Babel - A Translator from Smalltalk into CLOS

Ivan Moore[†], Mario Wolczko[‡], Trevor Hopkins[†]

[†]Department of Computer Science,
University of Manchester, Oxford Road,
Manchester M13 9PL, England

[‡]Sun Microsystems Laboratories Inc,
2550 Garcia Avenue, M/S MTV 29-116,
Mountain View, CA 94043, U.S.A.

Abstract

This paper[1] introduces Babel, a prototype translator and associated tools for developing stand-alone applications by translating Smalltalk into portable CLOS (Common Lisp Object System). Previous translators have required varying degrees of programmer intervention and imposed restrictions on the Smalltalk which can be automatically translated. Babel requires a minimum of programmer intervention and imposes minimal restrictions on the Smalltalk which can be translated automatically. Results of translating two non-trivial test programs are presented; the tools provided by Babel, and the performance of the translated code are assessed.

1 Introduction

For delivering applications, Smalltalk has some disadvantages compared to more conventional languages. Many different approaches are used for delivering applications developed in Smalltalk. An application can be delivered as source code, as a stripped image with a virtual machine or as a stand-alone executable. Delivering source code does not allow for confidentiality. Delivering a stripped image often requires the user to purchase, in addition to the application, a licence for the Smalltalk system on which it runs. For these reasons, we believe that in most cases a single stand-alone executable is the most desirable way to deliver an application program.

There are two approaches to producing an executable for a Smalltalk application program: either the application can be compiled directly, or the application can be translated into source code for another language and then compiled. The latter approach was suggested by Brad Cox, who developed Producer[Cox 87], a translator from Smalltalk into Objective C, to demonstrate the principle. Translators have been developed by others[Nash 91][TNI 90], but all of them have required varying degrees of programmer intervention and imposed restrictions on the Smalltalk which can be automatically translated. This paper introduces Babel[2][Moore 93], a prototype translator and associated tools for translating Smalltalk[3][Goldberg 89] into CLOS[Keene 89][Kiczales 91] (Common Lisp [Steele 90] Object System) with a minimum of programmer intervention and minimal restrictions on the Smalltalk which can be translated automatically. Translating from Smalltalk into another language requires the translation of both the application code, and at least some of the Smalltalk environment because, typically, classes and objects in the environment are used by application programs. The translator has been given the name "Babel", after the fish[Adams 79] rather than the tower[Bible].

[1]TOOLS USA 1994, published by Prentice-Hall

[2]Babel will be made available by anonymous ftp in the Manchester goodies library in the near future.

[3]Objectworks\Smalltalk version 4.1[ParcPlace 92] was used to develop Babel.

1.1 The Babel System

Babel is a collection of related tools, including a Smalltalk to CLOS translator. Figure 1 shows how these tools are related. The translated source code produced by Babel can be compiled using a suitable CLOS compiler.

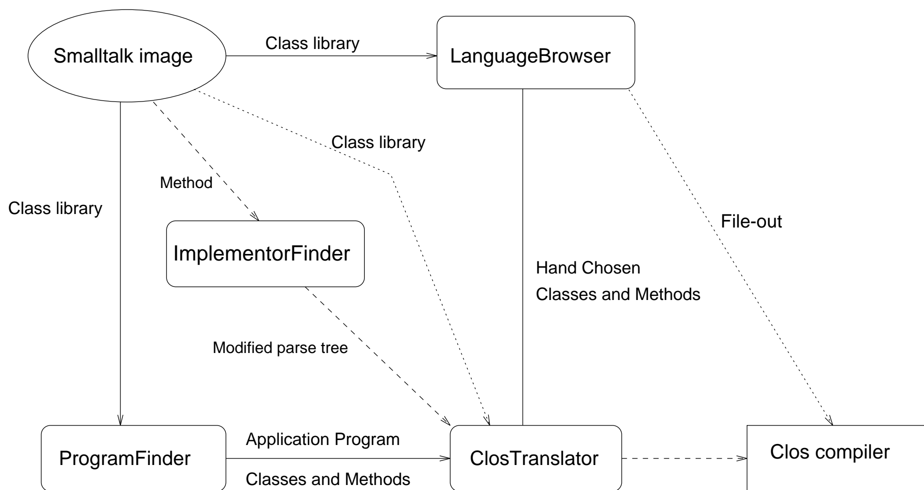

Figure 1: Using the Translator

These tools include LanguageBrowsers[4], modified from the Smalltalk System Browser; they provide facilities to browse, edit and file-out translated code for *any* suitable translator. Smalltalk to CLOS translation is performed by a ClosTranslator. An ImplementorFinder makes a naive attempt to work out which methods implement the messages sent by a method. Information from an ImplementorFinder is used by a ClosTranslator to improve the performance of the translated code. A ProgramFinder can be used to work out which classes and methods are required for an application program. This can be used to specify the translation of those methods and classes.

