# SBCL: a Sanely-Bootstrappable Common Lisp

Christophe Rhodes

Department of Computing
Goldsmiths, University of London
New Cross, London SE14 6NW
c.rhodes@gold.ac.uk

**Abstract.** This paper describes the development of an implementation of Common Lisp with the peculiarity that it is bootstrappable neither solely from itself, nor from some other language, but rather from a variety of other Common Lisp implementations. We explain the motivation for this bootstrap strategy, discuss some of the technical details involved in achieving it, and attempt to assess the technical and social effects that it has had on the development of the implementation and on Common Lisp users in general.

## 1 Introduction

The Lisp family of languages has a long history, being invented (or perhaps 'discovered') by John McCarthy in the late 1950s. Of the languages still in use today, only Fortran is older – though of course both modern Fortran and modern Lisp are worlds removed from the versions used in the 1950s. A wide range of dialects of Lisp were developed and disseminated in the following two decades, until in the 1980s moves towards consolidation between the various Lisp families occurred. The two most popular dialects of Lisp at the time of writing are Scheme and Common Lisp, both of which have international standards documents associated with them [1,2] (though in the case of Scheme there are also more lightweight community-led processes which have largely superseded the international standard [3,4]).

This paper is primarily concerned with implementation strategies for Common Lisp, and how those strategies affect the ease with which development of the implementations can occur. We do not address in this paper the implications of the details for other environments, presenting instead a case study of the social and technical effects observed in this single domain. In particular, we do not address the issues involved in cross-compiling Common Lisp for a different machine architecture, as these have been discussed elsewhere (see for example [5] and references therein).

Current Common Lisp implementations can usually support both image-oriented and source-oriented development. Image-oriented environments (for example, Squeak Smalltalk [6]) have as their interchange format an image file or memory dump containing all the objects present in the system, which can be later restarted on the same or distinct hardware. By contrast, a source-oriented environment uses individual, human-readable files for recording information for reconstructing the project under development; these files are processed by the environment to convert their contents into material which can be executed.

It would be unusual to find a Common Lisp application programmer these days working in an image-oriented manner; it is far more usual to work with source code stored in files and loaded using `compile-file`, than to define functions exclusively using the evaluator or read-eval-print loop and to store state by saving memory images, though the functionality of saving images is retained in contemporary Common Lisp implementations (despite not being part of standardized functionality) and is most often used as a deployment strategy.

Of course, Common Lisp application programmers are used to making incremental modifications to their software; Lisp environments are renowned for having the facilities to develop functions one at a time, coupled with the ability to use the image's introspective capabilities for finding information about callers and callees of functions and where variables are bound, for providing views of data structures (through an inspector or through more specialized browsers for classes, generic functions and the like), as well as for rapid recompilation and incorporation of modifications.

However, as image formats are not standardized, and indeed historically do change between releases of Common Lisp implementations, the application programmer is used to verifying from time to time that their current sources compile cleanly from scratch – that is, that no dependency on something which is only present in the image has been introduced in the sources.[1]

In the sphere of Lisp implementations themselves, however, this picture is reversed: it is somewhat unusual to find a Lisp implementation, written primarily in Lisp, which does not have a flavour of this image-oriented development within it. The typical build process in this case involves using a host lisp of the same implementation (but an earlier version), then mutating it incrementally to the point where it matches the new sources sufficiently to be able to compile those new sources, and then dumping an image. The mutation is in general different for each particular change at the source code level – many changes require no mutation at all, while changes to compiler-internal data structures may require very involved mutations: we give an example in section 4.1.

This paper discusses Steel Bank Common Lisp (SBCL), a Common Lisp implementation which is largely written in Lisp, while limiting and containing the image-based incremental modification of its own self as part of its build process to a strictly manageable level: the outcome of the build does not depend on the state of the host lisp compiler. The rest of this paper is organized as follows: in section 2, we describe the history and current state of Steel Bank

[1] A simple but real-world example of this comes from the abstraction of a syntactic pattern into a macro which has uses before its definition, because the most expedient place to put that definition was not in the first source file to be compiled from scratch. SBCL has a number of source files with prefix `early-` (for example, `early-package.lisp` and `early-setf.lisp`) for the purpose of holding definitions which must be seen early in the build.

