The Next 700 Programming Libraries

Antonio Menezes Leitao
INESC-ID/Technical University of Lisbon
Rua Alves Redol nº 9
1000–029 Lisboa
Portugal
antonio.leitao@dei.ist.utl.pt

ABSTRACT
Modern software requirements are more diverse than before and can only be timely fulfilled via the extensive use of libraries. As a result, modern programmers spend a significant fraction of their time just mixing and matching these libraries. Programming language success becomes, then, more dependent on the quality and broadness of the accompanying libraries than on the language intrinsic characteristics.

In spite of its recognized qualities, Common Lisp lags behind other languages as regards the quality and availability of its libraries. We argue that the best solution to overcome this problem is to automatically translate to Common Lisp the best libraries that are available in other languages. In this paper, we specifically address the translation of Java libraries using the Jnil translator tool and we provide a detailed explanation of the problems found and the lessons learned during the translation of a large Java library.

Although many problems remain to be solved, the experiment proved the feasibility of the translation process and significantly increased our confidence in the future of Common Lisp.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: Reusable Software; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Languages, Economics, Documentation

1. INTRODUCTION
Once called “the most powerful programming language”, Common Lisp now faces uncertain future. Just like Latin influenced the Romance Languages and was ultimately replaced by them, all the recent newcomers to the programming language arena, such as Java, Python, Perl, or C# inherited a lot from the Lisp camp and are slowly but inexorably replacing Common Lisp.

The Java language, for example, was introduced with the explicit goal of becoming a near-universal programming language and, in order to be quickly adopted, its syntax was carefully designed to attract C++ programmers. On the other hand, Java semantics are closer to the Lisp camp than to the C/C++ camp. The biggest selling point for Java, however, is not its clearer syntax and semantics but, instead, its excellent libraries that cover a large fraction of the modern programmer needs, from graphical interfaces to socket programming. Similar libraries exist for C#, Python, Perl, etc.

Moreover, besides being technologically competitive with Common Lisp, some of these newcomers are backed by huge investments in good compiler support, IDEs, and marketing, making it difficult for any other less popular language to survive against such an overwhelming force. A final challenge comes from the Free/Open source movements and the Mob style of programming[9]: hordes of programmers working for free developed (and continue to develop) huge amounts of “freely” available libraries, modules, compilers or entire programs that can be used and reused in a “postmodernistic” [16] style to quickly solve the most common programming problems. This workforce quickly causes a network effect that makes popular languages even more popular and transforms less popular languages into niche-languages.

If modern languages such as C# and Java imitate many of Common Lisp technological features and provide much more libraries, the obvious question to answer is: why isn’t the language already dead? After all, many other good languages are dead.

Common Lisp’s survivability capabilities are the result of the extreme flexibility of the language. Extensible syntax, sophisticated object model, and powerful debugging features made Common Lisp highly adaptable to the changing environment and allowed it to survive several “cold ages.” These capabilities will also allow Common Lisp to survive in the near future but, in our opinion, that is not enough for the long term. It is our thesis that, in modern times, the success of a programming language is more dependent on the quality and broadness of its libraries than on the characteristics of the language itself.
Although it appears that Common Lisp is currently enjoying an upswing in popularity [29], it is our experience that as soon as the newcomers face the state of Common Lisp libraries and the difficulty in developing their own replacements, they return to what language they were using prior to their involvement with Common Lisp. We have seen this happen several times in the past and it has damaging effects in Common Lisp's credibility.

Unfortunately, due to its lack of critical mass, the Common Lisp camp doesn’t seem capable of quickly developing the necessary libraries in the foreseeable future. The other option is, obviously, to provide Common Lisp with libraries that were written for other languages. This has been repeatedly done in the past using several different approaches but, as we will show, without really solving the problem.

In this article, we will describe a different approach that we have recently used with good results. The approach is based on the semi-automatic translation from Java to Common Lisp and it has successfully (and quickly) produced several libraries for Common Lisp.

In the next section we will discuss the state of Common Lisp regarding the number and quality of its libraries. In section 3 we discuss approaches for the development of libraries and in sections 4 and 5 we propose our practical approach to quickly increase the number of available libraries for Common Lisp. This approach is demonstrated with a real example in sections 6 and 7, where we also discuss the problems found and possible solutions. Finally, in section 8 we will present our conclusions.

2. COMMON LISP LIBRARIES

The majority of modern programmers no longer think about their programs in terms of algorithms and data structures. Instead, they use broader concepts such as frameworks, design patterns, packages, protocols, and modules. This nomenclature reflects the fact that modern programming deals with much larger entities than it used to. In fact, the set of requirements for the majority of current software projects usually includes a large number of complex tasks (accessing databases, getting and parsing web pages, reading and processing files, etc.) that would be very hard to satisfy if it were not for the already existent libraries: modern (or postmodern) programmers spend more time mixing-and-matching large software modules than writing new algorithms. As a result, the availability and quality of those software modules becomes a critical part of every software project and is an important factor that affects the success of a programming language.

Regarding availability, it is relevant to mention that the emergence of the Web dramatically changed programming practices: nowadays, programmers extensively search the Web for libraries, documentation, programming examples, bug reports, etc. This activity is so common that the verb “Googling” was invented to describe it. Googling is the first step that most programmers do when they face a new problem. In general, the quicker they find a sufficiently interesting and well-documented library, the quicker they become dependent on it. This behavior is understandable because it requires effort to learn a library: a programmer that starts using one will have a tendency to avoid repeating that effort on a different one. It is thus important for the success of a library (and, by extension, of the language that supports it) to have enough Web visibility.

Are there enough libraries for Common Lisp? We can answer the question by making a comparison with a popular language, for example, Perl, according to the following dimensions:

Entry points Googling for Perl libraries returns CPAN as best match, an excellent entry point for everything Perl-related. Googling for Common Lisp libraries returns an incoherent set of links, some for Wikipedia articles, some for blogs, some for dispersed libraries. The Common Lisp Wiki www.cliki.net comes out in fifth place in the Google results and the excellent recent effort www.cl-user.net appears in 33th place, meaning that is practically invisible.

Libraries It is simply impossible to compare the number of available libraries in Perl and Common Lisp. While more than 10 000 Perl modules are available at www.cpan.org, less than 550 Common Lisp libraries are reported at www.cl-user.net and less than 300 Common Lisp ASDF-install libraries are available at www.cliki.net. It should be obvious that, in all cases, the number of high-quality libraries is only a small fraction of the total, but probabilistically speaking, there is a much better chance of finding one such high-quality library in the Perl camp than in the Common Lisp camp. Another interesting feature of CPAN is that it automatically runs the test suites that accompany the Perl libraries, thus helping preserve its quality.