2 The Translator class - ClosTranslator

The Smalltalk compiler can be called from within Smalltalk, and is represented by the class Smalltalk-Compiler. ClosTranslator walks the explicit parse tree for the method to be translated. Methods have been added to perform the translation of each type of node, emitting code directly onto a stream without any post generation optimisation.

A message send in Smalltalk is translated directly into a CLOS message send, however the names are modified slightly because CLOS is case insensitive and certain characters have a special meaning. The ordering is also changed, as in Smalltalk the receiver comes first, followed by the message and arguments, whereas in CLOS the message comes first followed by the receiver as the first argument and subsequently the other arguments. A method is simply a sequence of statements, such as message sends, assignments and a return statement. In Smalltalk, the selection of a method is done according to the class of the receiver. In CLOS, methods can be selected according to the class of one or more of their arguments. To translate a Smalltalk method, the first argument of the CLOS method is the receiver, represented as "`self`"[5], and the method is selected according to the class of this first argument.

[4] Note that sans serif font is used for Smalltalk code fragments.
[5] Note that `typewriter` font is used for CLOS source code fragments.

For example, in Dictionary the method:

```
includesKey: key
        | index |
        index := self findKeyOrNil: key.
        ↑(self basicAt: index) notNil
```

becomes:

```
(defun _dictionary_.includes_key%_
       (self key_)
 (prog ( index_) sequence*
  (setf index_
   (find_key_or_nil%_ self key_))
  (return-from
    _dictionary_.includes_key%_
    (not_nil_ (basic_at%_ self index_)))))

(defmethod includes_key%_
           ((self _dictionary_) key_)
 (_dictionary_.includes_key%_ self key_))
```

ClosTranslator provides an option either to translate a method as a function and a method, as shown in the example above, or simply as a method. The advantages of the approach shown above are explained in section 2.2.

2.1 Translation to the level of primitives

In Smalltalk, some methods, mostly in the standard class library, are written in terms of *primitives*. There are two approaches to translating the classes which include such methods; either *replacement classes* can be provided, or the primitives can be hand-translated and the methods calling them automatically translated. To provide replacement classes, either the methods can be hand-translated[Nash 91][TNI 90], or if suitable classes exist in the target language then these can be used[Cox 87]. Using replacement classes allows a degree of flexibility in their implementation, as only the external interface and externally-visible behaviour needs to stay the same. Special features of the target language can be used to improve their performance. Furthermore, this approach avoids having to translate all of the primitives directly, some of which are difficult to translate, for example those involving the user interface.

The approach taken by Babel is to translate to the level of primitives, so that the standard class library is translated in the same way as applications classes and methods. This approach was suggested by Vokach-Brodsky[Vokach-Brodsky 90] and used by Cope[Cope 94]. Translating to the level of primitives avoids having to hand-code replacement classes. We believe that translating the primitives requires less effort than hand-coding replacement classes. The translator can be used to automatically and accurately translate those standard classes which are required. Furthermore, translating to the level of primitives enables different versions of Smalltalk, which have different standard classes, but similar primitives, to be translated with little additional effort. There is no reliance on a similar standard class library in the target language.

2.2 How to represent literals

The literals in Smalltalk are: special instances such as true and false, numbers, Characters, Arrays, Strings, Symbols and blocks. The way literals are represented in the translation has important consequences for the performance and the accuracy of the translation. Translated literals can be represented either as objects which are instances of the appropriate translated Smalltalk class, or as the closest

equivalent CLOS representation. If translated literals are represented as instances of the translated Smalltalk class, then messages understood by instances of the Smalltalk class will also be understood by instances of the translated class. The disadvantage of this approach is that it is less efficient, as wrapper objects have to be created to contain the underlying CLOS representations used in the translation of the primitives.

