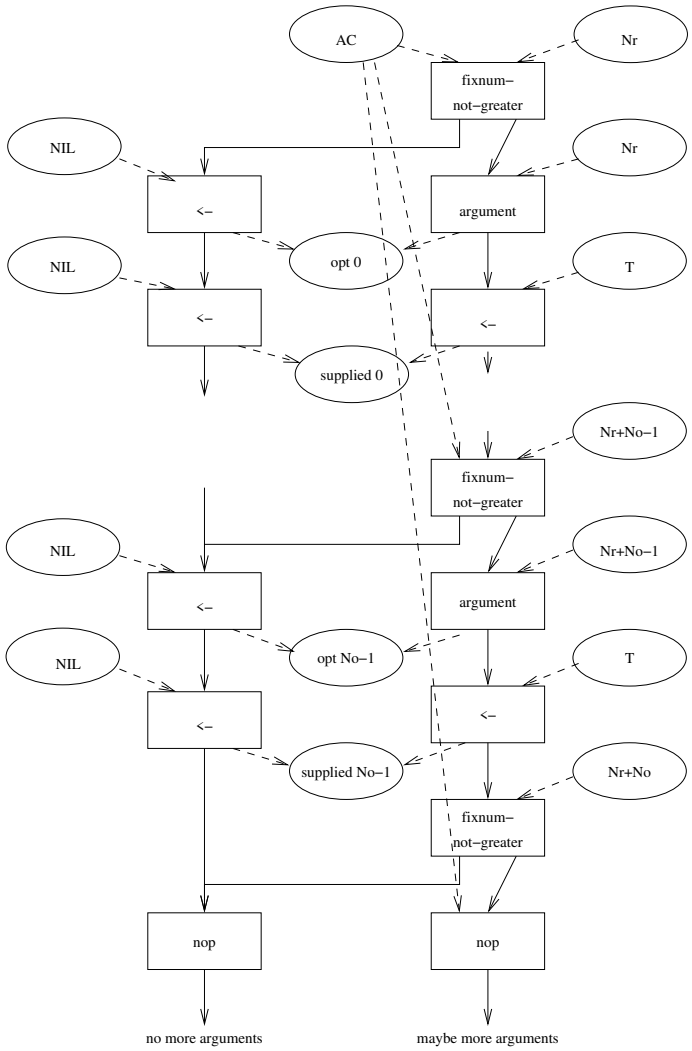

Figure 25.6: Initializing optional parameters.

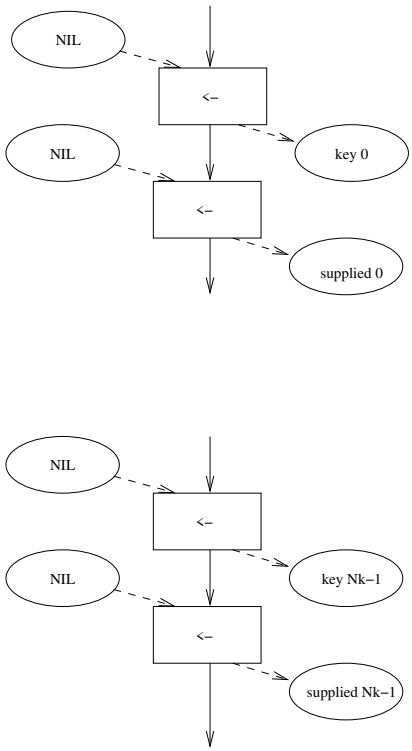

Figure 25.7: Initializing keyword parameters to `nil`.

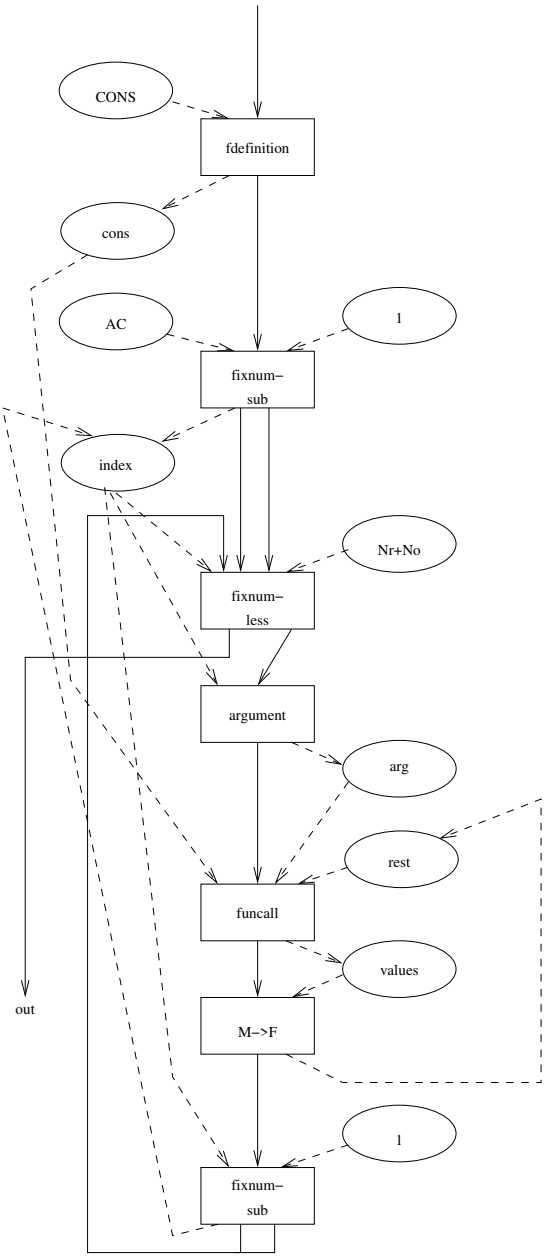

Figure 25.8: Creating the rest parameter.

As Figure 25.8 shows, there is a loop that starts with the last argument and ends with the first remaining argument after the required and the optional parameters have been initialized. In each iteration of the loop, the function `cons` is called in order to add another argument to the beginning of the list.

As shown in Figure 25.1, the `&rest` parameter, if present, has already been initialized to `nil` before this HIR code is executed.

25.8 Initializing keyword parameters

Figure 25.9 illustrates how keyword parameters are initialized.

As Figure 25.9 shows, we subdivide the initialization of keyword arguments further. We start by checking that there is an even number of keyword argument. After that, each keyword parameter is initialized in turn by a separate traversal of the remaining arguments. Finally, if required, we check whether the keyword `:allow-other-keys` is present in the argument list, and if so, what value it has. We end the process by checking the validity of each keyword argument, again only if required.

25.8.1 Checking that the number of arguments is even

Figure 25.10 illustrates the technique for checking that there is an even number of arguments to be used for initializing keyword parameters. Recall from Figure 25.1 that this code is executed only when there are more arguments than the sum of the number of required and the number of optional parameters.

As Figure 25.10 shows, we simply subtract the sum of the number of required and the number of optional parameters from the argument count, and then take the remainder when this difference is divided by 2. If that remainder is 0, then we have an even number of remaining arguments. Otherwise, we call `error` with an appropriate condition type.

25.8.2 Initializing a single keyword parameter

Figure 25.11 illustrates how a single keyword parameter is initialized.

As Figure 25.11 shows, the essence of the code consists of a loop over the remaining arguments, starting from the beginning. Each time around the loop, 2 is added to the index.

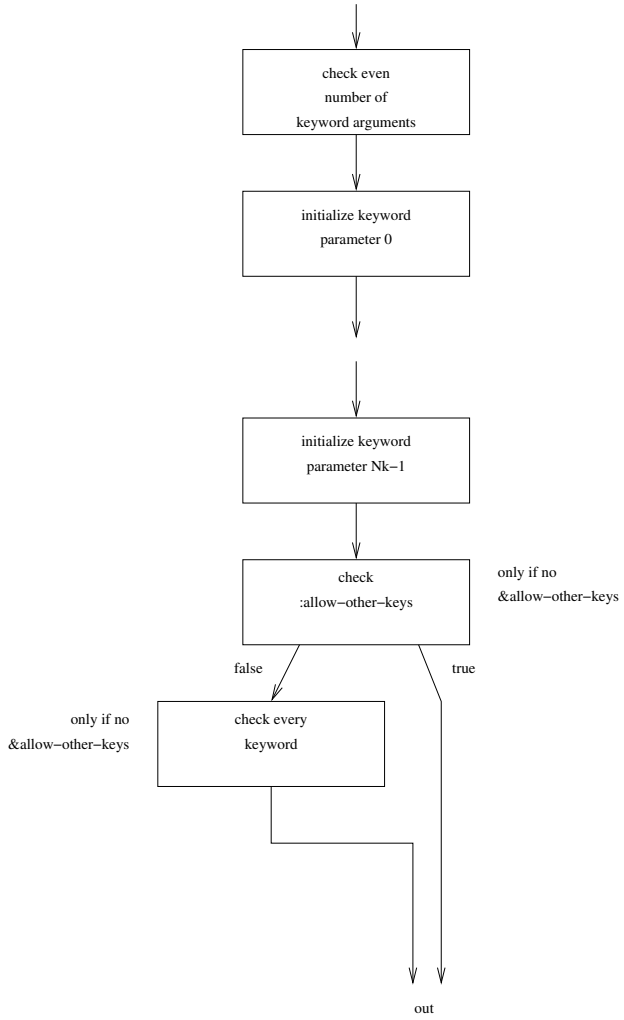

Figure 25.9: Processing keyword arguments.

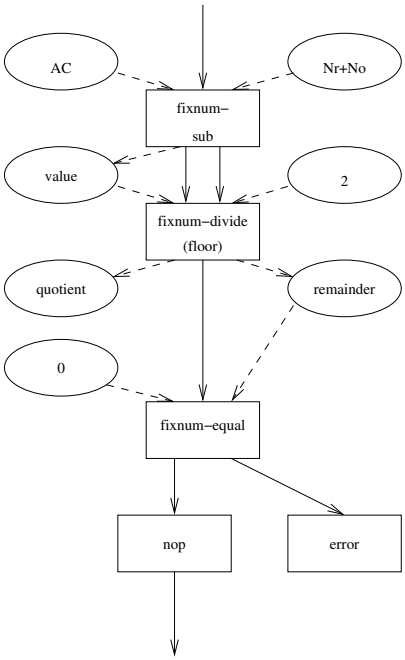

Figure 25.10: Checking that there is an even number of keyword arguments.