Common Lisp; then in section 3, we go into the detail of how SBCL is built, comparing our approach with other Common Lisps. We discuss the benefits and drawbacks of this build process in section 4, and draw conclusions in section 5.

## 2 Steel Bank Common Lisp

Although Steel Bank Common Lisp (SBCL) is a relatively new Common Lisp implementation, it shares much code and a long development history with its closest relative, Carnegie-Mellon University Common Lisp [7] (CMUCL). CMUCL was a project funded by DARPA under CMU's "Research on Parallel Computing" contract, and began life as SPICE Lisp. Under that contract, CMUCL was developed continuously at Carnegie-Mellon University from the early 1980s until funding was stopped in 1994; at that point, CMUCL support at CMU was discontinued, but the project continues to this day, with a group of users and developers collaborating over the Internet.

SBCL was announced as a CMUCL variant with a 'clean' bootstrap process, in December 1999 by Bill Newman on the CMUCL developers' mailing list [8]. Since then, it has been developed further, initially by Newman alone, then with an increasing number of contributions from individuals, starting from the move to public CVS hosting on SourceForge in September 2000. The number of contributors has since risen significantly; at the time of writing, there are 23 people with commit privileges to the master CVS repository, while over the course of 2007 code contributions from over 40 people were incorporated.

The system as of early 2008 contains approximately

- 90,000 lines of lisp code implementing the 'standard library', excluding the Common Lisp Object System (CLOS);
- 60,000 lines of lisp code implementing the compiler (and related subsystems, such as the debugger internals);
- between 10,000 and 20,000 lines of lisp code per architecture backend implementing the code generators and low-level assembly routines;
- 20,000 lines of lisp code implementing CLOS;
- 20,000 lines of lisp code implementing contributed modules or 'extras';
- 35,000 lines of C and assembly code, for services such as signal handling and garbage collection;
- 30,000 lines of shell and lisp code for regression tests.

It is perhaps worth discussing briefly why there is a substantial component written in C and assembler: some 10% of the total. Partly this is because of the large number of architecture/operating system pairs supported; each such pair contributes some 200 lines of code implementing platform-specific operators (such as finding the faulting address from within a memory fault handler function); additionally, each supported operating system (of which there five) contributes 2000 lines, and each architecture (of seven) 2500 lines. The Garbage Collector is about 8000 lines of code, and is written in C for essentially pragmatic reasons: when the system is unstable enough for the GC to require debugging, using an external debugger (such as the GNU debugger, `gdb`) removes some uncertainty in the debugging process: and such external debuggers are better tailored to debugging C than Lisp.

We discuss the technical details of SBCL's build process in more detail in the next section; to give a high-level overview, SBCL's build achieves independence from the host lisp used to build it by embedding an SBCL compiler within the host, before using that embedded compiler to generate a fresh, standalone SBCL image. These two compilers, embedded and standalone, are generated from the same source code files; this works because we have effectively done the same as is commonly described as idiomatic Common Lisp programming style: to write a domain-specific language for solving one's problem, then solving the problem in that language – but in our case, the domain-specific language happens to be Common Lisp itself.

## 3 The SBCL Build Process

### 3.1 Build processes of other Lisps

We discuss the build processes and implementation strategies of other contemporary Common Lisps, in order to put SBCL's strategy in context. For more general information about these Lisps, see a recent survey of implementors conducted in late 2007 [9].

We can briefly summarize the implementation strategies of current Common Lisp implementations by dividing them into two categories: those which have significant portions implemented in languages other than Lisp, and those which are primarily Lisp-based. (The key to the division is whether there is enough implemented in the other language to implement an interpreter, or whether all Lisp evaluation is written in Lisp).