It is more efficient to use the closest equivalent CLOS representation directly. However, this has the disadvantage that the class of the equivalent CLOS literal will inevitably be different from the class of the Smalltalk literal it translates. Therefore, messages understood by the Smalltalk literal will not be understood by the CLOS equivalent. This problem is easily solved by specialising methods for the Smalltalk literals on the class of the CLOS objects used to represent them. However, methods inherited by objects of the Smalltalk class will not be inherited by the CLOS objects used to represent them, so these methods are copied and specialised on the relevant CLOS classes. The ability to make subclasses of CLOS standard classes is implementation dependent, which reduces the number of alternative solutions. Translating a method as a function and a method, as shown earlier, has the advantage that these copied methods are trivial and do not require a significant amount of copying.

The representations of three classes of literals are discussed; Numbers, Booleans and blocks.

2.2.1 Numbers

The class of the Smalltalk literal "1" is SmallInteger, which is translated into a CLOS class called `_small_integer_`. The class of the CLOS literal "`1`" is `Integer`. It is *essential* to realise that the Smalltalk class Integer and its subclasses, such as SmallInteger, are different to the CLOS class `Integer`.

In order to make CLOS `Integer`s understand the same messages as Smalltalk Integers, the class specialisers for the translated Integer methods are translated as being specialised on the CLOS class `Integer`. Special translations for the method class are used for CLOS numbers. As methods are not inherited by CLOS `Integer`s from the superclasses of the translation of the Smalltalk class Integer, for example from Object, a copy is made of all methods defined for superclasses of Integer, and specialised on the CLOS class `Number`. Only one copy of these methods has to be made because all the number types in CLOS inherit from `Number`.

In the prototype implementation, numbers are represented as their CLOS equivalents, rather than as instances of their translated Smalltalk class, because the performance of arithmetic has a significant impact on the overall performance of the translated code. Note that the potential problem of arithmetic overflows is handled transparently in CLOS similarly to Smalltalk.

2.2.2 Booleans

ClosTranslator provides two different ways of translating Boolean literals. One way is to translate the literals true and false as CLOS globals called `-true-` and `-false-`, which are instances of the translated Smalltalk classes, `_true_` and `_false_`. The alternative approach is to use the closest CLOS equivalents, which are `t` for true and `nil` for false. In CLOS, anything other than `nil` inherits from `t`, so anything other than `nil` can be used. Therefore, true can still be represented as `-true-`. The class of `nil` is `null`, so methods for False have to be translated as specialised on `null`.

The disadvantage of the second approach is that the translation is no longer totally accurate, because things other than true are "true". This is less of a problem than it appears at first, because true and false are typically sent only a small number of messages which are applicable only to booleans. For example, ifTrue: in Smalltalk can only have a boolean receiver, and if it is not then an error message is sent by the Smalltalk virtual machine. This should not occur in a delivered application; nevertheless, even this situation can be translated accurately using the first approach.

One advantage of the second approach is improved efficiency. Furthermore, the special case translation of control structures, as discussed in the following section, is more elegant. Also, in some cases

comparison messages can be modified to use the equivalent CLOS function, rather than a message send, which can also improve performance.

2.2.3 Literal Blocks

This section describes the translation of literal blocks in general cases, as well as in special cases. Blocks do not have a direct equivalent in most languages. Fortunately, in Lisp there is a similar idea of an anonymous function, called a `lambda`. To represent a block in CLOS, we make an instance of `_block_closure_` with a slot `*block*`, which is initialised to the `lambda` which represents the translation of the sequence of statements in the block. When a translated block is sent the value message, it simply calls the `lambda` in its `*block*` slot using the Lisp function `funcall`. The translation of the sequence of statements inside a block is exactly the same as for a sequence of statements of a method. In order to translate blocks with non-local returns, the CLOS function `return-from` is used, which causes the method to return the value given, without continuing further. For normal blocks, that is blocks which do not have a non-local return, a simple `return` is used, so that the value is returned as the result of evaluating the block, and the rest of the method continues.