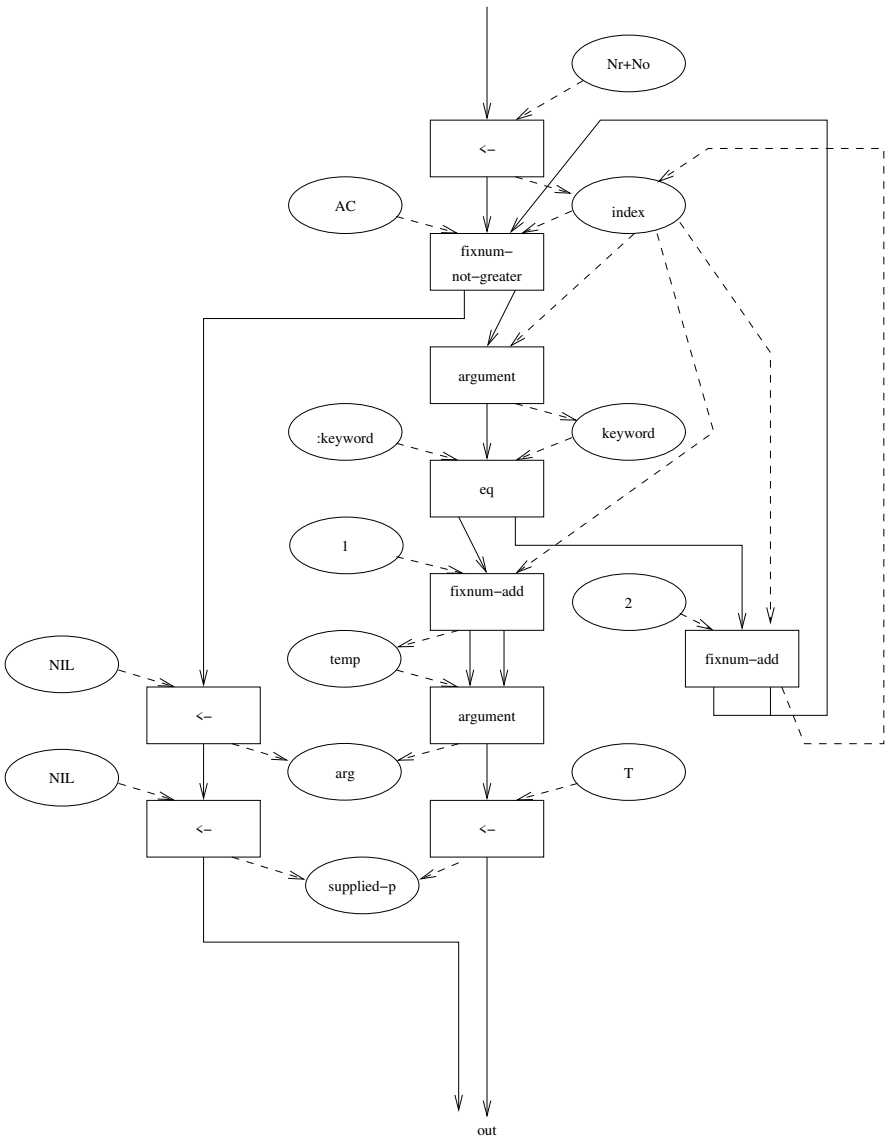

Figure 25.11: Initializing one keyword parameter.

When an `eq` comparison between the argument a the constant symbol representing the keyword in question yields true, then the loop is exited, and the argument immediately following the keyword is used to initialize the parameter, and the constant value `t` is used to initialize the `supplied-p` parameter.

If the loop reaches the end without the `eq` comparison yielding true, `nil` is used to initialize both the keyword parameter and the `supplied-p` parameter.

25.8.3 Checking the presence of `:allow-other-keys`

Figure 25.12 illustrates how the check is made whether the keyword argument `:allow-other-keys` is present, and if so, whether it has a true value. As Figure 25.9 shows, this check is only performed when the lambda list does not have the lambda-list keyword `&allow-other-keys` in it.

The way the HIR code in Figure 25.12 works, is similar to the way the HIR code of fig-initialize-one-keyword-parameter for initializing a single keyword parameter works. The difference is that there is no `suppolied-p` parameter, and the value of the argument is not kept. Instead, the value, if the keyword is present, is just compared for equality to `nil`. If the keyword is not present, or if the value is `nil`, then the left branch is chosen. Otherwise, the right branch is chosen.

Notice that the semantics correspond to that of ordinary keyword arguments, in that if there are several occurrences of the keyword in the argument list, then it is the value of the first occurrence that determines the result.

25.8.4 Checking the validity of every keyword

The final step of processing keyword arguments is to verify that each keyword given is valid. This step is illustrated by Figure 25.13.

As shown in Figure 25.1, this step is executed only when the lambda list keyword `&allow-other-keys` is absent from the lambda list, *emph* either the keyword `:allow-other-keys` is absent from the arguments or it has the value `nil`.

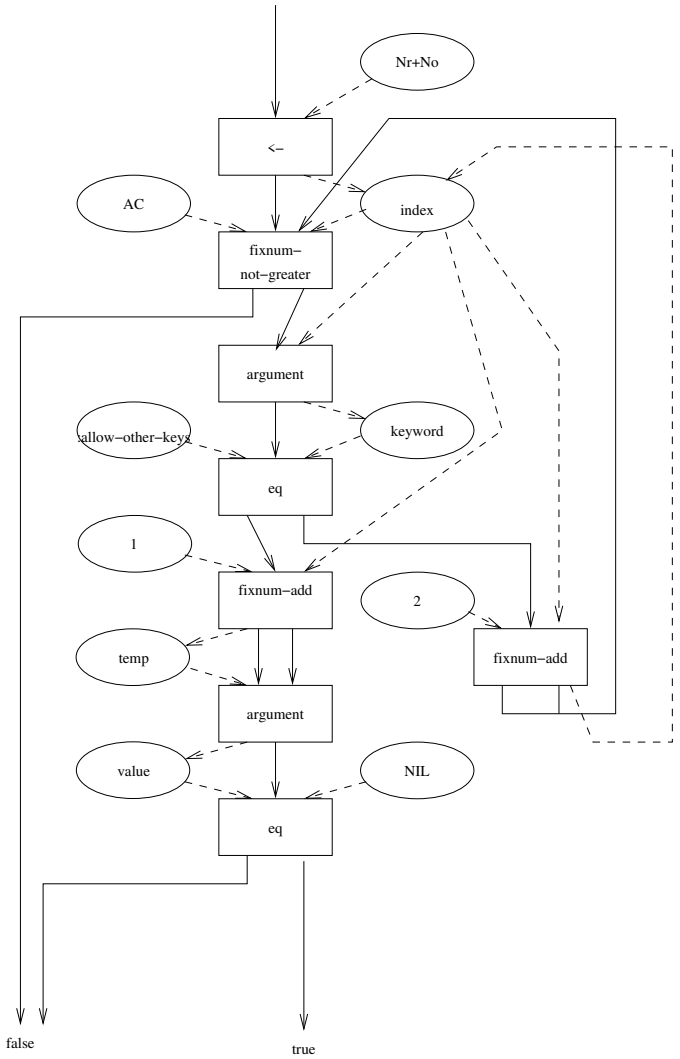

Figure 25.12: Checking for `:allow-other-keys`

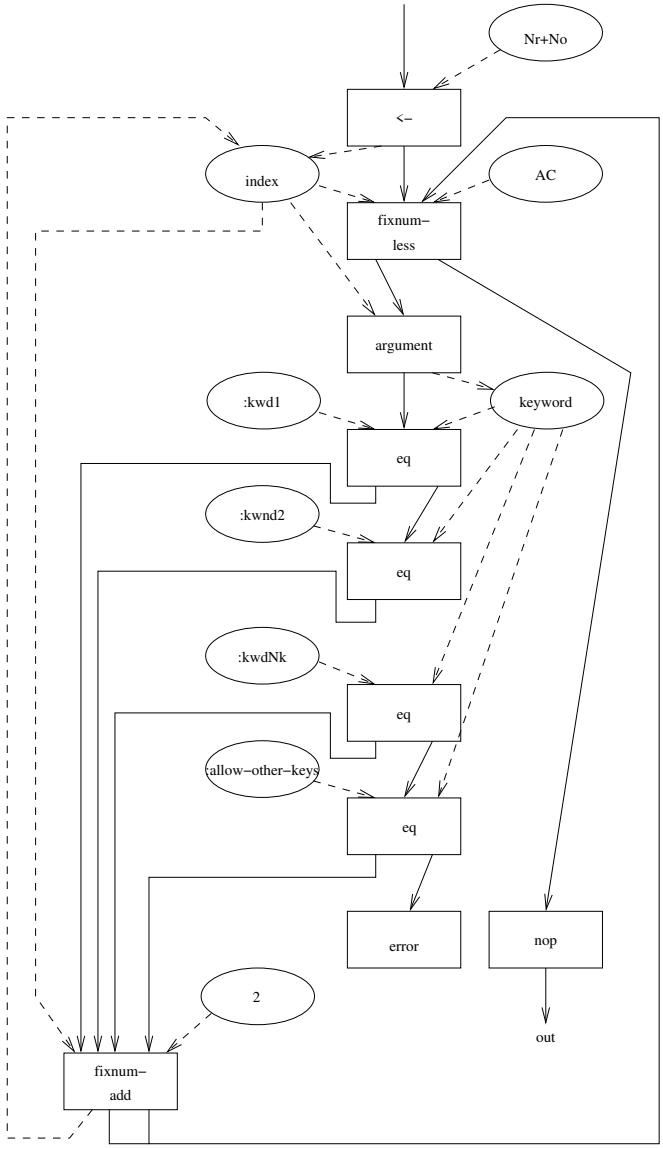

Figure 25.13: Checking the validity of every keyword.

Chapter 26

Processing return values

In this chapter, we describe how processing return values is accomplished by inserting HIR instructions immediately after HIR code is generated from an abstract syntax tree. As with the code for processing arguments, by doing it this way, we obtain several advantages:

- We simplify the translation of HIR code to LIR later on the translation process.
- HIR transformations such as constant hoisting and `fdefinition` hoisting can be applied to the code for processing return values, thereby simplifying this code.
- The HIR instructions introduced are subject to various HIR transformations such as value numbering, constant propagation, etc.

26.1 Replacing the `multiple-to-fixed-instruction`

Clients that would like to use this technique for processing return values would have a HIR transformation that replaces the `multiple-to-fixed-instruction`. Recall that the `multiple-to-fixed-instruction` accesses the distinguished location for multiple values and copies the values in that location to individual lexical locations that make up the outputs of the instruction.

The code that replaces the `multiple-to-fixed-instruction` is shown in Figure 26.1.

In Figure 26.1, N is the number of outputs of the `multiple-to-fixed-instruction`.