- Other-language based:
  - C implementation, C compiler: GCL, ECL (Kyoto Common Lisp derivatives)
  - C implementation, bytecode compiler: GNU CLISP
  - Java implementation, Java bytecode compiler: ABCL
  - C++ implementation, native compiler: xcl
- Implemented primarily in Lisp:
  - only buildable in themselves, using image-based techniques: Allegro Common Lisp[2], LispWorks[2], CMUCL, Scieneer CL[2], Clozure CL;
  - buildable in several Common Lisps: SBCL.

For the implementations where there is an evaluator in a non-Lisp language, the bootstrapping strategy is straightforward: building enough of an environment in that other language to be able to evaluate Lisp, and then build up the rest

[2] The closed-source nature of these implementations prevents the author from making any authoritative statement, but anecdotal evidence suggests that placing these implementations in this category is correct.

through successive evaluation. Note that this build strategy does not necessarily involve very much code in the 'other' language (see for example Lisp500[3], which has 500 lines of highly-obfuscated C); however, for systems intended for real-world usage, the figure is significantly higher: GNU clisp has 180,000 lines of 'D' – which is then preprocessed into C; gcl has 75,000 lines of C code implementing the compiler core, along with another 1,000,000 lines of C code from libraries for binary creation and accurate multiprecision arithmetic: binutils and gmp.

Part of our motivation for working on SBCL rather than other implementations is that we believe that Lisp is a good language for general programming, including the writing of interpreters and compilers and for manipulating complex data structures, and that therefore it would be a shame not to use it to the full in the development of a Lisp implementation: but the determinism granted by SBCL's build procedure as described in this paper is also important.

## 3.2 Building SBCL itself

The SBCL build process is not greatly dissimilar to the build process of compilers such as `gcc` [10, chapter 11]; there is an extra dimension to it, however, thanks to the accumulation of state during a compilation and loading process, which is not present in static C-like languages: this is the underlying reason for having namespaces beginning `sb!` in the cross-compiler (corresponding to `sb-` in a running SBCL); we discuss this further in section 3.3. The following diagrams illustrate the build process; in them, files either produced or used in the process are represented as ellipses, while Lisp processes themselves are rectangles.

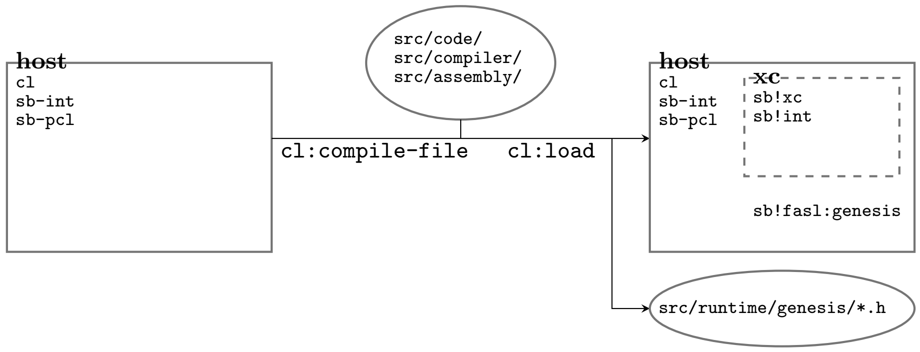

**Fig. 1:** The *host-1* build stage

The first step (figure 1) involves using the host compiler to compile and load SBCL's source files, which produces a cross-compiler (denoted by **xc**) running as

[3] Available at `http://www.modeemi.fi/~chery/lisp500/`. Lisp500 is not intended to be more than a 'toy' Common Lisp; its implementation is such that `eval` is written in C.

an application inside the host. This step also builds other applications, including `sb!fasl:genesis`, which will be used later. It is then possible to introspect over the definitions of the data structures for the target lisp, and produce a set of C header files describing Lisp data structure layouts.