Documentation In a large number of cases, there is absolutely no documentation whatsoever for the Common Lisp libraries. In most other cases, either the documentation is incomplete or is obsolete or both. In the Perl camp, there is excellent documentation for the most important libraries, including books, tutorials, examples, summaries, man pages, etc. Moreover, Perl recommends a documentation format that allows automatic extraction of documentation to a large range of markup languages, including HTML, $\LaTeX$, nroff, and man pages.

Examples Post-modern programmers frequently copy-and-paste code snippets to experiment and then adapt the software to their needs. It is usually the case that there are no such examples in the Common Lisp side. On the other hand, when examples are available, it is usually the case that it is impossible to use them without preparation such as some sort of (def)system manipulation, package creation, etc. These incantations are a show-stopper for programmers not used to the Common Lisp way.

Completeness Common Lisp libraries usually suffer from the 80% solution [19] where half a dozen programmers develop their own half a dozen 80% solutions to a common problem instead of working as a team and providing a 100% solution to the problem. The result is a
bunch of incompatible solutions that are continuously patched up to the point of becoming unusable.

**Stability** When a library that is being used has a problem, the usual (recommended) solution is to download a more recent version. It is frequently the case that the more recent version solves the problem but it is also frequently the case that the new version is partially incompatible with the previous version, forcing the programmer to update several parts of its own program.

It is a fact that some of these problems also affect other languages, including Perl, but, so far, it is our experience that it is highly difficult for a newcomer to develop modern (or post-modern) programs in Common Lisp and the main culprit is the lack and poor condition of Common Lisp libraries.

Another solution is to use libraries developed for other languages. To this end, several approaches have been attempted in the past. The most traditional approach is based in foreign function interfaces. DotScheme¹ [17] and RDNZL² indirectly link Scheme and Common Lisp, respectively, with the .NET platform by directly linking with a special library (written, e.g., in Microsoft Managed C++) that runs on that platform. Jfli³ does more or less the same for the Java platform, but using the Java Native Interface (JNI).

Unfortunately for Common Lisp, foreign functions are not standardized and this makes it hard to use them across the entire range of Common Lisp implementations. Each Common Lisp implementation has its own specific (and, frequently, incomplete) means to access foreign functions and the efforts to provide a unified interface, namely UFFI⁴ and CFFI,⁵ could only produce a sort of common denominator for the functionally provided by all implementations. Moreover, there are a few complex technical issues with object representation, GC synchronization, method dispatch, callbacks, etc., that makes the use of this approach significantly harder. Also, modern object-oriented frameworks allow the programmer to extend them by subclassing certain pre-defined classes, an operation that is usually beyond the capabilities of Common Lisp foreign function interfaces.

Another option is to compile from Common Lisp to the platform of the library. ABCL⁶ is a Common Lisp compiler that targets the JVM and that is also capable of translating calls to the runtime APIs thus providing good integration between Common Lisp and Java libraries. Unfortunately, this makes the software completely dependent on a specific Common Lisp implementation and, moreover, it makes it very hard to use libraries written in other languages besides Java.

The easiest but least efficient solution is to establish a communication channel (a socket, in Unix parlance) between the Common Lisp application and the intended platform. Foil⁷ does this both for .NET and the JVM. JLinker⁸ does it just for the JVM. There are also a few hard-to-solve-efficiently problems, such as preserving object identity across the communication channel, synchronizing the GCs in the two independent processes, etc. Also, to access the foreign language APIs, it is necessary to use reflection that, combined with the marshalling required to transport objects across the communication channel, causes serious performance problems.

Besides the above mentioned problems, the biggest difficulty with all these approaches is that they force us to program in a mixed model, reading library documentation for one language and mentally translating it to Common Lisp in order to make the necessary incantations in the Common Lisp side. For example, consider the following fragment of a Common Lisp program in RDNZL that accesses .NET libraries:

```
(defun download-url (url)
  (let ((web-client (new "System.Net.WebClient")))
    [GetString (new "System.Text.ASCIIEncoding")
      [DownloadData web-client url]]))
```

Note the use of strings (that will be used in the reflection layer) and special syntax (the square brackets) that causes a clear separation between the Common Lisp parts and the .NET parts. As another example, let’s now look at a fragment in ABCL that accesses the JDK:

```
(jcall (jmethod "java.lang.Runtime" "availableProcessors")
  (jstatic "getRuntime" "java.lang.Runtime"))
```

Again, note the use of strings and also the explicit calls to the reflection machinery.

In the case of UFFI and CFFI, the situation looks slightly better but only because the target language (C) is so simple that it is easier to hide it behind Common Lisp structures and functions.

Debugging such mixed-language programs is also very demanding as it requires the programmer to simultaneously deal with two different languages and to understand the subtleties of the interface between them.

¹http://rivendell.ws/dot-scheme/
²http://weitz.de/rdnzl/
³http://jfli.sourceforge.net/
⁴http://uffi.b9.com/
⁵http://common-lisp.net/project/cffi/
⁶http://armedbear.org/abcl.html
⁷http://foil.sourceforge.net/
Due to the syntactic, semantic and pragmatal differences between Common Lisp and other languages, it is our thesis that any approach that preserves the form of the library will always be difficult to use and will look unnatural for the seasoned Common Lisp programmer. Aesthetics is an important concept in programming and all the previous approaches fail in this point.

When other approaches fail, the last resort is to simply translate the intended libraries to Common Lisp. However, hand-translation to Common Lisp of libraries written in other languages is extremely time-consuming and error-prone. The only way to speed up the task and minimize the errors is to automate the translation.

4. AUTOMATIC TRANSLATION TO COMMON LISP

Several attempts have been made in the past to automatically or semi-automatically generate Lisp programs via translation of programs written in other languages.

Pitman [18] developed a Fortran to Lisp translator that used a Fortran Virtual Machine implemented as a set of MacLISP macros. The translator generated a MacLISP program containing calls to those macros that were expanded into native MacLISP constructs. In order to emulate Fortran call by reference, variables were allocated in a global array and represented by indexes in the array. The pre-macro expansion MacLISP code could be considered as a syntactical mixture of Fortran and MacLISP that both Fortran programmers and MacLISP programmers would understand.

The f2cl tool [4, 8] is another translator that converts Fortran to Common Lisp and that explores different strategies for implementing call by reference and Fortran arrays.

Smalltalk is also a good source of libraries for Common Lisp. Babel [14] is a prototype translator that converts Smalltalk into portable CLOS. The original goal was to be able to run Smalltalk applications but without requiring the presence of the complete (and large) Smalltalk environment. The goal was achieved by translating the Smalltalk application as well as all the parts of the Smalltalk environment that were necessary to run the application.

Using Babel, Smalltalk classes are translated into CLOS classes and Smalltalk methods are translated into CLOS methods. A more complex situation occurs with Smalltalk classes and methods that depend on Smalltalk primitives. In this case, replacement classes are used. These classes emulate the primitive Smalltalk behavior and they correspond directly to some primitive Common Lisp type or they are implemented by hand-translating the primitive Smalltalk behavior.