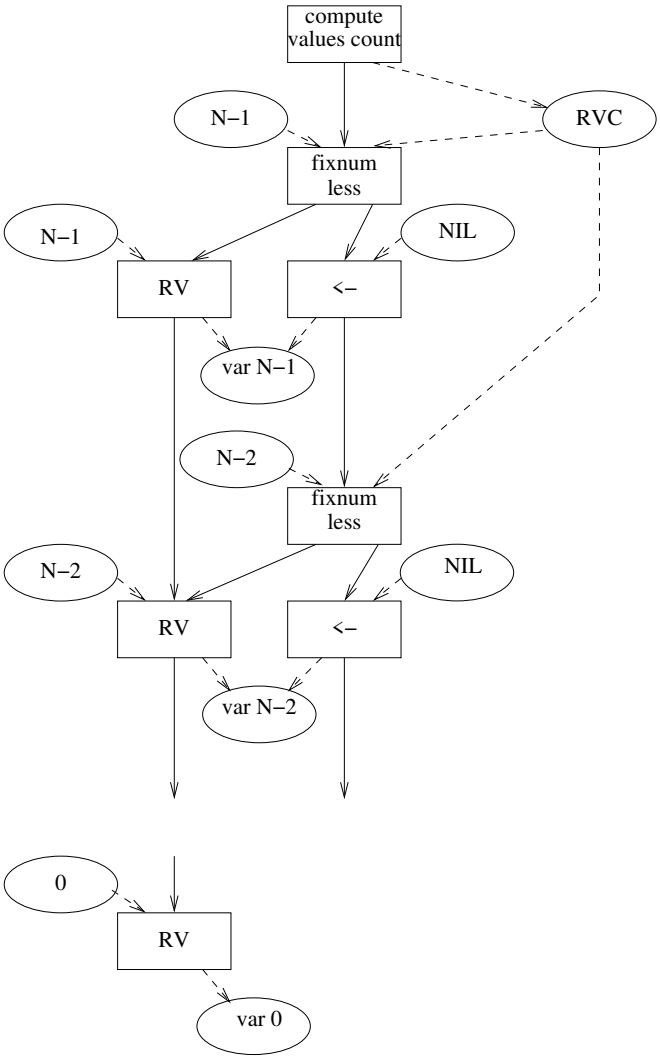

Figure 26.1: Processing return values.

The technique relies on the assumption that there are usually at least as many return values as the number of outputs of the `multiple-to-fixed-instruction`. If that is the case, then the first comparison succeeds, and no other comparison need to be executed.

The last comparison instruction takes the constant 1 as its first input. We do not need to check for the number of return values being greater than 0, because we are always allowed to access return value number 0, even when there are no return values. A function that returns no values puts `nil` in the first values location so that a caller that wants exactly one return value (the most common case) does not need to check the number of return values.

Again referring to Figure 26.1, RVC is a lexical location holding the return-values count as a fixnum. Instructions labeled RV are instruction of type `return-value-instruction` that each takes a constant fixnum input, accesses the value with that index in the distinguished values location and assigns that value to the lexical location of its output. The locations labeled "var 0" to "var N-1" in Figure 26.1 are the outputs of the `multiple-to-fixed-instruction` that is being replaced.

26.2 Replacing the `fixed-to-multiple-instruction`

Clients that would like to use this technique for processing return values would have a HIR transformation that replaces the `fixed-to-multiple-instruction`. Recall that the `fixed-to-multiple-instruction` copies the values of individual lexical locations into the distinguished location for multiple values.

The code that replaces the `fixed-to-multiple-instruction` is shown in Figure 26.2.

In Figure 26.1, N is the number of inputs of the `fixed-to-multiple-instruction`. The `initialize-values-instruction` takes has a fixnum input that initializes the distinguished values location to contain N values.

Following the initialization are N occurrences of the `set-return-value-instruction`, each taking as an input a fixnum indicating the index of the value, and the corresponding input of the `fixed-to-multiple-instruction`. If there are no inputs, i.e. if no values are to be returned, there is nevertheless one occurrence of the `set-return-value-instruction` with inputs 0 and the constant `nil`.

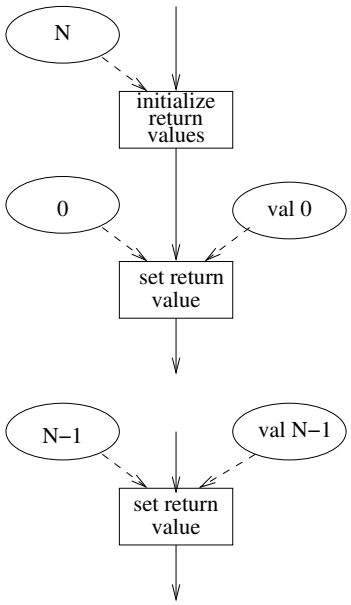

Figure 26.2: Setting return values.

Part III

Backends

Chapter 27

x86-64

27.1 Register usage

The standard calling conventions defined by the vendors, and used by languages such as C use the registers as follows:

Name	Used for	Saved by
RAX	First return value	Caller
RBX	Optional base pointer	Callee
RCX	Fourth argument	Caller
RDX	Third argument, second return value	Caller
RSP	Stack pointer	
RBP	Frame pointer	Callee
RSI	Second argument	Caller
RDI	First argument	Caller
R8	Fifth argument	Caller
R9	Sixth argument	Caller
R10	Temporary, static chain pointer	Caller
R11	Temporary	Caller
R12	Temporary	Callee
R13	Temporary	Callee
R14	Temporary	Callee
R15	Temporary	Callee

We mostly respect this standard, and define the register allocation as follows:

Name	Used for	Saved by
RAX	First return value	Caller
RBX	Dynamic environment	Callee
RCX	Fourth argument, third return value	Caller
RDX	Third argument, second return value	Caller
RSP	Stack pointer	
RBP	Frame pointer	Caller
RSI	Second argument, fourth return value	Caller
RDI	First argument, value count	Caller
R8	Fifth argument	Caller
R9	Argument count, fifth return value	Caller
R10	Static env. argument	Caller
R11	Scratch	Caller
R12	Register variable	Callee
R13	Register variable	Callee
R14	Register variable	Callee
R15	Register variable	Callee

27.2 Representation of function objects

- A static environment.
- The entry point of the function as a raw machine address. Since entry points are word aligned, this value looks like a fixnum.

27.3 Calling conventions

Figure 27.1 shows the layout of a stack frame.

Normal call to external function, passing at most 5 arguments:

1. Compute the callee function object and the arguments into temporary locations.
2. Store the arguments in RDI, RSI, RDX, RCX, and R8.
3. Store the argument count in R9 as a fixnum.
4. Load the static environment of the callee from the callee function object into R10.
5. Push the value of `RBP` on the stack.

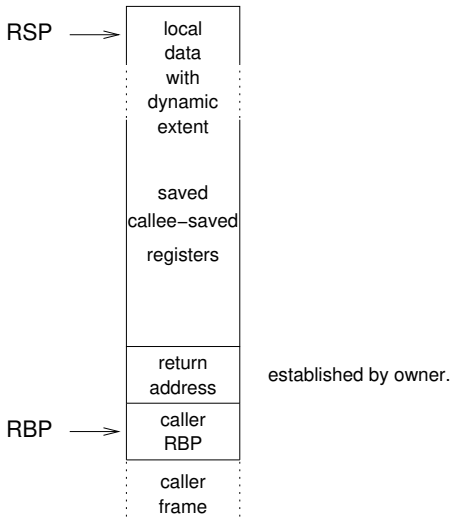

Figure 27.1: Stack frame for the x86-64 backend.

6. Copy the value of RSP into `RBP`, establishing the (empty) stack frame for the callee.
7. Load the entry point address of the callee from the callee function object into an available scratch register, typically RAX.
8. Use the CALL instruction with that register as an argument, pushing the return address on the stack and transferring control to the callee.

Normal call to external function, passing more than 5 arguments:

1. Compute the callee function object and the arguments into temporary locations.
2. Subtract $8(N-3)$ from RSP, where N is the number of arguments to pass, thus leaving room in the callee stack frame for the $N-5$ arguments, the return address, and the caller `RBP`.
3. Store the first 5 arguments in RDI, RSI, RDX, RCX, and R8.
4. Store the remaining arguments in [RSP+0], [RSP+8], ..., [RSP+$8(N-6)$] in that order, so that the sixth argument is on top of the stack.
5. Store the argument count in R9 as a fixnum.

6. Load the static environment of the callee from the callee function object into R10.
7. Store the value of `RBP` into [RSP + 8(N - 4)]
8. Copy the value of RSP+8($N-4$) into `RBP`, establishing the stack frame for the callee. The instruction LEA can be used for this purpose.
9. Load the entry point address of the callee from the callee function object into an available scratch register, typically RAX.
10. Use the CALL instruction with that register as an argument, pushing the return address on the stack and transferring control to the callee.

By using a `CALL`/`RET` pair instead of (say) the caller storing the return address in its final place using some other method, we make sure that the predictor for the return address of the processor makes the right guess about the eventual address to be used.

Figure 27.2 shows the layout of the stack upon entry to a function when more than 5 arguments are passed. Notice that the return address is not in its final place, and the final place for the return address is marked "unused" in Figure 27.2.

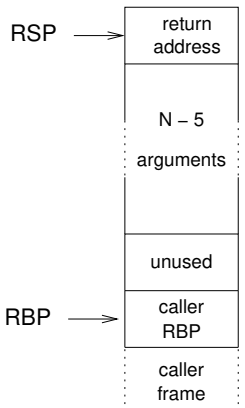

Figure 27.2: Stack frame at entry with more than 5 arguments.

Tail call to external function, passing at most 5 arguments:

1. Compute the callee function object and the arguments into temporary locations.
2. Store the arguments in RDI, RSI, RDX, RCX, and R8.
3. Store the argument count in R9 as a fixnum.

4. Load the static environment of the callee from the callee function object into R10.
5. Copy the value of `RBP−8` to RSP, establishing the stack frame for the callee, containing only the return address. The LEA instruction can be used for this purpose.
6. Load the entry point address of the callee from the callee function object into an available scratch register, typically RAX.
7. Use the JMP instruction with that register as an argument, transferring control to the callee.

Tail call to external function, passing more than 5 arguments:[1]

For *internal calls* there is greater freedom, because the caller and the callee were compiled simultaneously. In particular, the caller might copy some arbitrary *prefix* of the code of the callee in order to optimize it in the context of the caller. This prefix contains argument count checking and type checking of arguments. The address to use for the call is computed statically as an offset from the current program counter, so that a CALL instruction with a fixed relative address can be used. Furthermore, the caller might be able to avoid loading the static environment if it is known that the callee uses the same static environment as the caller.

Upon function entry after an ordinary call, when more than 5 arguments are passed, the callee must pop the return address off the top of the stack and store it in its final location. This can be done with a single POP instruction, using `[RBP−8]` as the destination. When fewer than 5 arguments are passed, the return address is already in the right place.

Return from callee to caller with no values:

1. Store NIL in RAX.
2. Store 0 in RDI, represented as a fixnum.
3. Store the value of `RBP−8` in RSP so that the stack frame contains only the return address. To accomplish this effect, the callee can use the LEA instruction.
4. Return to the caller by executing the RET instruction.

Return from callee to caller with a $1-5$ values:

1. Store the values to return in RAX, RDX, RCX, RSI, R9.

[1] FIXME: Determine the protocol.

2. Store the number of values in RDI, represented as a fixnum.
3. Store the value of `RBP`−8 in RSP so that the stack frame contains only the return address. To accomplish this effect, the callee can use the LEA instruction.
4. Return to the caller by executing the RET instruction.