These C header files, along with the C source and assembly files, are then used (figure 2) to produce the `sbcl` executable itself. The executable is as yet not useful; while it provides an interface to the operating system services, and a garbage collector, it requires a compatible Lisp memory image (produced in the next steps) to function. Additionally, a small C program is compiled and executed, generating Lisp source code describing system constants and types.

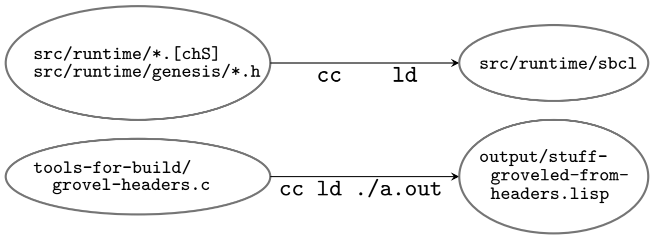

**Fig. 2:** The *target-1* build stage

Next, the cross-compiler version of `compile-file` is used to compile the SBCL sources again, along with the generated Lisp source file from the previous stage (figure 3). This produces a set of object files ('FASL files' in Lisp terminology, here given the filesystem extension `.lisp-obj`). Although in figure 3 the host and cross-compiler are displayed as though unchanged, in fact there are some fine differences between the cross-compiler at the start and the end of this process: the cross-compiler will have acquired new constant definitions (from the generated lisp file, for example). However, the functions and data structure definitions on the host, including those of the cross-compiler application, are unchanged by this process.

The next stage (figure 4) simulates the act of loading the FASL files and saving a memory image, using the `genesis` application. We cannot simply load those FASL files using the standard Common Lisp `load` function, because `load` uses the host compiler's FASL file format, not SBCL's format. Nor can we use the `load` function of the target image, because that target image does not yet exist: indeed, its creation is the purpose for wanting to load these FASL files in the first place.

Instead, we effectively build up the memory image by pseudo-loading each FASL file in the specified build order: we represent the memory areas of the target lisp as vectors of bytes, and perform the actions that would occur on loading the FASL file, not on the host lisp's data structures but rather on the representation of the appropriate memory area. There are actions that cannot be simulated in this way – these actions are deferred until the initial function of

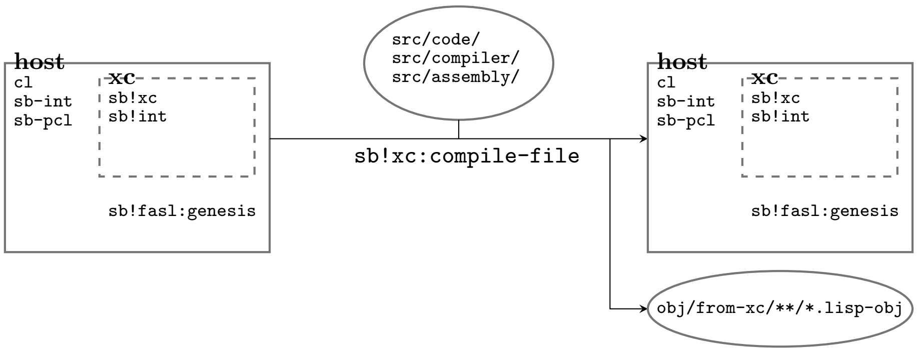

**Fig. 3:** The *host-2* build stage

the target image is run – but in particular function definition can be performed, so that the initial function is capable of calling other, named functions.

Once all of the FASL files has been processed in this way, the memory areas are saved to file in the format expected by the `sbcl` binary created earlier, producing a 'cold core'; there is a special case for dumping symbols of the `sb!xc` package, which are dumped as though they were in the `cl` package.