For performance reasons, Babel takes particular care in the translation of Smalltalk literal blocks that occur as arguments to the methods that implement the control-flow operators, such as ifTrue, whileTrue, etc. To speed up the performance of the generated code, these blocks are eliminated and the equivalent Common Lisp control-flow operators are used with the block code inlined.

<table>
<thead>
<tr>
<th>library</th>
<th>Fortran</th>
<th>Smalltalk</th>
<th>Lisp</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 350</td>
<td>913</td>
<td>623</td>
<td>1 070</td>
<td>724</td>
</tr>
<tr>
<td>1 260</td>
<td>1 080</td>
<td>1 70</td>
<td>328</td>
<td>559</td>
</tr>
<tr>
<td>1 260</td>
<td>1 260</td>
<td>328</td>
<td>674</td>
<td>1 080</td>
</tr>
<tr>
<td>1 260</td>
<td>1 260</td>
<td>328</td>
<td>674</td>
<td>1 080</td>
</tr>
</tbody>
</table>

Table 1: Googling for language and keywords as of 2006/12/16 (in thousands of hits)

Unfortunately, the generated Common Lisp code contains numerous Smalltalk-like features, such as special boolean values and unnatural method names. In practice, the generated code is unmaintainable by a Common Lisp programmer.

4.1 The Case for Java

All the previously described translators have an additional problem: they target libraries in languages that also suffer from the same problem that Common Lisp suffers: lack of critical mass. Even in the more useful case of Babel, there are not enough Smalltalk programmers nor sufficient investment in the language to provide a large set of libraries for the modern programming tasks. If Common Lisp wants to use libraries developed for other languages, it is preferable to find a language that already has a large number of libraries and that has also enough critical mass so that even more library development can be expected. To better understand this issue, Table 1 presents the number of results collected by “googling” for the name of the language in column titles combined with the keywords in the line titles, e.g., searching for “Fortran date time library” returned 724000 results.9

It is obvious that the large majority of the results should be irrelevant for the intended search but they do show that there is a much higher probability that a useful library can be found in the Java camp than in the Fortran, Smalltalk or Lisp camps. Even for numerical libraries, a traditional Fortran feud, Java has a larger number of results. For the generic library query, the results are simply overwhelming.

Another advantage of using Java as starting point for the translation process is that the language has so much critical mass that not only there are lots of libraries but the libraries also compete with each other for the largest acceptance. A nice side effect is that the best libraries come with good documentation and, in many cases, extensive unit testing. This is a consequence of two different factors:

Documentation Java promotes a stereotyped form of program documentation known as Javadoc[11]. It is arguable whether Javadoc is the best tool for the job but what is not arguable is that Javadoc stimulated Java programmers to comment their code, a practice that is missing in many other languages, Common Lisp included.

Unit Testing Although initially developed for Smalltalk, the unit testing framework Sunit [1] was ported to Java

9In the search, we used “Lisp” instead of “Common Lisp” in order to increase the number of results via the inclusion of Emacs Lisp libraries.
and renamed Junit [3]. JUnit is fundamental for Extreme Programming [2], a development process that is much more successful in the Smalltalk (and Common Lisp) camps than it can possibly be in the Java camp. Nevertheless, JUnit became the de facto standard for testing Java programs and the majority of the best Java libraries include a large number of unit tests.

These two factors are very important for the translation of Java libraries to Common Lisp because (1) they provide a starting point for the documentation of the Common Lisp libraries and (2) the collection of unit tests can be also translated and used to detect translation problems. Even more important is the fact that the unit tests allow us to check that no errors are introduced in any subsequent refactoring process, e.g., to adapt a translated library to the Common Lisp look&feel.

A final advantage of using Java as a source of libraries is that the language has many similarities with Common Lisp, meaning that it might be possible to translate a Java library to Common Lisp while maintaining the original library structure without necessarily destroying the aesthetics of the Common Lisp program using it.

We will now describe Jnil, a translator from Java source code to pragmatically correct Common Lisp code that we have been using to generate Common Lisp libraries from Java libraries.

5. JNIL
Jnil is an automatic translator from Java to Linj and Common Lisp. Jnil was developed by Tiago Maduro-Dias with guidance and help from the author and is available at http://common-lisp.net/project/jnil/. Jnil was initially developed to provide round-tripping between Linj [5] and Java. Linj is a programming language that was carefully designed to be very similar to Common Lisp but whose semantics was guided by the Java language specification. Although Linj implements some of the most used features of Common Lisp (including read-macros, macros, optional and keyword arguments, standard method combination, multiple inheritance and many others), it also has the ability to represent everything that can be written in Java 1.4 and, in fact, Linj includes several features that might not make any sense in Common Lisp but that are essential to generate good-looking Java code.

Linj can represent (in a Common Lisp-like syntax) whatever is representable in Java and this explains Jnil capability to translate Java programs into Linj. In simple terms, Jnil operates by walking the annotated abstract syntax tree of a Java program and by associating to each tree node a function that constructs an appropriate S-expression made of the node information and of the S-expressions of the subnodes. Different passes associate (and compose) different functions and a final pass pretty prints the S-expressions. The abstract syntax tree is provided by Eclipse [10] and Jnil acts as an Eclipse client, using JLinker for the interprocess communication. All the required semantic information (use-def and def-use relations, type information, etc) is provided by the excellent Eclipse JDT [7]. A more detailed explanation of Jnil will be provided in a forthcoming paper.

Ideally, Jnil could be used to reconstruct Linj sources that were translated to Java and then further modified by Java programmers. However, due to the generalized use of macros in Linj and to some specific Common Lisp features that Linj also implements, such as keyword parameters, multiple inheritance and double dispatch methods, what Jnil can actually reconstruct is an “expanded” version of the original program. The reconstructed Linj program is semantically equivalent to the original Linj program but at a lower abstraction level. In order to be helpful for round-tripping between Linj and Java, Jnil must be selectively used and only for the translation of small parts that can then be manually improved to recover the original abstraction level.

In spite of its limitations, Jnil can be very helpful to provide a starting point for the construction of Linj libraries from Java source code. Obviously, given the similarities between Linj and Common Lisp, it was very tempting to also use Jnil to translate from Java to Common Lisp. To this end, Jnil was supplemented with some specific passes oriented towards the generation of Common Lisp code.

To determine Jnil capabilities for the effective translation between Java and Common Lisp we experimented the translation of a relatively big Java library: Joda Time. Joda Time is the top result for a Google search for “Java date time library” and is one of the best Java APIs for dealing with dates, times, intervals, periods, calendars, and timezones. Besides translating Joda Time, we also translated the huge set of unit tests that were developed for Joda Time, allowing us to check the correctness of the translation.