Return from callee to caller with more than 5 values:

1. Store the first 5 values to return in RAX, RDX, RCX, RSI, R9.
2. Store the remaining (up to 15) values in [RBP-8], [RBP-16], etc.
3. Store the number of values in RDI, represented as a fixnum.
4. Store the value of `RBP`−8 in RSP so that the stack frame contains only the return address. To accomplish this effect, the callee can use the LEA instruction.
5. Return to the caller by executing the RET instruction.

27.4 Use of the dynamic environment

The dynamic environment is an ordinary list of *entries* allocated on the stack rather than on the heap. The head of the list is pointed to by the register `RBX`. (See Section 27.1.)

⇒ `dynamic-environment-entry` [*Class*]

This class is the base class of all entry classes of the dynamic environment.

⇒ `exit-point-entry` [*Class*]

This class is a subclass of the class named `dynamic-environment-entry`. It is a superclass of all entries that represent *exit points*. The Common Lisp standard is somewhat unclear as to what constitutes and exit point. The glossary includes `unwind-protect` in the set of exit points. However, section 5.2 suggests otherwise. Step 1 of the procedure described in section 5.2 is to "abandon" all intermediate exit points. But with the definition in the glossary, the exit points representing `unwind-protect` forms would also be "abandoned", whatever that might mean.

Here, we restrict the term *exit point* to be program points established by `catch`, `block`, and `tagbody`.

⇒ `valid-p` *exit-point-entry* [*Generic Function*]

This generic function returns *true* if and only if *exit-point-entry* is valid, i.e. if it has not been invalidated (or "abandoned") since its creation.

⇒ `invalidate` *exit-point-entry* [*Generic Function*]

This generic function invalidates the entry *exit-point-entry*. After this function has been called, a call to `valid-p` with *exit-point-entry* as an argument always returns *false*.

⇒ `special-binding-entry` [*Class*]

Instances of this class are used to represent the binding of a special variable. It is a subclass of the class named `dynamic-environment-entry`.

⇒ `:symbol` [*Initarg*]

The value of this initarg is the symbol representing the special variable to be bound.

⇒ `:value` [*Initarg*]

The value of this initarg is the value to which the special variable is to be bound.

⇒ `symbol` *special-binding-entry* [*Generic Function*]

This generic function returns the symbol passed as the value of the initarg `:symbol` when the entry *special-binding-entry* was created.

⇒ `value` *special-binding-entry* [*Generic Function*]

This generic function returns the current value of the special variable represented by *special-binding-entry*.

⇒ `(setf value)` *new-value special-binding-entry* [*Generic Function*]

This generic function sets the current value of the special variable represented by *special-binding-entry*.

⇒ `catch-entry` [*Class*]

An instance of this class is used to represent a `catch` tag. It is a subclass of the class named `exit-point-entry`.

⇒ `:tag` [*Initarg*]

The value of this initarg is the `catch` tag represented by this entry.

⇒ `tag` *catch-entry* [*Generic Function*]

This generic function returns the value of the `:tag` initarg that was given when *catch-entry* was created.

⇒ `block/tagbody-entry` [*Class*]

An instance of this class is used to represent an exit point created by `block` or `tagbody`. It is a subclass of the class named `exit-point-entry`.

⇒ `:identifier` [*Initarg*]

The value of this initarg is a unique identifier for this entry. This identifier becomes part of the static environment of any function nested inside the `block` or `tagbody` form that contains a `return-from` or a `go` to this form.

⇒ `identifier` *block/tagbody-entry* [*Generic Function*]

The value returned by a call to this generic function is the identifier of the `block/tagbody-entry` as given by the initarg `:identifier` when the entry was created.

⇒ `unwind-protect-entry` [*Class*]

This class is a subclass of the class named `dynamic-environment-entry`.

⇒ `:cleanup-thunk` [*Initarg*]

The value of this initarg is a thunk that encapsulates the cleanup forms of the `unwind-protect` form.

⇒ `cleanup-thunk` *unwind-protect-entry* [*Generic Function*]

A call to this generic function returns the cleanup thunk of *unwind-protect-entry* that was supplied as the value of the initarg `:cleanup-thunk` when the entry was created.

⇒ `multiple-values-entry` [*Class*]

This class is a subclass of the class named `dynamic-environment-entry`. It is used to store a sequence of multiple values when the registers and stack entries for this purpose are insufficient. In particular, `multiple-value-prog1` and `multiple-value-call` may need one or more of these entries.

`catch` is implemented as a call to a function. This function establishes a `catch` tag and calls a thunk containing the body of the `catch` form.

`throw` searches the dynamic environment for an entry with the right `catch` tag which is also valid. The point to which control is to be transferred is stored as the return value of the stack frame containing the `catch` tag.

A `block` form may establish an exit point. In the most general case, a `return-from` is executed from a function lexically-enclosed inside the `block` with an arbitrary number of intervening stack frames. When this is the case, upon entry to the `block` form, a `block/tagbody` entry with a fresh identifier is established. When a `return-from`

is executed, the point to which control is to be transferred is known statically. The identifier is also stored in a lexical variable in the static environment of closures established inside the `block` form that may execute a corresponding `return-from` form. When the `block` form is exited normally, the `block`/`tagbody` entry is popped off the stack and off the dynamic environment.

`return-from` accesses the identifier from the static environment and then searches the dynamic environment for a corresponding identifier in a `block`/`tagbody` entry. If one with the right identifier is found, the lexical environment is restored, and control is transferred to the statically known address.

A `tagbody` may establish several exit points. In the most general case, a `go` is executed from a function lexically-enclosed inside the `tagbody` with an arbitrary number of intervening stack frames. When this is the case, upon entry to the `tagbody` form, a `block`/`tagbody` entry with a fresh identifier is established. When a `go` is executed, the point to which control is to be transferred is known statically. The identifier is also stored in a lexical variable in the static environment of closures established inside the `tagbody` form that may execute a corresponding `go` form. When the `tagbody` form is exited normally, the `block`/`tagbody` entry is popped off the stack and off the dynamic environment.

`go` accesses the identifier from the static environment and then searches the dynamic environment for a corresponding identifier in a `block`/`tagbody` entry. If one with the right identifier is found, the lexical environment is restored, and control is transferred to the statically known address.

27.5 Transfer of control to an exit point

Whenever transfer of control to an exit point is initiated, the exit point is first searched for. If no valid exit point can be found, an error is signaled. If a valid exit point is found, the stack must then be unwound. First, the dynamic environment is traversed for any intervening exit points, and they are marked as invalid as indicated above. Traversal stops when the stack frame of the valid exit point is reached. Unwinding now begins. The dynamic environment is traversed again and thunks in `unwind-protect` entries are executed. The traversal again stops when the stack frame of the valid exit point is reached.

27.6 Address space layout

The architecture specification guarantees that the available address space is at least 2^{48} bytes, with half of it at the low end of the full potential 2^{64} byte address space, and the other half at the high end of the full potential address space. The upper half is typically reserved for the operating system, so for applications only the low 2^{47} bytes are available. We divide this address space as follows:

- The space reserved for dyads in the global heap (see Appendix C) starts at address 0 and ends at 2^{45}. Only a small part of it is initially allocated, of course and it grows as needed.
- The space reserved for racks in the global heap (see Appendix C) starts at address 2^{45} and ends at 2^{46}. Only a small part of it is initially allocated, of course and it grows as needed.
- Each thread is given an address space of 2^{30} bytes with the low part (a few megabytes) reserved for the nursery and the rest reserved for the stack. Only a small part of the stack is initially allocated, and it grows as needed. The address spaces for threads start at address 2^{46}.

This layout is illustrated in Figure 27.3.

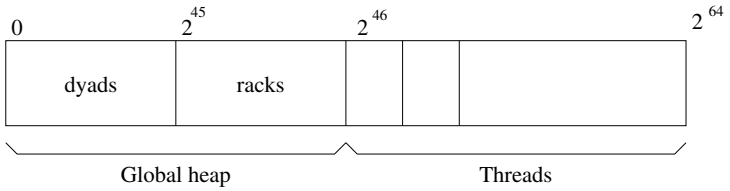

Figure 27.3: Address space layout for the x86-64 backend.

27.7 Parsing keyword arguments

When the callee accepts keyword arguments, it is convenient to have all the arguments in a properly-ordered sequence somewhere in memory. We obtain this sequence by pushing the register arguments to the stack in reverse order, so that the first argument is at the top of the stack. When more than 5 arguments are passed, the program counter is popped off the top of the stack, thereby moving it to its final destination before the register arguments are pushed.

Chapter 28

HIR interpreter

This backend is characterized by the fact that source code is compiled to the high-level intermediate representation (HIR). The HIR code is then interpreted by a Common Lisp program. Recall that the high-level intermediate code only manipulates Common Lisp objects so that all address calculations are implicit.

At the center of this backend is a SICL first-class global environment represented as a host class instance. This environment contains mappings from names to all objects that are part of any global environment, such as functions, macros, variables, classes, packages, types, `setf` expanders, etc.

Part IV

Contributing to SICL

Chapter 29

General Common Lisp style guide

29.1 Purpose of style restrictions

The purpose of imposing a particular style is based on a few simple facts that hold true for both natural languages and programming languages:

- The set of all idiomatic phrases is a tiny subset of the set of all grammatical phrases.
- The main purpose of these phrases is to serve as communication between people.

To illustrate the first fact, consider a natural language such as English. In English, we say "tooth brush", but "dental floss". The words "dental brush" and "tooth floss" would be perfectly grammatical, but they are not used. A person trying to communicate with other people must use the words that have been widely agreed upon, even though some other words are perfectly legitimate. It might seem that such idiosyncrasies would be limited to languages with multiple heritage such as English, but that is not the case. In French, we say "brosse à dents", "pâte dentifrice", and "fil dentaire". There is a large number of reasonable combinations, but only one is used.

The same thing is true for programming languages. The community has collectively decided on a particular subset of all the grammatical phrases, and a programmer who wishes to communicate with other programmers should stick to that subset.

It should also be emphasized that the choice of idioms is different in different languages. An example from natural languages would be that in English we say "I wash my hands", in French "I wash myself the hands", and in Swedish we say "I wash the hands [on myself]". Just as it would be pointless trying to use an idiom from one language in a translated version in a different language, it is as pointless to translate idioms from one programming language to a different programming language.

Finally, the choice of what phrases are idioms and what phrases are not, is almost totally arbitrary, and based on coincidences of history. Therefore it is rarely productive to ask oneself why a particular phrase is an idiom and a different one is not. There is no possible enlightening answer to such a question.

29.2 Width of a line of code

Horizontal space is a precious resource that should not be wasted. The width of a line should preferably not exceed 80 characters. This limit used to be hard, because some printers or printer drivers would truncate longer lines. Since it is less common to print code these days, the limit is now soft. The purpose of keeping lines somewhat short is so that it is possible on a reasonable monitor to display two documents side by side. One document is typically a Common Lisp source file, and the other document is typically the buffer containing interactions with the Common Lisp system.