Babel copes with all types of blocks in all types of circumstances. For example, blocks with arguments, with local variables, being passed as parameters, and so on. Many other translators cope only with blocks in special cases, such as ifTrue: with a literal block. Although authors of such systems point out that this is the most common use of blocks, we believe that blocks are such an important and useful construct in Smalltalk that they should be translated without such restrictions. Vokach-Brodsky[Vokach-Brodsky 90] and Cope[Cope 94] showed how general blocks can be translated into other languages.

As well as translating all types of blocks in general, special cases, such as ifTrue: are translated differently by Babel, for performance reasons only. These special cases are translated into the equivalent CLOS control structure rather than translated literally into a message send with block arguments. The special cases are only used when the receiver and/or arguments (as appropriate) are literal blocks. Note that the messages which have been special-cased cannot be redefined and called using literal blocks as arguments. This limitation is the same in Smalltalk, i.e. it is not different in the translation. The messages which are special-cased are the same as those which are special-cased in Smalltalk.
As an example, consider the following code[6]:

```
fibonacci
        self <= 1 ifTrue: [↑1]
                  ifFalse: [↑(self - 1) fibonacci +
                           (self - 2) fibonacci + 1]
```

Using special cases and translating false as `nil`:

```
(defmethod fibonacci_ ((self integer))
 (prog () sequence*
 ;;; special case for
     if_true%if_false%_ ;;;
  (if (<=_ self 1)
   (return-from fibonacci_ 1)
   (return-from fibonacci_
    (+_ (+_
     (fibonacci_ (-_ self 1))
     (fibonacci_ (-_ self 2))
        ) 1)
  ))
  (return-from fibonacci_ self)))
```

[6] This is not strictly fibonacci, but a similar function known as “n-fib”.

The version with special cases is over 100 times faster than the one without special cases. The difference in performance in this case is more than for a typical Smalltalk application, but is still an indication of how important this is.

2.2.4 Literals created once

In order to improve the performance of the translated code, rather than creating an instance of a literal each time the method that uses it is called, appropriate literals are created just once in the translated code and stored in a CLOS `hash-table`. The time overhead for finding something in a `hash-table` is smaller than for instance creation. Babel takes into account that some literals which have the same value are in fact occurrences of the same object, i.e. $a == $a is true, whereas for Arrays, Strings and pure blocks, they are occurrences of two different objects, i.e. 'hello' == 'hello' is false.

3 ImplementorFinder

ImplementorFinder is used by ClosTranslator to improve the performance of the translated code. ImplementorFinder tries to work out which methods will actually be called in response to a message send. Translating a method as a function and a method allows the function implementing the method to be called directly in cases where ImplementorFinder can determine which function to call. This improves the performance of the translated code by reducing the amount of dynamic binding. ImplementorFinder produces a modified parse tree for a method, indicating which class implements each message where this can be determined statically. Unlike more sophisticated and successful type inferencing approaches[Palsberg 91], ImplementorFinder uses only local code analysis. ClosTranslator uses an ImplementorFinder to improve the performance, and does not depend on it for the accuracy of its translation.

Note that the class of the receiver does not always have to be determined to work out which method will be called when a message is sent. For example, in the statement anyObject printString, we can determine that the method called will be the implementation of printString in the class Object if it is the only implementation of printString. The current version of ImplementorFinder does not make allowances for cases where doesNotUnderstand is overridden, or for classes which do not inherit from Object, but this could easily be included as an option. ImplementorFinder works one method at a time using very simple heuristics. For example, if anObject := AClass new then the class of anObject is taken to be AClass, unless otherwise indicated in a table for those few cases where this is not true, for example, aString := String new: 10 results in aString being a ByteString. Other situations handled include sends to super, and sends to self in methods for classes which have no subclasses. Manual type declarations can be included in Smalltalk source code, which have no effect on the Smalltalk, but are used by ImplementorFinder. A system like Strongtalk[Bracha 93] would make the implementation of ImplementorFinder trivial.
In the Smalltalk method:

```
implementorFinderExample
        | x |
        x := Dictionary new.
        x add: 1->2.
        Transcript show: (x at: 1) printString.
```

ImplementorFinder can determine the implementation of all of the messages.

4 LanguageBrowser

The Smalltalk System Browser is very familiar to Smalltalk programmers, as it is the standard way of entering, retrieving and editing Smalltalk source code. A subclass of Browser has been created, called LanguageBrowser, to integrate a translator with a browser. A LanguageBrowser provides the convenient and familiar interface of a Smalltalk System Browser so that translated source code can be treated in a way consistent with Smalltalk source code.