6. DATE&TIME LIBRARIES
As stated in the Hyperspec, “Time is represented in four different ways in Common Lisp: decoded time, universal time, internal time, and seconds.” These are abstract representations because, in practice, the few Common Lisp date and time functions accept and return numbers. Besides encoding and decoding calendar times, no more standardized operations are provided. Dates and times are one of the obvious libraries that could benefit Common Lisp.

Fortunately, we already have lots of Date&Time libraries for Common Lisp. For example, CLSQL includes one such library for its internal use containing more than one hundred operations for creating, comparing and operating dates and times. LOCAL-TIME is another similar library based on [15]. Metatilities and net.talent.date provide date and time parsing and printing. Cl-l10n also include operations for properly reading and writing localized dates and times.

Now, let’s suppose we have an application that already uses net.talent.date but we also want to use the Common Lisp Web Framework cl-wdim: among other libraries, it depends on CLSQL, on CL-Graph (that depends on Metatilities), and Cl-l10n. The result is that we will get not one but, at least, four different Date&Time libraries on the same application. As time goes by, the redundancy seems to increase. For example, the recently released library Postmodern (for interacting with Postgres databases) also includes its own Date&Time sub-library.
The reason for this situation is, in our opinion, a combination between the NIH (Not Invented Here) syndrome and the 80% solution phenomena. Dates and times seem such simple concepts that everybody prefers to implement its own version. The result is that several Date&Time libraries emerge but they implement only part of what is needed, and they do so in incompatible ways: it is not possible to feed CLSQL concepts that everybody prefers to implement its own version.

To make things worse, as was already acknowledged in [6, 15], dates and times are simple only at first look. Dealing with leap years, calendar adjustments, time zones, different calendars, daylight savings time and a large number of different formats for reading and printing makes dates and times much more complex than they seem. Moreover, there are several important abstractions that need to be considered, namely, instants, partial instants, intervals, durations and periods. None of the available Common Lisp Date&Time libraries addresses all these features.

Any new Date&Time Common Lisp library that does not implement all the required features is simply a waste of time. Unfortunately, this is a task that cannot be done in a short time frame, unless we are willing to use the very complete libraries that are available for other popular languages.

In the Java world, there is one Date&Time library that stands out: Joda Time. Joda Time is an open-source, BSD-style licensed quality replacement for Java Date and Time classes. The library is easy to use, easy to extend, provides multiple calendars and has good performance. Another interesting feature is that it provides its own compiler for the public domain time zone tz database thus becoming independent from operating system services for dealing with time zones. Finally, Joda Time provides good internal and external documentation, including user guides, tutorials, and FAQ.

Given the completeness and quality of Joda Time, it was obvious to consider it as the starting point for the development of a Date&Time library for Common Lisp.

In the remaining of the article we will describe the steps taken for the translation of Joda Time to Common Lisp, the problems detected, the lessons learned and the changes that we plan to make in the translator in order to make it more useful to speed up the effective translation of other libraries.

7. TRANSLATING JODA TIME TO COMMON LISP

The first step for the translation of any Java library to Common Lisp is to create a project for that library in the Eclipse IDE and to ensure that the library is properly compiled by Eclipse. This is a fundamental step because Jnil can only translate libraries that do not contain any syntactical errors. Care must be taken to include all the dependencies in the project so that all names can be resolved.

The second step is to start a tiny Eclipse plugin that implements the Java end of the jLinker connection. Jnil will be on the Common Lisp end of the connection.

The third step is simply to invoke the Jnil compiler indicating the project, package, or compilation unit that is to be translated, the target language (Linj or Common Lisp) and some translation options that force the translator to be more conservative or more aggressive.

We will now discuss a selected set of problems that we found in the translation of Joda Time from Java to Common Lisp.

7.1 Packages

The first problem found is the use of packages. In Java, packages isolate type names so that they don’t conflict with identical type names in other packages. In Common Lisp, packages isolate names, independently of their meaning. Besides the packaging mechanism, Java also includes an orthogonal access control mechanism that, based on accessibility declarations (e.g., private), allow or deny access to the members of packages or classes.

Given these differences, we can’t provide a mapping between Java packages and Common Lisp packages but we can construct a good initial approximation just by collecting the names of all the public members of packages and classes and by automatically creating a package definition that exports those names (taking care of shadowing those that conflict with the “standard” base package). Obviously, the package must be manually adjusted to use the necessary packages, to export only the correct names and to solve all name conflicts that might arise.

Besides package declarations, Java also uses import declarations. Superficially, Java import declarations look similar to the use or :shadowing-import-from clauses of a defpackage form but, in fact, they are much harder to deal with because import declarations are specified on a compilation unit basis, i.e., in practice, file by file. This means that two different Java files that belong to the same package might have different (and conflicting) import statements and, in general, they cannot be merged in order for its inclusion in the appropriate Common Lisp package definition. Moreover, they usually represent dependencies that must be manually solved by the Common Lisp programmer. For these reasons, Jnil doesn’t attempt to translate the import declarations, leaving them as comments so that the programmer can decide what to do with them.

7.2 Classes and Interfaces

Java classes and interfaces are translated into CLOS classes containing as slots all the non static fields. All field qualifiers (private, final, transient, etc) are simply eliminated. Type information is also discarded. Getters are identified so that they can be treated as readers. Setters are not converted to writers because using writers imply a different order of arguments (or a different call syntax, in the case of setf writers) and Jnil does not know, in the general case, which calls will resolve to a particular setter. However, when explicitly requested, the writers are generated even though the generated code is incorrect.

As an example of the translation of Java classes, consider the following Joda Time PeriodFormatter class:
public class PeriodFormatter {
    private final PeriodPrinter iPrinter;
    private final PeriodParser iParser;
    private final Locale iLocale;
    private final PeriodType iParseType;
    
    public PeriodPrinter getPrinter() {
        return iPrinter;
    }
    public PeriodParser getParser() {
        return iParser;
    }
    public Locale getLocale() {
        return iLocale;
    }
    public PeriodType getParseType() {
        return iParseType;
    }
    
    The translation to Common Lisp is the following:

    (defclass period-formatter ()
        ((i-printer :initform nil :reader get-printer)
         (i-parser :initform nil :reader get-parser)
         (i-locale :initform nil :reader get-locale)
         (i-parse-type :initform nil :reader get-parse-type)))

7.3 Class Constructors
At the moment, Jnil does a poor job at translating class constructors. Although it can be configured to preserve the Java semantics, (in which case it generates a different function to specify the correct “constructor” responsible for the initialization of the object) it is preferable to simply left the constructor code to the user consideration. In some cases, it is trivial to eliminate such constructors and manually add :initarg and :initform options to the class definition. This is clearly visible in the two constructors provided in the PeriodFormatter example:

public class PeriodFormatter {
    ... public PeriodFormatter(PeriodPrinter printer, PeriodParser parser) {
        super();
        iPrinter = printer;
        iParser = parser;
        iLocale = null;
        iParseType = null;
    }
    private PeriodFormatter(PeriodPrinter printer, PeriodParser parser,
        Locale locale, PeriodType type) {
        super();
        iPrinter = printer;
        iParser = parser;
        iLocale = locale;
        iParseType = type;
    }
    ...}