The systematic use of long lines makes the practice of displaying two documents side by side impossible, or at least impractical. If a single monitor is used, the programmer then has to flip back and forth between the source code and the interaction loop. When two monitors are used, the effect is to waste half a monitor that could otherwise be used for displaying documentation or something else.

29.3 Commenting

Use a single semicolon to introduce a comment that follows the code on a line. Use two semicolons for comments that are not at the top level in a file and that should be aligned with the code that it comments on. Use three semicolons for top-level comments that concern some top-level forms in a file, but not the entire file. Use four semicolons for comments that concern the entire file.

29.4 Blank lines

A single blank line is common in the following situations:

- Between two top-level forms.
- Between a file-specific comment and the following top-level form.
- Between a comment for several top-level forms and the first of those top-level forms.

A single blank line *may* occur inside a top-level form to indicate the separation of two blocks of code concerned with different subjects, but it would be more common to put those two blocks of code in separate functions.

There should never be any instance of two consecutive blank lines, and the last line of the file should not be blank.

29.5 `car`, `cdr`, `first`, etc are for `cons` cells

The Common Lisp standard specifies that the function `car`, `cdr`, `first`, `second`, `rest`, etc return `nil` when `nil` is passed as an argument. This fact should mostly be considered as a historical artifact and should not be systematically exploited. Take for instance the following code:

```
(if (first x) ...)
```

To the compiler, it means "execute the false branch of the `if` when either `x` is `nil`, or when `x` is a list whose first element is `nil`".

To the person reading the code, it means something different altogether, namely "`x` holds a non-empty list of Boolean values, and the false branch of the `if` should be executed when the first element of that list is *false*. See also Section 29.6.

29.6 Different meanings of `nil`

Consider the following local variable bindings:

```
(let ((x '())
      (y nil)
      z)
  ...)
```

To the compiler, the three are equivalent. To a person reading the code, they mean different things, however:

- The initialization of `x` means that `x` holds a *list* that is initially empty.
- The initialization of `y` means that `y` holds a Boolean value or a default value that may or may not change in the body of the `let` form.
- The absence of initialization of `z` means that no initial value is given to `z`. In the body of the `let` form, the variable `z` will be assigned to before it is used.

The following body of the `let` form corresponds to the expectations of the person reading the code:

```
(let ((x '())
      (y nil)
      z)
  ...
  (push (f y) x)
  ...
  (unless y (setf y (g x)))
  ...
  (setf z (h x))
  ...)
```

The following body of the `let` form violates the expectations of the person reading the code:

```
(let ((x '())
      (y nil)
      z)
  ...
  (push (f y) z)     ; z is used before it is assigned.
  ...
  (unless x          ; x is treated as a Boolean.
```

```
    (setf y (g x)))
  ...
  (push (f x) y)     ; y is treated as a list.
  ...)
```

29.7 Tests in conditional expressions

The *test* of a conditional expression should be a (possibly generalized) Boolean expression. The following expressions correspond to the expectations of the person reading the code:

```
(if visited-p ...)
(when (member ...) ...)
(cond ((plusp x) ...) ...)
```

The following code violates the expectation:

```
(let ((item (find ...)))
  (when item ...))
```

because `item` is not a (generalized) Boolean value. It is an item returned by `find`, though there is an *out of band* value (`nil`) indicating that no item was found by `find`. In this case, the corresponding code that corresponds to the expectations would look like this:

```
(let ((item (find ...)))
  (unless (null item) ...))
```

29.8 General structure of recursive functions

When possible, a recursive function should be structured like a mathematical proof by induction. By that we mean that the special case should be handled *first* so as to reassure the person reading the code that this case can be handled correctly by the function.

So for instance, assume we want to write a function that counts the number of atoms in a tree, we should not write it like this:

```
(defun count-atoms (tree)
  (if (consp tree)
      (+ (count-atoms (car tree))
         (count-atoms (cdr tree)))
      1))
```

but rather

```
(defun count-atoms (tree)
  (if (atom tree)
      1
      (+ (count-atoms (car tree))
         (count-atoms (cdr tree)))))
```

Even when the base case does not return anything useful, it should be handled first. The following code violates the expectations:

```
(defun map-conses (function tree)
 (unless (atom node)
   (funcall function node)
   (traverse (car node))
   (traverse (cdr node))))
```

and should be written like this instead:

```
(defun map-conses (function tree)
  (if (atom node)
      nil ; nothing to do
      (progn (funcall function node)
             (traverse (car node))
             (traverse (cdr node)))))
```

though, admittedly, this example is a little too simple to illustrate the importance of this rule.

29.9 Using `car` and `cdr` vs. using `first` and `rest`

While the two functions `car` and `first` have the exact same definitions, as do `cdr` and `rest`, they send very different messages to the person reading the code.

The functions `car`, `cdr`, etc., should be avoided when the argument is to be considered as a *list*, and should be reserved for other uses of `cons` cells such as for *trees* or *pairs* of values.

It follows that the two families of functions should never be mixed for the same argument.

Chapter 30

SICL-specific style guide

30.1 Commenting

In most programs, comments introduce unnecessary redundancies that can then easily get out of sync with the code. This is less risky for an implementation of a specification that is not likely to change. Furthermore, we would like SICL to be not only a high-quality implementation, but we would like for its code to be very readable. For that reason, we think it is preferable to write SICL in a "literate programming" style, with significant comments explaining the code.

Accordingly, we prefer comments to consist of complete sentences, starting with a capital letter, and ending with punctuation.

30.2 Designators for symbol names

Always use uninterned symbols (such as `#:hello`) whenever a string designator for a symbol name is called for. In particular, this is useful in `defpackage` and `in-package` forms.

Using the upper-case equivalent string makes the code break whenever the reader is case-sensitive (and it looks strange that the designator has a different case from the way symbol that it designates is then used), and using keywords unnecessarily clutters the keyword package.

30.3 Docstrings

We believe that it is a bad idea for an implementation of a Lisp system to have docstrings in the same place as the definition of the language item that is documented, for several reasons. First, to the person reading the code, the docstring is most often noise, because it is known from the standard what the language item is about. Second, it often looks ugly with multiple lines starting in column 1 of the source file, and this fact often discourages the programmer from providing good docstring. Third, it makes internationalization harder.

For this reason, we will provide language-specific files containing all docstrings of Common Lisp in the form of calls to `(setf documentation)`.

We also recommend using `format` (at read time) so that the format directive `~@` can be used at the end of lines, allowing the following line to be indented as the rest of the text. That way, we avoid the ugliness of having subsequent lines start in column 1.

30.4 Naming and use of slots

In order to make the code as safe as possible, we typically do not want to export the name of a slot, whereas frequently, the reader or the accessor of that slot should be exported. This restriction implies that a slot and its corresponding reader or accessor cannot have the same name. Several solutions exist to this problem. The one we are using for SICL is to have slot names start with the percent character ('`%`'). Traditionally, a percent character has been used to indicate some kind of danger, i.e. that the programmer should be very careful before directly using such a name. Client code that attempts to use such a slot would have to write `package::%name` which contains two indicators of danger, namely the double colon package marker and the percent character.

Code should refer to slot names directly as little as possible. Even code that is private to a package should use an internal protocol in the form of readers and writers, and such protocols should be documented and exported whenever reasonable.

30.5 Standard functions

Standard functions should always check the validity of their arguments and of any other aspect of the environment. If such a function fails to accomplish its task, it

should signal an appropriate condition.

We would like error messages to be phrased in terms of the code that was directly invoked by user code, as opposed to in terms of code that was indirectly invoked by system code. As an example, consider a sequence function such as `substitute`. If it is detected that a dotted list has been passed to this function, it should not be reported by `endp` or any other system function that was not directly called by user code, but instead it should be reported by `substitute` in terms of the sequence that was originally passed as an argument. On the other hand, if substitute invokes a user-supplied test that fails, we would like the error message to be reported in terms of that user-supplied code rather than by `substitute`. This is how we are currently imagining solving this problem:

- Standard functions do not call any other standard functions directly, other than if it is known that no error will be signaled. When a call from a standard function f to a standard function g might result in an error being signaled by g, that call is replaced by a call to a special version of the standard function, say h that signals a more specific condition than is dictated by the Common Lisp HyperSpec.
- If acceptable in terms of performance, a standard function such as f that calls other functions that may signal an error, handles such errors by signaling an error that is directly related to f.
- Error reporting is done in terms of the name and arguments to f.

30.6 Standard macros

Standard macros must do extensive syntax analysis on their input so as to avoid compilation errors that are phrased in terms of expanded code.

As with standard functions, standard macros that expand into other system code that may signal an error should not use other standard functions or other standard macros directly, but instead special versions that signal more specific conditions. The expanded code should then contain a handler for such errors, which signals an error in terms of the name and the arguments of the macro.

30.7 Compiler macros

SICL will make extensive use of compiler macros. Compiler macros are part of the

standard, so this mechanism must be part of a conforming compiler anyway. In many cases, instead of encoding special knowledge in the compiler itself, we can use compiler macros. By doing it this way, we simplify the compiler, and we provide a set of completely portable macros that any implementation can use.

Compiler macros should be used whenever the exact shape of the call site might be used to improve performance of the callee. For instance, when the callee uses keyword arguments, we can eliminate the overhead of keyword-value parsing at runtime and instead call a special version of the callee that does not have to do any such parsing.[1]

Similarly, functions that take a `&rest` argument can provide special cases for different common sizes of the `&rest` argument.

We propose using compiler macros at least for the following situations:

- to convert calls to `list` and `list*` into nested calls to `cons`;
- to convert simple calls to some built-in functions that accept `:test` and `:key` keyword arguments (such as `find`, `member`, etc) into calls to special versions of these procedures with particularly simple functions for these keyword arguments (`identity`, `car`, `eq`, etc);
- to convert calls to some functions that accept optional arguments such as `last` and `butlast` into calls to special versions when the optional argument is not given.

Compiler macros should not be used in the place of inlining.

30.8 Conditions and restarts

SICL functions should signal conditions whenever this is required by the Lisp standard (of course) and whenever it is *allowed* by the Lisp standard and reasonably efficient to do so. If the standard allows for subclasses of indicated signals (I think this is the case), then SICL should generate as specific a condition as possible, and the conditions should contain all available information as possible in order reduce the required effort to find out where the problem is located.

SICL function should also provide restarts whenever this is practical.

[1] FIXME: There is a suggestion that this creation could be automated by the compiler. I don't know how doable that would be.

30.9 Condition reporting

Condition reporting should be separate from the definition of the condition itself. Separating the two will make it easier to customize condition reporting for different languages and for different systems. An integrated development environment might provide different condition reporters from the normal ones, that in addition to reporting a condition, displays the source-code location of the problem.

Every SICL module will supply a set of default condition reporters for all the specific conditions defined in that module. Those condition reporters will use plain English text.

30.10 Internationalization

We would like for SICL to have the ability to report messages in the local language if desired. The way we would like to do that is to have it report conditions according to a `language` object. To accomplish this, condition reporting trampolines to an implementation-specific function `sicl:report-condition` which takes the condition, a stream, and a language as arguments.