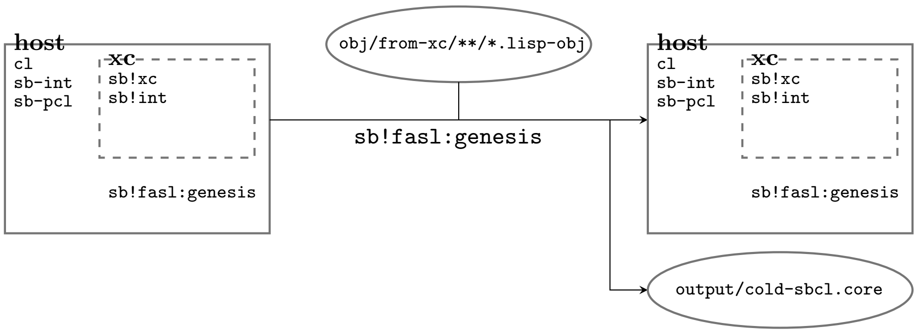

**Fig. 4:** The *genesis-2* build stage

The `sbcl` binary is run with the 'cold core' as its memory image: this core has a particular entry point or 'toplevel', which performs a sequence of actions to allow the processing of Common Lisp code. For instance, in the `genesis` sequence, no top-level forms from the FASL files have actually been run, because there is no way that they can be run using only the host lisp's facilities. Instead, they are deferred to this time. There are many other similar kinds of fixups which need to be performed at this time, and in a particular order; this 'cold init' phase is probably the most fragile portion of the SBCL build currently, and it is also the hardest to debug (because if it goes wrong, there will be no helpful Lisp debugger).

Finally, after the 'cold init' phase, the packages can be renamed to their final names (so that `sb!` prefixes are converted to `sb-`), and the specialized version of Portable Common Loops (PCL) for SBCL compiled and loaded; finally, a new memory image is saved as `output/sbcl.core`, and the build process is complete (figure 5).

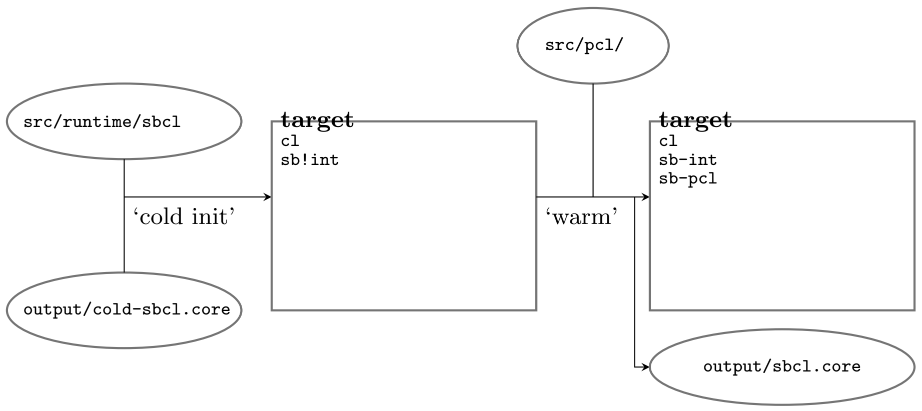

**Fig. 5:** The *target-2* build stage

## 3.3 Separation of host and target

During the SBCL build process, there needs to be a clear separation of the host and target worlds. In particular, if the SBCL build host is another version of SBCL, the build must neither use any properties of the host SBCL, nor mutate any of its structures.

To a large extent, this is achieved through one simple mechanism: packages which in a fully-built SBCL have prefix `sb-`, during the build have names beginning `sb!` – so that, for example, the package named `sb!kernel` during the build corresponds to the `sb-kernel` package in a running SBCL. This package name transformation occurs very early during the initial 'cold' boot of the lisp image dumped by the genesis phase of the build process.

The handling of the `*features*` variable is conceptually similar. The standard facility for conditional compilation in Common Lisp (really conditional code inclusion, somewhat like the C preprocessor's `#ifdef` construct) dispatches on the host's value of `*features*`, including or commenting out code based on feature expressions following `#+` and `#-` reader macros. The SBCL build provides analogous `#!+` and `#!-` reader macros for conditional reading of forms based on the target value of `*features*`, so that (for example) the cross-compiler can include architecture-specific optimizations or operating system specific system calls.