Translating just the first constructor produces the following:

#<LISP-SEXP (call-next-method) @ #x787f22c2>
(setf (slot-value this 'i-printer) printer)
(setf (slot-value this 'i-parser) parser)
(setf (slot-value this 'i-locale) nil)
(setf (slot-value this 'i-parse-type) nil)
@ #x78817882>

Note that when Jnil doesn’t know how to generate correct Common Lisp code, it generates Linj code instead but marks it as unreadable to alert the programmer that the code needs further consideration.\(^\text{10}\)

In the particular case of the period-formatter class, all that remains to be done is to manually eliminate both constructors and to provide extra :initarg options in the defclass form. Obviously, it then becomes possible to construct instances of period-formatter in more ways than was possible with the Java version. It is up to the programmer to decide if that is an advantage or not.

The elimination of constructors become particularly interesting when the Java code explores the ability to construct an object in several different ways. It is known that this ability quickly degenerates in the Java tendency for constructor madness. As an example, the Joda Time class BasePartial and its subclass TimeOfDay provide a total of 9 + 15 constructors. A large subset of these constructors simply implement the equivalent to optional or keyword parameters. Others are there just for the sake of repeating in the subclass the same constructors that were defined in the superclass. After manual analysis and refactoring, they ended up replaced by a combination of :initargs, :initforms, :default-initargs, and two initialize-instance specializations, significantly reducing the code size and, at the same time, allowing even more construction possibilities.

7.4 Methods
Methods defined in a Java class are translated to CLOS methods whose first parameter is specialized in the corresponding CLOS class. This, obviously, is not equivalent because the Java method belongs to a class while the CLOS method belongs to a generic function. Moreover, CLOS requires the lambda list of the method to be congruent with the lambda list of the generic function and this frequently causes problems. Finally, Java employs method overloading that must not be confused with multiple dispatch. Currently, when Jnil finds overloaded methods, it adopts a conservative strategy based on combining the name of the method with the type of its parameters, allowing correct code generation but frequently requiring further human intervention to improve the code quality. Consider the following example:

class TimeOfDay extends ... {
    ... public DateTime toDateTimeToday() {
        return toDateTimeToday(null);
    }
    public DateTime toDateTimeToday(DateTimeZone zone) {
        ...}
    ...}

and its translation into Common Lisp:

\(^{10}\)Jnil can also be configured to generate readable but non-executable code for such cases. It is thus possible to choose whether to find translation errors at compile time or at run time.
(deffn to-date-time-today-time-of-day (...) (...))
...
(defmethod to-date-time-today
  (((this time-of-day))
    (to-date-time-today-date-time-zone this nil))
)...}

In this example, it is clear that overloading is being used to provide a default for the parameter zone. In this case, we can collapse both methods into one by making the parameter optional. Care must be taken to ensure that our modifications do not disturb method redefinitions in lower levels of the class hierarchy. Also, all the relevant method calls must be adjusted.

A more serious (and recurrent) problem occurs when our changes force completely unrelated but named methods to clash into the same generic function. In this case, either we resort to packages to distinguish the different generic functions or else we rename one of them.

### 7.5 Static Methods

Java static methods are much more similar to Common Lisp functions than to Common Lisp methods. As a result, as a first approximation, Jnil translates them to Common Lisp defun forms. However, there is one important difference: Java static methods are scoped by the class containing them. To avoid conflicts, Jnil uses for the name of the Common Lisp function the concatenation of the class name with the static method name (converted to Common Lisp naming conventions). This frequently causes the definition of functions with excessively large names and a further refactoring should be done to simplify those names.

#### 7.6 Statements

In general, Jnil is capable of mapping the control-flow operators of the Java language into equivalent operators in Common Lisp. Java if statements are translated to Common Lisp if forms, Java statement blocks are translated to Common Lisp progns, etc. In other cases, a combination of forms is used. For example, a try-catch statement is translated to an handler-case, a try-finally is translated to an unwind-protect and a try-catch-finally is translated to a combination between unwind-protect and handler-case.

Unfortunately, when the Java code employs control structures that do not have direct correspondence in Common Lisp, the resulting code might look unnatural. For example, the following Joda Time fragment:

```java
int limit = text.length();
for (; i<limit; i++) {
  char c = text.charAt(i);
  if (c < 'A') {
    break;
  } else if (c >= 'a' && c <= 'z') {
    Character.isLetter(c) {
      continue;
    }
  }
  break;
}...
```

gets translated into

```lisp
(let* (((limit (length text)))
        (i position))
  (loop while (< i limit)
    do (block 04485
      (let* ((c (char text i)))
        (when (char<= c #\A)
          (return))
        (when (or (or (and (char>= c #\a) (char<= c #\z))
                      (alpha-char-p c))
                   (return-from 04485))
          (return)))
    (incf i)))
)
```

Note that an inner block was introduced to replace the continue statement with a return-from. These combinations of forms have a tendency to make the code less readable and, to avoid maintenance problems, it is preferable to carefully refactor it. Another possibility would be to introduce new Common Lisp macros that implemented the Java control-flow operators. We considered this option but we didn’t follow it because we want to generate Common Lisp code that can be improved by a Common Lisp programmer without forcing him to work in a mixed language model. Refactoring is a more expensive operation in the short term but the results are much better from the point of view of future maintenance.

One Java statement that deserves a bit more attention is the return statement. Common Lisp is an expression-based language where the value of the expression at the end of the control flow is treated as the function returned value(s). Java, on the other hand, is statement-oriented and requires explicit return statements to identify the value returned. Although it can be semantically equivalent to use Common Lisp return-froms in place of Java returns, the code looks very unnatural. To prevent this problem, we extended Jnil to do some very basic control flow analysis to minimize the number of return-from statements. Here is one Joda Time example where the improvement is visible:

```java
public DateMidnight plusYears(int years) {
  if (years == 0) {
    return this;
  }
  long instant = getChronology().years().add(getMillis(), years);
  return withMillis(instant);
}
```

is translated into

```lisp
(defmethod plus-years ((this date-midnight) years)
  (/+ (let* ((instant
              (add-long-int (years (get-chronology this)))
            (get-millis this years)))
       (with-millis this instant)))
)
```

Note that the second return statement was removed because Jnil noticed that it was at the end of the control flow. The first one, however, had to remain. Obviously, the code still doesn’t look pragmatically correct but that is a problem that is now easier to solve using a refactoring tool. We have used C3PO [12] for this task because it can learn refactoring rules from a manual refactoring example and then the
transformation can be repeatedly applied to other matching fragments.