The value of the special variable `sicl:*language*` is passed by the condition-reporting function to `sicl:report-condition`.

In other words, the default `:report` function for conditions is:

```
(lambda (condition stream)
  (sicl:report-condition condition stream sicl:*language*))
```

Similarly, the Common Lisp function `documentation` should trampoline to a function that uses the value of `sicl:*language*` to determine which language to use to show the documentation.

30.11 Package structure

SICL has a main package containing and exporting all Common Lisp symbols. It contains no other symbols. A number of implementation packages import the symbols from this package, and might define internal symbols as well. Implementation packages may export symbols to be used by other implementation packages.

This package structure allows us to isolate implementation-dependent symbols in different packages.

30.12 Assertions

30.13 Threading and thread safety

Consider the use of locks to be free. We predict that a technique call "speculative lock elision" will soon be available in all main processors.

Chapter 31

List of tasks of limited size

In this chapter, we give a list of tasks that can be accomplished in a shorter period of time, typically between a few days and a few weeks. The tasks in this list are meant for people who would like to contribute to SICL, but who either lack a significant amount of time, or the knowledge, to intervene in more complex tasks.

Each task in this list is meant to be interesting to the person who decides to take it on. Thus, we have avoided trivial tasks such as untabifying source code, fixing grammar in comments, and generally altering code to conform to the guidelines in the previous two chapters.

31.1 Implement hash tables

Hash tables have not yet been implemented in SICL. We would like to investigate several possible implementations, and perhaps propose several such implementations, with different characteristics, in the code base of SICL.

Since SICL has a very efficient technique for generic dispatch, we think that there could be a hierarchy of classes with different characteristics, all implementing the standard protocol for hash tables specified by the Common Lisp standard.

In general, we want hash tables to be thread safe. If it is possible to make them lock free, that is even better.

The code should be structured so that it looks "natural" in intrinsic setting, i.e. when the code is loaded into SICL. However, we would like for the code to be structured

such that it can be tested in an extrinsic setting as well.

The code should contain a separate `test` system, probably using extensive testing through the use of randomly generated operations.

31.1.1 Implementation using a list

We would like a simple implementation where elements of the table are kept in an ordinary list. Such an implementation avoids the entire question of hashing functions. As such, it is particularly well suited to serve during bootstrapping

31.1.2 Implementation using open hashing

31.1.3 Implementation using vector buckets

31.2 Implement streams

Streams have non yet been implemented in SICL. The full set of streams defined by the standard must be implemented at some point.

Initially, we are not concerned with extreme performance requirements. For that reason, we think that the protocol functions can be implemented on top of the Gray-streams protocol.

The code should be structured so that it looks "natural" in intrinsic setting, i.e. when the code is loaded into SICL. However, we would like for the code to be structured such that it can be tested in an extrinsic setting as well.

31.3 Better error messages for the `loop` module

The parser for the `loop` macro uses a home-grown version of combinator parsing. Since `loop` clauses do not need backtracking, this feature of normal combinator parsing is not included.

However, we think that combinator parsing is perhaps not a well suited technique when good error messages are a requirement.

This task consists of investigating whether good error messages can be obtained with

the current technique, while maintaining a reasonable structure of the code. Or whether some other technique might be better suited. We are thinking that Earley parsong might be a good candidate.

31.4 Better error messages by the lambda-list parser

The parser for the `loop` macro uses a home-grown implementation of the Earley parsing technique. In theory, this technique should be excellent when it comes to generating good error messages in case of parse failures. However, we have not implemented such error messages yet.

This task consists of adding such error messages.

Part V

Appendices

Appendix A

All standard macros

and assert call-method case ccase check-type cond ctypecase declaim decf
defclass defconstant defgeneric define-compiler-macro define-condition define-method-co
define-modify-macro define-setf-expander define-symbol-macro defmacro defmethod
defpackage defparameter defsetf defstruct deftype defun defvar destructuring-bind
do do* do-all-symbols do-external-symbols do-symbols dolist dotimes ecase
etypecase formatter handler-bind handler-case ignore-errors in-package incf
lambda loop loop-finish multiple-value-bind multiple-value-list multiple-value-setq
nth-value or pop pprint-exit-if-list-exhausted pprint-logical-block pprint-pop
print-unreadable-object prog prog* prog1 prog2 psetf psetq push pushnew
remf restart-bind restart-case return rotatef setf shiftf step time trace
typecase unless untrace when with-accessors with-compilation-unit with-condition-restart
with-hash-table-iterator with-input-from-string with-open-file with-open-stream
with-output-to-string with-package-iterator with-simple-restart with-slots
with-standard-io-syntax

Appendix B

Removed systems

This appendix contains a list of systems that were removed for one of several reasons:

- The code used to work, but is no longer working due to updates in code that it depends on. More work is needed for it to be usable again.
- The code never worked, and it needs more work in order for it to be considered part of SICL.
- The code never worked, and we have no ambition to make it work, but it nevertheless contains interesting ideas that we might revive in the future.

B.1 Stack-oriented C backend

Removed 2016-08-09.

SHA1 ID: b983763485ac5efb6fe3319f015b414b3e7ea5f5

B.2 Concrete Common Lisp backend

Removed 2016-08-09.

SHA1 ID: 3e0ca56b20ab49b27ae16bbe2046b975ac27be3e

B.3 Extrinsic HIR interpreter backend

Removed 2016-08-09.

SHA1 ID: 5a902f694be00857b33e7d79f0ddfa237a6dab7d

B.4 Abstract machine backend

Removed 2016-08-09.

SHA1 ID: b905eba4da8daf89c450b6ab31748242b80e288e

B.5 X86 assembler

Removed 2016-08-10

SHA1 ID: 74172920471bade46e86212a41ebf487e36e36f6

B.6 Global system definition and associated package file

Removed 2016-08-12

SHA1 ID: 8016f987986a85a6276fff5f1e54b7b6bafcd428

B.7 File containing definitions of tag bits

Removed 2016-08-12

SHA1 ID: 94860f0654ecc40331d3db1ee6f8f9fc881f7307

Appendix C

Memory allocator

C.1 Memory is divided into chunks

Available memory is divided into consecutive *chunks* with no space in between two chunks. If there are two consecutive chunks $C1$ and $C2$ somewhere in available memory, then $C1$ is said to be the *preceding* chunk with respect to $C2$ and $C2$ is said to be the *following* chunk with respect to $C1$.

A chunk can either be *in use* or *free*. When a chunk is freed, and either the preceding chunk is free, the following chunk is free, or both the preceding and the following chunks are free, then the free chunks are coalesced into a single free chunk. As a consequence, there are never two consecutive free chunks in memory.

Every chunk C has an initial 64-bit word containing the *size* of the chunk in 8-bit bytes. Since this value is always a multiple of 4[1], the last two bits are always 0. We use the *next to last* bit to indicate whether C is in use or free. If the bit is 1, the chunk C is in use. If the bit is 0, the chunk C is free. We use *the last* bit to indicate whether the *preceding chunk* with respect to C is in use or free. If the bit is 1, the preceding chunk is in use. If the bit is 0, the preceding chunk is free. If there is no preceding chunk, i.e. C is the first chunk in memory, then the last bit of the first word is set to 1 to avoid any attempt to coalesce C with some non-existing preceding chunk. Notice that for all chunks except the last one, there are therefore two bits in memory indicating whether the chunk is in use or free; one bit in the first word of the

[1] FIXME: There is a question concerning whether it would be better to align user data on cache lines.

chunk itself, and one bit in the following chunk.

To obtain the size of the chunk in bytes, the last two bits of the word must first be masked out, using the `and` operation with a mask containing a 1 in every position except the two least significant ones.[2]

A free chunk is linked into a doubly linked list of chunks in the same bin. See Section C.2 for more details about the available bins. If there are two consecutive chunks $C1$ and $C2$ in the linked list in a bin, then $C1$ is said to be the *previous* chunk with respect to $C2$ and $C2$ is said to be the *next* chunk with respect to $C1$.

The second 64-bit word of a free chunk is a pointer to the previous chunk in the bin. The third 64-bit word of a free chunk is a pointer to the next chunk in the bin. These words do not point to the beginning of the previous or the next chunk. Instead, the second word contains the address of the third word of the previous chunk in the bin, and the third word contains the address of the second word of the next chunk in the bin. If there is no previous chunk in the bin, then the second word contains the address of a *sentinel* that is the beginning of the bin. Similarly, if there is no next chunk in the bin, then the third word contains the address of a sentinel that is the end of the bin. By using these sentinels, we are able to simplify the algorithms for linking and unlinking a chunk. The last word of a *free chunk* contains the size of the chunk, just like the first word. The last word of a chunk in use is reserved for user data.

Since a free chunk has at least four words in it, this is also the minimum size allowed for any chunk.

Figure C.1 shows the constellation of two chunks where the first chunk is in use and the second chunk is free.

As Figure C.1 shows, the next-to-last bit of the first word of the first chunk and the last bit of the first word of the second chunk are both 1, indicating that the first chunk is in use. The next-to-last bit of the first word of the second chunk is 0 indicating that the second chunk is free. The first chunk contains the size word followed by user data. The second chunk contains the size in the first and the last word, and the second and third words of the second chunk contain links to the previous and the next chunk in the bin.

Figure C.2 shows the constellation of two chunks where the first chunk is free and the second chunk is in use.

As Figure C.2 shows, the next-to-last bit of the first word of the first chunk and the

[2] FIXME: There is a suggestion to store the size multiplied by 4 so that it can be obtained by shifting right by two bits. This suggestion would avoid having a 64-bit mask. It is quite unlikely that storing sizes this way would not be enough, at least for the foreseeable future.

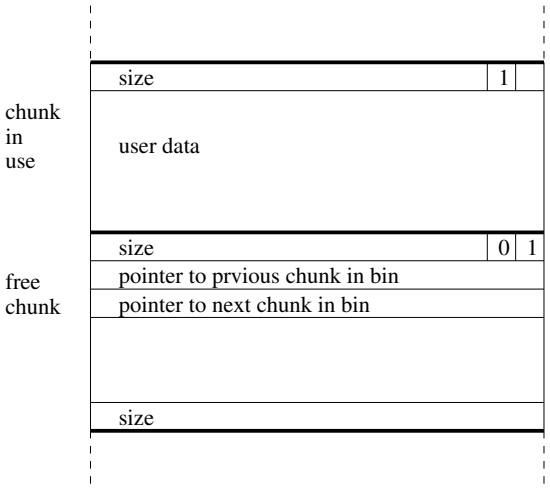

Figure C.1: Chunk in use followed by free chunk.