There is a complication, however, for handling routines that will be part of the `common-lisp` package in the target lisp: we clearly cannot overwrite any of the host compiler's functions or constants, but we need to be able to refer to target versions of these: for instance, so that the cross-compiler can compile `defmacro` forms. The solution there is to have a shadow `common-lisp` package named `sb!xc`, which only exists during the build process: operators which are needed for the build but which collide with `common-lisp` operators in the host lisp are placed there, and genesis has a special case for dumping symbols from the `sb!xc` package, so that the cold boot phase runs with all the necessary `common-lisp` operators already present.

To give a concrete example, the operator `cl:defconstant`, when evaluated by the host compiler, will define a constant in the host's world. The operator `sb!xc:defconstant`, when evaluated by the cross-compiler, will define a constant in the cross-compiler; further, when a call to `sb!xc:defconstant` is *compiled* by the cross-compiler, it will be as if `cl:defconstant` has been run, once the compiled call has been 'evaluated' during the genesis and cold boot phase.

As well as those simple special cases of host and target separation, there are a couple of slightly more complicated cases that nevertheless need some care. One is `eval-when`, which can cause confusion in even seasoned Common Lisp programmers. The `eval-when` operator allows the programmer to specify that certain forms should be executed when the file containing the form is file-compiled (`:compile-toplevel`) or when the resulting FASL file is loaded (`:load-toplevel`), or both – as well as when in a normal execution context (`:execute`, which is never active in the SBCL build process itself). In the context of phase separation and maintainability, the `eval-when` operator may not be the best solution [11], but it is available in conforming Common Lisps, so the SBCL build process can rely on it.

Consider the following example fragment:

```
(eval-when (:compile-toplevel :load-toplevel)
  (defun foo (x) (+ 7 x)))
```

Since the cross compiler cannot run any code of its own, for the `:compile-toplevel` case it must use `cl:defun` (*i.e.* the host's `defun`) here, not `sb!xc:defun`. However, for the `:load-toplevel` case, we are compiling a call that would eventually be executed by the target, so the cross-compiler's version of the macroexpander for `defun` must be used.

The other case revolves around `make-load-form`. The build process makes use of `make-load-form` extensively, for dumping compiler structures into FASL files. However, care must be taken here, because the host compiler's implementation of `make-load-form-saving-slots` is not necessarily compatible with the cross-compiler's – and yet the same code must be capable of dumping both compatibly for the host, while building the cross-compiler, and compatibly with the target, while building the target compiler. This is achieved by having a `make-load-form` method which dispatches on the presence of the `:sb-xc-host` on the *host's* lisp's `*features*` variable, which indicates which phase in the build is currently being executed.

### 3.4 Lisp library differences

There are other non-portabilities in the SBCL build process that are addressed at the time of writing to a greater or lesser extent.

Constant-folding and, potentially, type derivation in the cross-compiler will interfere with correct operation if the host Lisp's model of `float` subtypes is not the same as the target's. SBCL divides the `float` type into two subtypes: `single-float` (the same as `short-float`) for IEEE single floats, and `double-float` (the same as `long-float`) for IEEE doubles. Compiling from a host lisp which had different interpretations of `single-float` and `double-float` to SBCL's would be challenging, though note that SBCL's code is not sensitive to the host lisp's interpretation of `short-float` and `long-float`.

One observation remains: it is surprisingly difficult to write theoretically portable code for handling a large block of memory. Common Lisp provides the abstraction of a `vector`, of course, but the standard only specifies that the maximum size of a vector offered by the implementation must be 1024 elements or greater. While the author has not encountered a system where that limit is quite so low, implementations with an `array-total-size-limit` of the order of $2^{24}$ (on a 32-bit implementation where vectors are represented in memory with a header word consisting of an 8-bit type tag and 24 bits for the vector length) prompted a rewrite of the genesis phase to use 'big vectors', so that an in-memory data structure representing the bytes of the target lisp image (typically 20MB in size) could be built without falling foul of the array limit. The current 'big vector' implementation, representing linearly-addressible space as a vector of vectors of bytes, is in principle not sufficient, as the maximum space portably representable with such a data structure is 1MB (though in practice no implementation has an `array-total-size-limit` as low as 1024, and so our vector-of-vectors implementation is sufficient).