7.7 Expressions
As was already visible in the previous examples, Java method calls are translated into Common Lisp (generic) function calls. When the method call receiver is implicit in Java, Jnil makes it explicit via the method specialization parameter **this**. Associative combinations of binary primitive operators such as `&&` and `||` could be translates to their variable arguments counterparts in Common Lisp but, at the moment, this is not done, thus requiring a further refactoring step to make the code look nicer.

Regarding arithmetic expressions, Jnil simply translates from the Java operators to the corresponding Common Lisp arithmetic functions. At the time, we thought this would be the most sensible translation because it would generate good looking code and the Common Lisp numeric tower would be a semantic improvement over Java primitive types. Initial translation experiments increased our confidence that this was the right decision: Java programs that computed wrong answers when the size of the manipulated numbers exceeded `int` or `long` capacities were working flawlessly in Common Lisp due to the wonders of **bignums**.

After the translation of Joda Time we now consider this decision as one of our biggest blunders. The problem is that several Java programs, including Joda Time, depend on modular arithmetic. Similar problems occur with integer division (that truncates in Java but might produce ratios in Common Lisp) and shifting. It is safe to say that most of the time spent in the translation of Joda Time to Common Lisp was wasted searching for these kinds of bugs.

To solve this problem we had to temporarily shadow all Common Lisp arithmetic operators with debugging versions that warned when the result exceeded the bounds of Java `long` sizes. We then ran a large set of unit tests to identify the problematic fragments where modular operators must be used. At the end of this debugging phase (that is not finished yet) we will revert to the standard Common Lisp operators.

Another solution would be to always use modular versions of the arithmetic operations as this would allow the software to run with the exact same semantics as it did in Java. Except for the specific cases were modular arithmetic was an absolute requirement, we did not follow this route because we want the final code to represent a proper Common Lisp library and not an emulation layer where Java libraries can run. Moreover, this route usually entails a performance penalty that can only be reduced with special compiler provisions.

The maintainability of the translated library is one of our primary concerns and it can be best achieved by making the translated code depend, as much as possible, on standard Common Lisp constructs instead of emulated Java constructs. As another example of the difference between using an emulation layer or directly targeting Common Lisp, consider the following actual fragment of a Joda Time test:

```java
if (t != 100 && t != 200 & t != 300) {
   ...
}
```

When translated by Jnil, we get:

```lisp
(setq t 50)
(when (and (/= t 100) (logand (/= t 200) (/= t 300)))
   ...)
```

Although the code worked in Java, it no longer works in Common Lisp. The culprit is the translation of the bitwise **&** operator to the **logand** function. When applied to boolean operands, Java’s **&** works just like the logical operator **&&** but that doesn’t happen with Common Lisp’s **logand**. Most probably, Joda Time’s author did not intend to use the bitwise **&** but the problem went unnoticed. Unfortunately (or not), at the time of this particular translation, Jnil was not sophisticated enough to distinguish the different uses of the operator and the Java “bug” was detected. However, we had then to face the dilemma of leaving Jnil as it was so that other similar “bugs” could be detected, or update it to properly preserve Java semantics. In this particular case, we opted for preserving the semantics because it doesn’t require the use of emulated operators but, in general, the best decision is not obvious: a translation that completely preserves the semantics of the source language might generate semantically correct but pragmatically incorrect code.

7.8 Inner and Anonymous Classes
Inner and anonymous classes are highly problematic for Jnil to translate into Common Lisp. They are fully supported in Linj and Jnil round-tripping capabilities can reconstruct them from the corresponding Java class forms. Unfortunately, when the translation target is Common Lisp, Jnil doesn’t have any similar construct to use and it degrades to use the same conversions that were valid for Linj, thus generating incorrect Common Lisp code.

The obvious solution to this problem is to further develop Jnil so that it generates correct Common Lisp code even in the case of inner and anonymous classes, but this is a task that is far from trivial. Fortunately, there’s a much simpler solution: we can take advantage of the excellent Eclipse refactoring operations to convert the Java source to a simpler form that Jnil can directly translate to Common Lisp. In this pre-processing step we refactor the code so that all anonymous inner classes are promoted to inner classes and all inner classes are promoted to outer classes. This approach has the advantage of relieving Jnil from implementing the complex verifications and transformations that are required in order to preserve the semantics of the Java program before and after the refactoring step. We are currently incorporating this pre-processing phase into Eclipse.

7.9 The Java Standard Library
Any non-trivial Java program depends on one or more of the classes that are available in the standard Java language APIs. These APIs include classes for collections, enumerations, mathematical operations, regular expressions, parsing, input/output, etc. For some of these APIs it is possible to find a Common Lisp counterpart that does essentially the same thing. For example, Java regular expressions can easily be replaced using the excellent `cl-ppcre` library. However, many other Java APIs do not have any correspondence in Common Lisp.
When we started translating Joda Time we noticed that several parts of Joda Time were dependent on the Java Collections Framework. At the beginning, we manually changed the generated code in order to use “proper” Common Lisp replacements such as lists, arrays, and hash tables. It soon became obvious that the differences between the Common Lisp and Java data types would require extensive changes with a very high probability of causing numerous bugs. Moreover, certain types such as hash tables, were more flexible in Java than their Common Lisp counterparts, forcing us to also extend or redefine them in order to support the same functionality.

Another fundamental difference occurred in the processing of collection elements. While Common Lisp has traditionally favored internal iterators (mapcar and friends) or explicit iterations using macros (such as loop and dolist), Java prefers external iterators (as represented by the Iterator interface) that are usually harder to implement but more flexible to use.

We found this “impedance mismatch” very difficult to surmount and, after much effort, we gave up and concluded that standard Common Lisp is also missing a library for generic collections and external iterators. Fortunately, in this case, we were able to find Common Lisp libraries (such as cl-containers or CL Enumerations) that implemented part of what we needed. Unfortunately, in both cases, the implementations are sufficiently different from what is provided in the Java standard APIs and they would still require extensive modifications, either to the libraries, or to the translated Joda Time.

Given the potential for bugs, we decide to follow a different route: we used Jnil again to translate some of the Java standard APIs so that we would be able to test Joda Time without being forced to do extensive changes to the generated code. This allowed us to quickly get an initial working version of the library. Using this strategy it is then possible to do any refactorings we want (including the hypothetical removal of the dependencies from the Java standard APIs) after being sure that the library is correct.

One might think that this approach will cause a cascade of translations but, in fact, it does not. The fundamental Java APIs are generally implemented in terms of Java primitive types so, for Joda Time, no further uses of Jnil are required. However, we do expect that, as more and more libraries are translated, the bigger will be the set of Common Lisp counterparts to the Java fundamental APIs. In the limit, the entire J2SE might end up being translated to Common Lisp but we don’t think of that as a terrible effect. Quite the contrary.