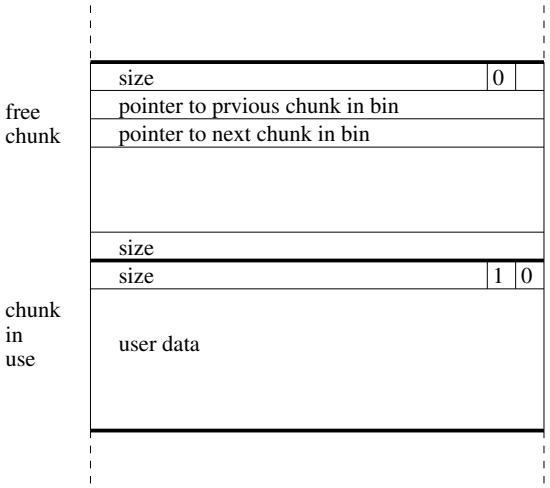

Figure C.2: Free chunk followed by chunk in use.

last bit of the first word of the second chunk are both 0, indicating that the first chunk is free. The next-to-last bit of the first word of the second chunk is 1 indicating that the second chunk is in use. The first chunk contains the size in the first and the last word, and the second and third words of the first chunk contain links to the previous and the next chunk in the bin. The second chunk contains the size word followed by user data.

Figure C.3 shows the constellation of two chunks where both chunks are in use.

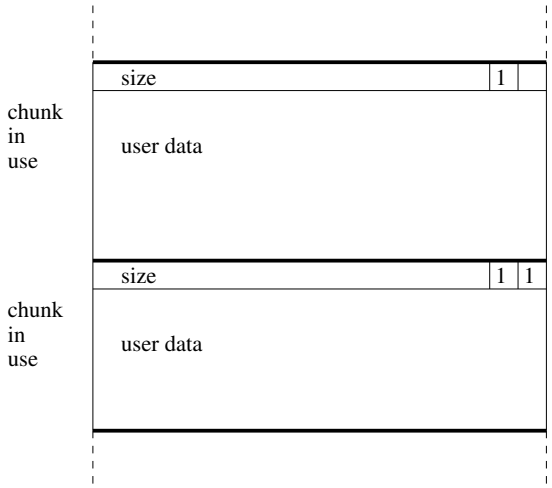

Figure C.3: Two chunks in use.

As Figure C.3 shows, the next-to-last bit of the first word of the first chunk and the last bit of the first word of the second chunk are both 1, indicating that the first chunk is in use. The next-to-last bit of the first word of the second chunk is also 1 indicating that the second chunk is in use as well. Each of the chunks contains the size in the first word and the remaining words of the chunk contains user data.

C.2 Bins of chunks of similar size

As indicated in Section C.1, we maintain a number of *bins* containing chunks of similar size. There are 512 bins in total, numbered from 0 to 511. Each of the bins from 0 to 63 contains chunks of a single size. Bin 0 contains chunks with 4 words (the minimum size) and bin 63 contains chunks with 67 words. Bins starting at 64 contain chunks with a size greater than the maximum size of chunks in the previous bin and less than

or equal to some maximum size that is indicated explicitly. The maximum size of chunks in bins 64 to 511 grows by roughly less than 10% compared to the previous one. Thus the maximum size of chunks in bin 64 is 73, that of chunks in bin 65 is 79, etc. The maximum chunk size of bin 511 is 2^{61} words, or 2^{64} bytes. In bins 0 to 63, chunks are sorted by address. In bins 64 to 511 chunks are sorted first by size and then (if several chunks have the same size) by address.[3]

Three vectors of 512 elements each are used to manage the bins. One vector contains the maximum size[4] of chunks in the bin that corresponds to the index. The second vector contains the first sentinel of each bin. The third vector contains the last sentinel of each bin. When the bin is empty, the element in the second vector contains the address of the element in the third vector and vice versa. In addition to these three vectors, we also maintain a bitmap[5] consisting of 8 64-bit words, containing a 1 if the corresponding bin has at least one chunk in it and 0 if the corresponding bin contains no chunks. Figure C.4 illustrates the organization of the bins.

In the example in Figure C.4, bin 0 contains two chunks, bin 63 a single chunk, and bin 66 contains three chunks. Bins 64, 65, and 511 contain no chunks.

C.3 Linking a chunk into a bin

To link an arbitrary chunk of n words into a bin, we first determine which bin it should be linked into. If $n \leq 67$ then the chunk goes into bin $n - 4$. If not, we do a binary search to find the bin with the smallest maximum chunk size that is greater than or equal to n. We then do a linear search of the chunks in the bin, ending either when we find the last sentinel, or when we find the first chunk that should be located after the one to be inserted. Finally, the chunk to be inserted is linked in using the normal insertion procedure for doubly linked lists.

[3] FIXME: There is a question concerning whether odd-sized chunks are needed. I think the answer is that they are not strictly needed, but that more space would be wasted without them.

[4] In the implementation, the sizes in this vector are given in number of 8-bit bytes, rather than in number of 64-bit words, so as to avoid unnecessary arithmetic operations in the allocator algorithms.

[5] This bitmap is not yet implemented.

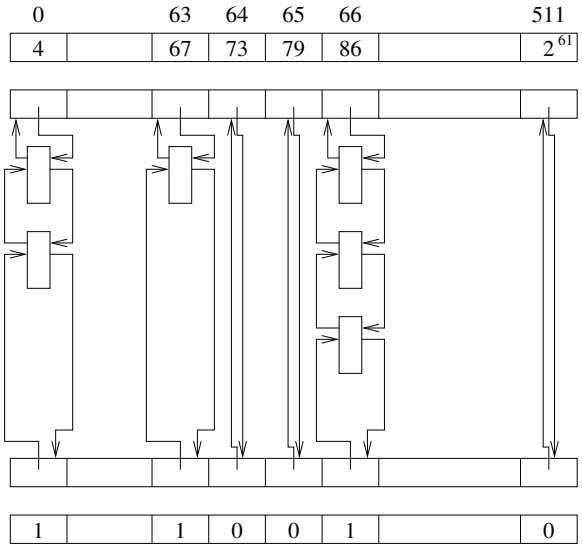

Figure C.4: Organization of bins.

C.4 Allocating a chunk

To allocate a chunk with at least n words in it, we first determine the bin b with the smallest possible maximum chunk size that is greater than or equal to n. If $n \leq 67$ then the b is computed as $n - 4$. If not, we do a binary search to find b.

It is possible that b is empty. We must therefore find the smallest bin b' such that $b' \geq b$ and b' is not empty. We first compute $q = b\,div\,64$ and $r = b\,mod\,64$ indicating that b corresponds to position r in bitmap q. We then compute a mask m consisting of r leading 0s and $64 - r$ trailing 1s. This mask is `and`-ed with the bitmap. We then find the first 1 in the resulting value. If such a position exists, then we have found b'. If not, we search bitmaps $q + 1$, $q + 2$ etc., without any mask, until a set bit is found. The first such bit found corresponds to b'. If no such bit is found, we request more memory from the operating system.

When a non-empty bin is found, if the bin index is less than or equal to 63 we use the first chunk in the bin. If the bin index is greater than or equal to 64, we do a linear search of the chunks in the bin, and use the first chunk that is greater than or equal to n words.

Once we find a chunk c that is greater than or equal to n words, there are two cases:

1. If the size of c is less than or equal to $n - 4$, then we unlink the chunk from the bin and return it. The reason for the 4 is that we can not use a residue less than 4 words.
2. If the size s of c is strictly greater than n, then we unlink c from its bin. We then split c into a chunk $c1$ of size n and a chunk $c2$ of size $s - n$. The chunk $c2$ is then linked into the bin that corresponds to its size. The free/used bits are updated to reflect the fact that $c1$ is now used. Finally, the chunk $c1$ is returned.

C.5 Freeing a chunk

When a chunk C is freed, there are four possible situations (recall that "preceding" and "following" refer to the order between chunks in the address space):

1. The chunk P preceding C and the chunk F following C are both in use. Then, we just link C into an appropriate bin.
2. The chunk P preceding C is in use but the chunk F following C is free. Then we first coalesce C with F, and then we link the resulting chunk into an appropriate bin.
3. The chunk P preceding C is free but the chunk F following C is in use. Then we first coalesce C with the P, and then we link the resulting chunk into an appropriate bin.
4. The chunk P preceding C and the chunk F following C are both free. Then we first coalesce C both with P and F, and then we link the resulting chunk into an appropriate bin.

To determine whether the chunk P preceding C is in use or free, we consult the last bit of the first word C. If it is 1, then P is in use. If it is 0, P is free. Only when this bit is 0 is it possible to find the beginning of P, because only then does the last word of P contain the size of P. And that size is needed to find the beginning of P. This is the main reason for storing the in-use bit in the chunk following the one that is concerned.

To determine whether the chunk F following C is in use or free, use the size of C to find the beginning of F. We then consult the next-to-last bit of the first word of F. If it is 1, then F is in use. If it is 0, then F is free.

To coalesce the chunk P preceding C (case 2 above) with C, P must first be unlinked from its bin. Then the size contained in the first word of P is modified to contain the

sum of the initial sizes of P and C. This new size is then stored in the last word of C as well. Finally, the resulting chunk is linked into the bin corresponding to the new size.

To coalesce the chunk C with the chunk F following C (case 3 above), F must first be unlinked from its bin. Then the size contained in the first word of C is modified to contain the sum of the initial sizes of C and F. This new size is then stored in the last word of F as well. Finally, the resulting chunk is linked into the bin corresponding to the new size.

To coalesce the chunk P preceding C, C itself, and the chunk F following C (case 4 above), both P and F must first be unlinked from their respective bins. Then the size contained in the first word of P is modified to contain the sum of the three sizes. This new size is then stored in the last word of F as well. Finally, the resulting chunk is linked into the bin corresponding to the new size.

C.6 Concurrency

To be filled in. Talk about what kind of synchronization is required.

Appendix D

Bootstrapping principles

In this appendix, we describe general principles of bootstrapping, as opposed to implementation details.

D.1 General restrictions

We define the *purity* of some object to be a non-negative integer. An object of purity p is an instance of a class of purity $p-1$. An object of purity 0 is a *host object*. An object of purity 1 is a is a *bridge object*. An object of purity 2 is an *impure ersatz object*. An object of purity 3 or more is a *pure ersatz object*. Each *phase* of the bootstrapping procedure creates objects of one purity. Currently, bootstrapping phase n creates objects of purity $p = n - 2$. Phase n puts generic function objects in environment E_{n+1} and class objects in environment E_n.

Suppose we want to access some part of a generic function metaobject of purity p. Perhaps we want to add methods to it, or set its discriminating function. During bootstrapping, such access must be done by fully functioning generic functions. For that reason, we use functions of purity $p-1$ for such access, and we can assume that when we need to accomplish such access to a generic function of purity p, then either the accessors of purity $p-1$ are either already fully functional, or the machinery for making them fully functional is fully functional, so that we can freely use generic functions of purity $p-1$ to access a generic function of purity p.

In general we would like for the slots of an object of purity p to contain objects of purity p, with the exception of meta-level information.

For generic function metaobjects, this restriction implies that we want the methods of a generic function of purity p to be objects of purity p, and we want the method functions of those methods to be objects of purity p. Idem for the *method combination*, the *effective method functions*, and the *discriminating function*. It follows that the *generic-function class* and the *method class* of a generic function of purity p are objects of purity $p-1$.