## 4 Discussion

### 4.1 Advantages

The immediate benefit of a straightforwardly reproducible build process is that no-one need learn the intricacies of the build process to contribute small, non-invasive patches. To a large extent, the author believes that this single fact is responsible for the current relative popularity of SBCL among Common Lisp implementations, and perhaps has even contributed to the increase in interest of Common Lisp as a whole.

Additionally, the straightforward build process, and in particular the clear separation between the build host and the target, allows for a smoother path for more invasive patches. As a simple example, there is no difficulty at all in renumbering the tags for type information, which is useful to allow more efficient assembly sequences for type checking.

In several cases the clear separation has resulted in bug fixes that were both cleaner and more straightforward to develop when compared to the more image-based Lisp implementations. For instance, there was a bug in both SBCL and its parent CMUCL in the handling of accessors for structures when name clashes occur through inheritance: a corner case to be sure, but one that was detected and fixed in December 2002 (SBCL) and January 2003 (CMUCL), by adding a slot to record inherited accessor information to the structure representing descriptions of defstructs.

It is in dealing with this kind of circularity that the simplicity of SBCL's build truly wins. In the case of SBCL, the fix was simple to implement: the slot was added to the sources, and then the sources were recompiled using the standard build procedure. In the case of CMUCL, however, in addition to changing the sources, two 'bootfiles' were necessary, and the system needed to be built at least twice: once loading the first bootfile beforehand (and interactively choosing a particular restart from an error condition); and once loading the second. The fix for the bug, which was very simple conceptually, was developed for SBCL by someone not in the development team; for CMUCL, it required an expert in the CMUCL build process itself to implement the bootfiles, demonstrating the principle that there can be a large impedance to contributions from newcomers.

Empirically, we can also say that SBCL as a whole, including its approach towards buildability, supports a community of users and developers which spans the gamut between experimentation with language and environment (examples include modular arithmetic, sequences [12], generic specializers [13]) and industrial use (as in ITA Software's QPX and RES products [9]). It should of course be noted that we cannot point to cause and effect here: there were all sorts of other factors allowing the SBCL (and Common Lisp) community to grow in number and scope, notably the development of a test suite for standardized functionality [14] and success stories from other Lisp vendors (see references in [9] for information current as of late 2007).

## 4.2 Downsides

The most obvious downside to the reworking of the build process described here is that each build inherently takes twice as long as in simple image-based systems: the compiler must effectively be built twice. These days, thanks to ubiquitous fast processors, this problem is much less evident than when the project started in 1999; the author remembers typical builds of upwards of an hour for SBCL, where its close relative CMUCL took on the order of 10 minutes (there were other factors for the more-than-doubling, including peak higher memory usage). Over the years, as processors have become faster, available memory has become greater, and SBCL's compiler has been optimized, there is only a small difference in build time[4]: and of course there is no need to think about *how* to build SBCL, or whether some kind of bootstrap script is needed – it can simply be set off.

Although a working model of SBCL is built as the cross-compiler, this is an incomplete model, and in particular it does not include CLOS (but it does include some hand-rolled object systems, in particular for implementing `subtypep`). SBCL's implementation of CLOS is derived from the Portable Common Loops [15] package, with many modifications to improve conformance with the description of the Metaobject Protocol for CLOS [16]. In particular, the implementation of CLOS has inherent metacircularities, and representing this metacircularity in a host-independent way has not yet been addressed. Allowing CLOS to be part of the cross-compiler would probably simplify some portions of the logic within SBCL's compiler and type inferencer, at the possible cost of making the bootstrapping procedure rather more complex.