### 7.10 Documentation

In section 2, we criticized the state of Common Lisp libraries and we emphasized the lack of good documentation as one of its serious problems. Obviously, the mere translation of Java libraries to Common Lisp can alleviate the problem of the lack of libraries but it does nothing for the documentation problem.

For our purposes, there are two aspects of the documentation problem that we want to address:

**Internal documentation** It is important that, as much as possible, all the relevant functions and methods contain documentation. Without it, the generated code would be in an even worse shape than most other Common Lisp libraries.

**External documentation** For the users of the library, it is good to have internal documentation but it is even better to have tutorials and user manuals that explain how to use the library, possibly including examples that can be copied&pasted to quickly experiment.

Regarding the internal documentation, at the moment Jnil simply transports to the generated library all the Javadoc documentation that it finds in the Java source code, without doing any processing of the Javadoc tags. This means that some parts of the documentation might not be as relevant to the generated Common Lisp code as they were for the Java code. The list of thrown exceptions is one example.

Another problem is that the names mentioned in the documentation might not refer to the actual names that occur in the Common Lisp code and this should be manually corrected.

Regarding the translation of the Javadoc, it is not yet clear to us what is the best format for the Common Lisp documentation. Common Lisp documentation strings are the traditional Common Lisp practice and the development environments might be able to take advantage of them but, on the other hand, they tend to clutter the code and were not designed to take advantage of the navigational possibilities that are provided by Javadoc. At the moment, we prefer to include the Javadoc documentation (with minimal adaptation) inside ## |# comments. The following example shows the look&feel of the generated code, prior to any manual refactoring and adaptation of the documentation:

```lisp
(defun to-date-time-local-time (this local-date) time) |#
  @param time the time of day to use, null means current time
  @return the DateTime instance
  @throws IllegalArgumentException if the chronology of the time
    fields from the current time.
  @return this date as a datetime with the time as the current time
  @return this date as a datetime with the time as the current time
  @param this this date
  @param time-zone the time zone
  @method to-date-time-local-time-date-time-zone
  |#

| Convert this LocalDate to a full datetime using the default
| time zone setting the date fields from this instance and the
| time fields from the current time.
| This instance is immutable and unaffected by this method call.

@return this date as a datetime with the time as the current time
| (to-date-time-at-current-time ((this local-date))
| (to-date-time-at-current-time-time-time-zone this nil))

| Convert this object to a Datetime using a LocalTime to fill
| in the missing fields and using the default time zone.
| The resulting chronology is determined by the chronology of this
| LocalDate. The chronology of the time must match. If the time
| is null, the current time in the date’s chronology is used.
| This instance is immutable and unaffected by this method call.
| @param time the time of day to use, null means current time
| @return the DateTime instance
| @throws IllegalArgumentException if the chronology of the time
| does not match
| (to-date-time-local-time ((this local-date) time)
| (to-date-time-local-time-time-time-time-zone this nil))
```
Regarding the external documentation, the effort must obviously be entirely manual. Moreover, this effort is directly proportional to the amount of refactoring that was done to the generated code because the refactoring makes the code become less related to the Java documentation. However, if the top-level interface of the translated library was preserved, it is expectable that the effort required to translate the documentation is not excessive. Our experience with the translation of small fragments of the Joda Time tutorial confirm this. A small example of this translation is presented in Figure 1.

It should be mentioned that, in spite of the saved efforts provided by the reuse of existent documentation, careful proofreading and adaptation is still required to translate the documentation to the Common Lisp version.

7.11 Testing the Generated Code

Using Jnil for the translation of Java libraries to Common Lisp is just the first step of what needs to be done in order to provide good, useful libraries for Common Lisp. As was discussed previously, Jnil can only generate correct code when the target language is Ljnj. For Common Lisp, Jnil provides a starting approximation of what the final code might look like but considerable (manual) effort is still needed to remove all the bugs and to adapt the library to the “Common Lisp way.”

This is where the Unit Testing practice so highly promoted in the Java camp comes into place. Due to the fragility of current Jnil translations, we only attempt to translate Java libraries that include extensive unit testing. In this regard, Joda Time is, once again, a very good example to follow. As was discussed previously, Jnil can only generate correct code when the top-level interface of the translated library was preserved because polymorphism can’t always be used, forcing the programmer to resort to code duplication.

To understand the issue we will use the printing capability of Joda Time as an example. Joda Time provides two different ways to generate a textual representation of times:

One might question the validity of testing automatically translated software using automatically translated tests. Fortunately, it is usually the case that unit tests are so simple that no bugs are introduced in their translation. In fact, most tests are nothing more than class instantiations that serve to verify that no bugs are introduced in their translation. In this example, the resulting code was also refactored to confirm this. A small example of this translation is

```lisp
(let* ((chrono (g-j-chronology-get-instance-utc))
        (f (with-locale
             (with-chronology
              (date-time-format-for-pattern 
                "YYYY-MM GG")
            chrono
            +locale=uk+))
         (dt (make-instance 'date-time 
              :year 2005 :month 10 :day 1 
              :hour 0 :minute 0 :second 0 
              :millis 0 :chronology chrono)))
  (assert-equals dt (parse-date-time f "2005-10 AD"))
  (assert-equals dt (parse-date-time f "2005-10 CE")))
```

The use of Joda Time unit tests were fundamental to quickly detect and correct numerous bugs that resulted from Jnil translation. At this moment, the translation passes 2046 of the 2084 translated tests, with all the failed tests identified with a planned solution.

We are sufficiently confident with the results achieved so far that we are already using the translated Joda Time library in two different Common Lisp projects. However, there are still several parts of the code that require a manual refactoring in order to improve its habitability.

7.12 Improving the Generated Code

Common Lisp is a dynamically typed language. Java, in spite of its many similarities with Common Lisp, avoids full dynamic typing, preferring static type checking combined with polymorphism. We claim that this isn’t a proper replacement because polymorphism can’t always be used, forcing the programmer to resort to code duplication.

To understand the issue we will use the printing capabili-

ties of Joda Time as an example. Joda Time provides two different ways to generate a textual representation of times:

- Using a `StringBuffer` that is passed as argument. The textual representation of the time will be appended to this buffer.
- Using a `Writer` that is passed as argument. The textual representation of the time will be printed to this writer.

Both these ways are provided by any of the many formatter classes such as `DateTimeFormatter`. Here is one example of the two different methods defined in `DateTimeFormatter`:

```java
public void printTo(StringBuffer buf, long instant) {
    checkPrinter();
    printTo(buf, instant, null);
}
```

As can be seen in the above definitions, the body of the methods is completely identical modulo a change in a parameter name and a necessary (checked) exception declaration. However, both methods must exist because, class-wise, use keyword parameters in the construction of `date-times`.
Changing TimeZone

**DateTime** comes with support for a couple of common timezone calculations. For instance, if you want to get the local time in London at this very moment, you would do the following:

```java
// get current moment in default time zone
DateTime dt = new DateTime();
// translate to London local time
DateTime dtLondon = dt.withZone(DateTimeZone.forID("Europe/London"));
```

Where **DateTimeZone** for ID "Europe/London" returns the timezone value for London. The resulting value *dtLondon* has the same absolute millennium time, but a different set of field values.