For classes, the restriction implies that the slot metaobjects, the superclasses, the subclasses, etc. of a class of purity p should also be of purity p.

In general, we would like for a function of purity p to call other functions of purity p, but during bootstrapping we can not always accomplish this restriction. As a general principle, however, we want to minimize the exceptions to this rule, because these exceptions require specific bootstrapping code to be handled correctly.

D.2 Object creation

An object is created by a call to `make-instance`. Suppose we want to create an object of purity p. To do so, we need to instantiate a class of purity $p-1$. Instantiating a class involves calling `make-instance`, which is a generic function. But `make-instance` must call other functions. One such function `compute-defaulted-initargs` which computes the *defaulted initialization arguments*, given the class metaobject passed to `make-instance` and the initialization arguments passed to it. The function doing this computation must inspect the class metaobject in that it must call the accessor `class-default-initargs`. This latter function must therefore have purity $p-2$ since it takes an argument of purity $p-1$. If we assume that `make-instance` and the function for computing the defaulted initialization arguments have the same purity, then `make-instance` has purity $p-2$ as well. Now, values of keyword arguments passed to `make-instance` may be objects that will become part of the object being created, in which case, those values should have the same purity as that object, namely p.

Once the defaulted initialization arguments are computed, their validity must be checked. This task is accomplished by the function `check-initargs-valid`.

Once initialization arguments have been validated, `make-instance` calls `allocate-instance` in order to create the instance of the class.

Finally, the new instance must be passed to `initialize-instance` for initialization.

Object creation is illustrated in Figure D.1. As we can see, `make-instance` must call functions of two different purity values. For that reason, `make-instance` must be

handled specially during bootstrapping.

Another interesting aspect of `make-instance` is that, if it is given a symbol as opposed to a class metaobject, it must call `find-class`. Now `find-class` is probably an ordinary function, and it must find a class metaobject of purity $p-1$. If we assume that `find-class` has the same purity as `make-instance`, then we have a function of purity $p-2$ that must find a class metaobject of purity $p-1$. This information may determine in which environment we decide to allocate functions and classes of different purity.

D.3 Checking the validity of initargs to `make-instance`

Once the defaulted initialization arguments are computed, their validity must be checked. This task is accomplished by the function `check-initargs-valid`. This function call accessors on the class metaobject. In addition, it must inspect methods on `allocate-instance`, `make-instance`, `initialize-instance`, and `shared-initialize`. These functions do not all have the same purity. For that reason, we do not check the validity of initargs during bootstrapping.

D.4 Object initialization

Once an instance has been created, `initialize-instance` is called, and `initialize-instance` immediately calls `shared-initialize`. During bootstrapping, both these functions are generic functions. For that reason, when used to initialize an object of purity p, They must be defined in environment E_{p+2}.

D.5 Processing the `defclass` macro

The expansion of the `defclass` macro results in a call to `ensure-class` which is an ordinary function that immediately calls the generic function `ensure-class-using-class`. The *class* argument to `ensure-class-using-class` is either `nil` if the class does not exist, or the class metaobject to be reinitialized. Thus, if the new or the existing class is an object of purity p, then `ensure-class` and `ensure-class-using-class` should be of purity $p-1$. The AMOP states that the *direct-superclasses* argument to `defclass` becomes the value of the `:direct-superclasses` argument to `ensure-class` so there is no call to `find-class` involved here.

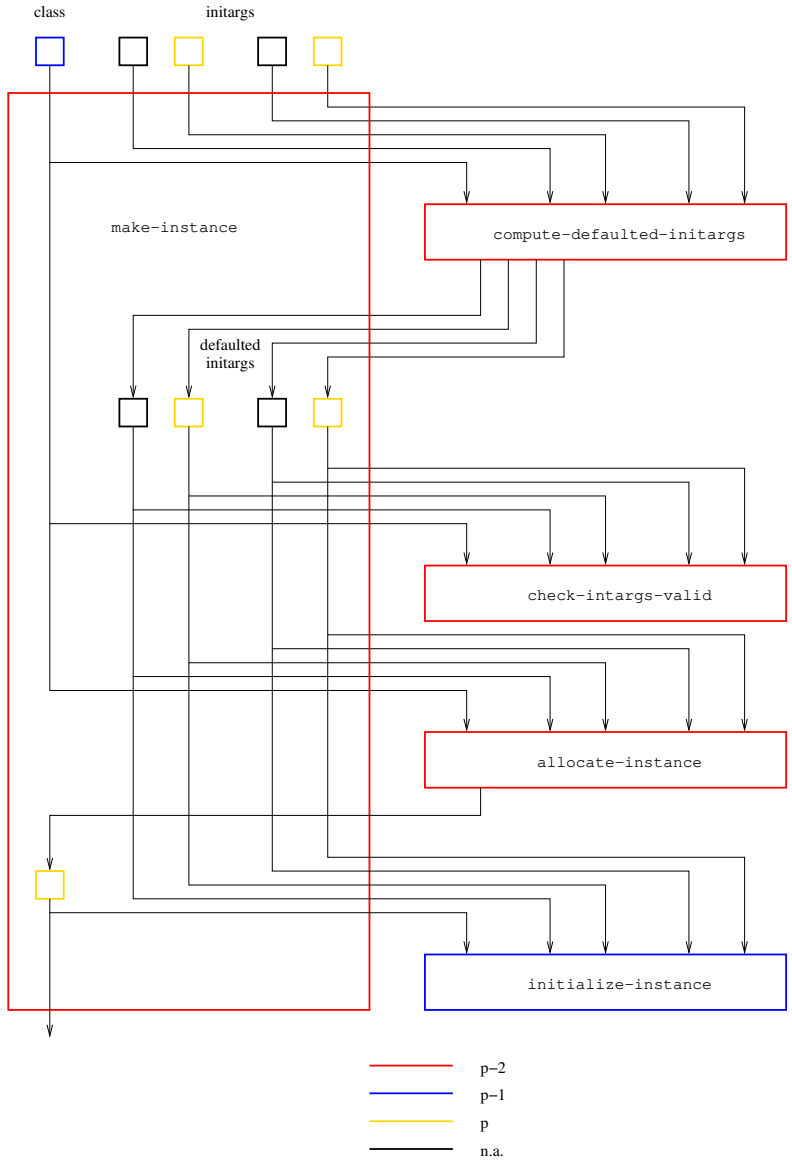

Figure D.1: Object allocation.

The most interesting aspect of the `defclass` macro is the conversion of a slot specification to a *canonicalized slot specification*, and specifically the conversion of the `:initform` option to the value of the `:initfunction` option. This conversion is done by `compile` that takes the initform wrapped in a `lambda` expression and turns it into a function. So `compile` in this case, must build a function of purity p since it is going to become part of a slot-definition metaobject of that purity.

The `:metaclass` option to the `defclass` macro becomes the value of the `:metaclass` keyword argument to `ensure-class`, so no conversion is involved.

The keyword argument `:direct-superclasses` passed to `ensure-class-using-class` may contain class names or class metaobjects. If it contains a class name, it is converted to a class metaobject. It must do so by calling `find-class` or something similar. So preferably, `find-class` has purity $p-1$ as well, and it must find a class with purity p. This behavior is consistent with the `find-class` we encountered in Section D.2.

The keyword argument `:metaclass` passed to `ensure-class-using-class` may contain a class name or a class metaobject. If it contains a class name, it is converted to a class metaobject. It must do so by calling `find-class` or something similar. In this case, `find-class` must find a class with purity $p-1$ which is in direct conflict with what it must do for the `:direct-superclasses` option. We solve this problem in SICL by using an indirection called `find-metaclass` that does what is needed.

D.6 Initialization of class metaobjects

In SICL, class initialization is accomplished by an `:around` method on `shared-initialize`. It calls an ordinary function to accomplish its task. If the class metaobject to be initialized has purity p, then `shared-initialize` has purity $p-1$.

The `:direct-slots` argument is a list of canonicalized slot specifications. Each element is converted to a direct slot definition metaobject. This conversion is done in two steps. First the generic function `direct-slot-definition-class` is called, passing the class metaobject as an argument. Therefore `direct-slot-definition-class` is a generic function of purity $p-1$, just like `shared-initialize`. Second, `make-instance` is called with the class returned by `direct-slot-definition-class`, and the canonicalized slot specification. So, here `make-instance` is called on a class of purity p. Therefore `make-instance` is of purity $p-1$.

D.7 Accessing slots

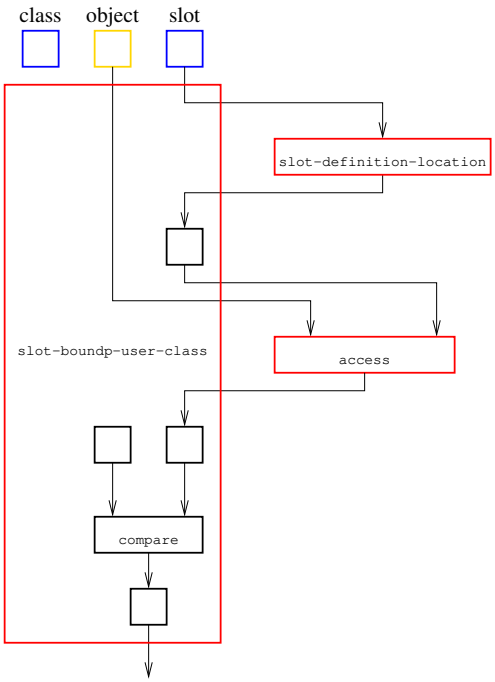

Figure D.2: `slot-boundp-using-class`.

Bibliography

[Ada18] Ulf Adams. Ryū: Fast float-to-string conversion. *SIGPLAN Not.*, 53(4):270–282, June 2018.

[BD96] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 108–116, New York, NY, USA, 1996. ACM.

[DS17] Irène Durand and Robert Strandh. Fast, Maintainable, and Portable Sequence Functions. In *Proceedings of the 10th European Lisp Symposium*, ELS '17, pages 64 – 71, April 2017.

[HL88] Bing-Chao Huang and Michael A. Langston. Practical in-place merging. *Commun. ACM*, 31(3):348–352, March 1988.

[HL90] Bing Huang and Michael A. Langston. Fast Stable Merging and Sorting in Constant Extra Space. Technical report, Knoxville, TN, USA, 1990.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011.

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic J. of Computing*, 3(1):27–40, March 1996.

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Ste90] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.).* Digital Press, Newton, MA, USA, 1990.

[Str14a] 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.

[Str14b] 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.

[Wat89] Richard C. Waters. XP: A Common Lisp Pretty Printing System. In *A.I. Memo 1102a, MIT Artificial Intelligence Laboratory*, 1989.

[Wat92] Richard C. Waters. Using the new Common Lisp pretty printer. *SIGPLAN Lisp Pointers*, V(2):27–34, April 1992.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

Index