## 5 Conclusions and Further Work

We believe that Common Lisp itself is well suited to the domain of Common Lisp compilers, and as such it is an appropriate technique to use Common Lisp as the implementation language for a Common Lisp implementation. We have further shown that it is possible to avoid a circular dependency: SBCL is written in Common Lisp, not in its own dialect.

We have presented evidence that this has some positive effects; the removal of the cognitive overhead in working out whether any given change to the system requires special measures to build allows both for more rapid development by individuals and for an easier path for newcomers to get involved with the system – a particularly critical requirement given the relative lack of popularity of Common Lisp these days, and the fact that the CL implementation market is fragmented.

One straightforward improvement to the self-sustainability of the system would be a compilation mode which would remove all non-deterministic elements from the FASL files produced by the cross-compiler (for example, the corresponding source code pathnames, or a build timestamp); this would allow for more straightforward testing of Common Lisp implementations which are currently not capable of building the system, and determining whether the problem lies in those implementations or an as-yet unidentified unportability in SBCL itself.

[4] On an Intel Pentium-M with clock speed 1.7GHz, building SBCL 1.0.16 using SBCL 1.0.15 took 541 seconds of user time, while building CMUCL 19d_p2 (without any extras such as CLX, the Motif bindings or Hemlock) with CMUCL 19d took 485 seconds of user time. Note that the CMUCL build involves three compilation phases: given sources corresponding sufficiently closer to the CMUCL compilation host, some of those phases may be unnecessary – but the default is to compile three times.

## Acknowledgments

Thanks are due first and foremost to Bill Newman, who began the process of producing Steel Bank Common Lisp in 1999, and who encouraged and restricted further development with impeccable taste. The many other collaborators on that project, too numerous to name here, have provided a superb community within which to do interesting and meaningful work. We also thank the reviewers for their helpful suggestions. The author acknowledges the support provided by a PPARC studentship and EPSRC research projects, which have allowed the indulgence of an enjoyable hobby.

## References

1. IEEE: IEEE Standard for the Scheme Programming Language. Technical Report 1178-1990, IEEE (1990)
2. Pitman, K., Chapman, K., eds.: Information Technology – Programming Language – Common Lisp. Number 226–1994 in INCITS. ANSI (1994)
3. Kelsey, R., Clinger, W., Rees, J.: Revised$^5$ Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation **11**(1) (1998)
4. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A.: Revised$^6$ Report on the Algorithmic Language Scheme. Technical report, `r6rs.org` (2007)
5. Brooks, R.A., Posner, D.B., McDonald, J.L., White, J.L., Benson, E., Gabriel, R.P.: Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp. In: Lisp and Functional Programming. (1986) 67–85
6. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. ACM SIGPLAN Notices **32**(10) (1997) 318–326
7. MacLachlan, R.: CMUCL User's Manual. Technical Report CMU-CS-92-161, Carnegie-Mellon University (1992) (updated version available at `http://common-lisp.net/project/cmucl/doc/cmu-user/`).
8. Newman, W.H.: It's alive! (SBCL, a CMUCL variant which bootstraps cleanly). Message-ID: `19991214185346.A1703@magic.localdomain` on `cmucl-imp@cons.org`; (December 1999)
9. Weinreb, D.L.: Common Lisp Implementations: A Survey. Available at `http://common-lisp.net/~dlw/LispSurvey.html` (2007)
10. von Hagen, W.: The Definitive Guide to GCC. Apress (2006)
11. Flatt, M.: Composable and Compilable Macros: You Want It *When*? In: International Conference on Functional Programming. (2002) 72–83
12. Rhodes, C.: User-extensible Sequences in Common Lisp. In: International Lisp Conference Proceedings. (2007)
13. Newton, J., Rhodes, C.: Custom Specializers in Object-Oriented Lisp. In preparation (2008)
14. Dietz, P.: The GNU ANSI Common Lisp Test Suite. In: International Lisp Conference Proceedings. (2005)
15. Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: Common Loops: Merging Lisp and Object-Oriented Programming. In: OOPSLA'86 Proceedings. (1986) 17–29
16. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)