There is also support for the reverse operation, i.e. to get the datetime (absolute millisecond) corresponding to the moment when London has the same local time as exists in the default timezone now. This is done as follows:

```java
// get current moment in default time zone
DateTime dt = new DateTime();
// find the moment when London will have / had the same time
dtLondonSameTime = dt.withZoneRetainFields(DateTimeZone.forID("Europe/London"));
```

A set of all timezone ID strings (such as "Europe/London") may be obtained by calling **DateTimeZone**.getAvailableIDs().

**StringsBuffers and Writers** belong to different branches of the hierarchy tree, making it impossible to take advantage of Java’s polymorphism to avoid defining two methods.

When we think about the translation of the above fragment to Common Lisp we immediately notice that the two different methods do exactly the same thing and, if it were not for the Jnil strategy of combining the methods name with the types of the parameters, we would get two absolutely identical methods. When we delve into the callees of each method, the situation becomes worse. Before actually printing anything, Joda Time must compute the time zone, the time zone offset, the time adjustments required for that time zone, the locale, etc. All these computations are decomposed into a cascade of method calls that finally, at the leaves of the call tree, call the necessary **StringBuffer** or **Writer** operations to actually print something. But since **StringBuffers** and **Writers** neither have a common superclass nor implement a common interface, Joda Time authors had to copy&paste the complete call tree, only changing the type of the output destination.

To make things worse, the previous methods were only defined for the simplified representation of times: the number of milliseconds from some epoch (a *long*). But the library also provides several, more expressive, Date&Time classes. Unfortunately, in Java, the primitive type *long* can’t be mixed with the other reference types used for Date&Time representation, so overloading must be used instead, causing another duplication of code. This phenomena can easily be seen in the fundamental interface that all (10) Date&Time printers must implement in Joda Time: it requires four different methods, one for each combination of “output device” and time representation. In most cases, the implementation of these different methods is absolutely identical but, as before, when the implementation is not trivial and must be decomposed into other method calls, the duplication continues. At some point, the combinatorial explosion becomes unmanageable and simplifying measures must be taken. This is obvious in Joda Time where the printing protocol at some point drops the “output device” distinction and diverts to a different protocol that uses overloading only to distinguish different time representations.

Other effects come into play in this duplicated code phenomena. As was already mentioned, Java’s lack of optional and keyword parameters also contribute to increase overloading and Joda Time printing also suffers from this. As an extreme example of the situation, in one of the abstract classes that participate in the printing process there are ten very similar methods that any concrete subclass must implement. They all do essentially the same thing but differ in the defaulted arguments and in the types of entities they have to pass on to other methods.

This is a situation that can easily be handled by Common Lisp dynamic typing capabilities, ordinary lambda lists, and multiple dispatch methods. All that is required to compute the necessary printing information can be done in one set of methods and can then call a generic function with multiple specializations for the different combinations of output device and date representation. Also, defaults can be provided with keyword parameters and/or dynamically scoped variables.

It is obviously beyond Jnil capabilities to infer the most adequate Common Lisp code from the large code duplication that is present in the Java sources. Our current strategy is to let Jnil do the simplest translation it can and then we search for redundant code using our redundancy detector tool **R**D²[13]. **R**D² compares the syntax trees of Common Lisp code fragments, searching for similarities and differences that confirm or refute the belief that there is redundancy. When it gathers sufficient evidence, it presents the user with the similar code fragments, thus pinpointing the places where code improvement should be done. It is now the user responsibility to find an appropriate refactorizing of the code. Regarding Joda Time, we already did some significant refactorizing but there are still many places that deserve to be improved.

---

**Figure 1: JodaTime external documentation for Java and for Common Lisp.**
7.13 Performance Issues

It is important to remember that Jnil translates from statically typed Java to dynamically typed Common Lisp, dropping all type information in the process. It would not be difficult to include some type information in the Common Lisp program but, for pragmatical reasons, we prefer not to: type declarations have a tendency to make the Common Lisp code harder to read. However, this decision has the obvious consequence that, in general, the Common Lisp program will not run as fast as the original Java program. Another performance penalty occurs when we transform overloaded Java methods into normal Common Lisp methods.

Usually, we don’t mind trading performance for the flexibility of using Common Lisp instead of Java but, in some cases, the price is too high. Parsing dates and times is one such case because the parsing is driven by a tree of specialized parsers, each implementing a different parsing method (that expect literal strings, characters, padded numbers, etc).

Fortunately, we can use the well-known Common Lisp technique of compile-time generation of code (that is much harder to use in Java) to optimize the common case where the parsing pattern is known in advance. To this end, we complemented the translated program with a protocol for generation of code from the tree of specialized parsers. The protocol is specialized for each type of parser so that it returns the code fragment that implements the parser behavior but where all compile-time information that was available was used. This is then used by a macro that, given a format string, asks the translated parsing machinery to build the appropriate tree of parsers, invokes the protocol on that tree to generate the parsing code and uses that code as its own expansion.

Using this technique, it was possible to speed-up the parsing performance without making any changes to the already existent parsers. In practice, it was like developing a parser compiler capable of generating faster code from the parsers that were designed for Java.

8. CONCLUSION

Modern software requirements have changed the programming language landscape. Current mainstream languages are successful not because of their intrinsic qualities but, instead, because of the availability and broadness of their libraries.

In order to survive, the Common Lisp language must adapt to this different environment by providing libraries that are competitive with those available in other languages. However, due to the limited size of its user base, we do not expect that new libraries can be developed in a short time frame and the only solution is to reuse libraries that were developed for other languages. This solution has been explored several times in the past, either in the form of foreign function interfaces or as language translations.

In this article, we claim that the best option for providing Common Lisp with good and maintainable libraries is to automatically translate from the best ones available in other modern object-oriented languages such as Java. Usually, maintainability entails that the code must be written according to the language pragmatics, i.e., the common coding practices of the language, but this is a very difficult goal to achieve by purely automatic means. As a result, it is expected that any translated library should be further improved by a manual refactoring process.

This paper documents the translation of Joda Time to Common Lisp using Jnil. Joda Time is a very large Date&Time library for Java. Jnil is a translator constructed for round-tripping from Linj to Java but that is also capable of generating Common Lisp.

In spite of its limitations, Jnil could be effectively used to generate a good starting point for a pure Common Lisp Date&Time library. This starting point was then repeatedly refactored and modified in order to improve it.

Several lessons were learned during this process, both regarding the translation of specific Java language features and the best strategy for an effective translation. We plan to improve Jnil to address these lessons.

9. REFERENCES