A LanguageBrowser has a button which is pressed to switch between Smalltalk and another language. For example, a LanguageBrowser in the Smalltalk mode is shown in figure 2, the same LanguageBrowser when the CLOS button is pressed is shown in figure 3.

A LanguageBrowser is independent of a ClosTranslator. It can be used with other translators provided they respond to certain messages. The interface is provided by a class called AbstractTranslator, which is subclassed in order to create a translator suitable for integration with a LanguageBrowser.

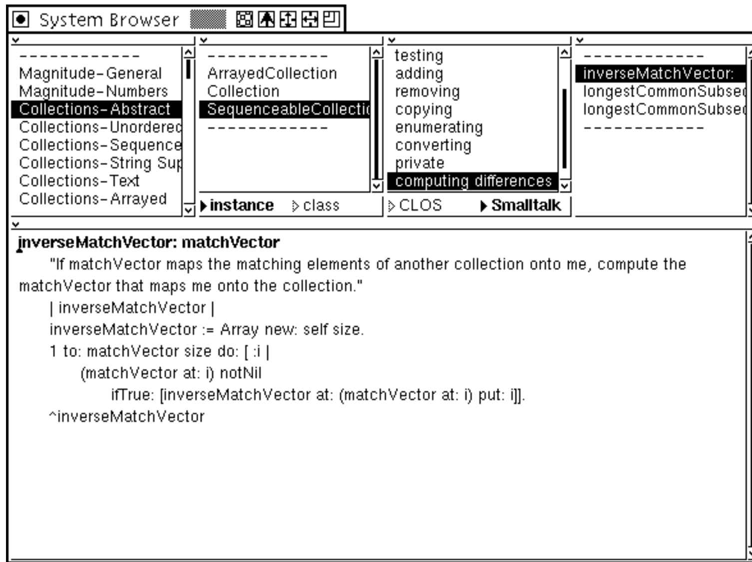

Figure 2: LanguageBrowser - Smalltalk mode

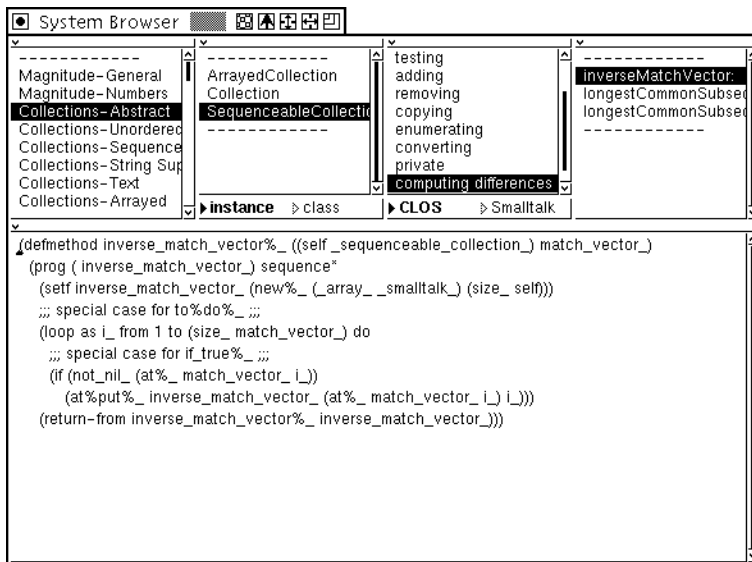

Figure 3: LanguageBrowser - CLOS mode

A LanguageBrowser can be used to file-out methods, protocols, classes and categories. A Language-Browser can be used to edit and accept modifications to the translated code. This can be important when optimising a translated application; for example, special features of the target language can be used. Note that AbstractTranslator handles the storage of edited methods and class definitions, and retrieves them, rather than re-translating, when they are requested. In the case of the other Smalltalk translators, editing translated code is done by editing the output files produced by the translator, outside of the translation system, or with only a small amount of integration. A LanguageBrowser is particularly convenient for hand translating the code because the automatically generated translation can be edited, rather than writing the translation from scratch.

Options available for a translator can be changed by opening a DialogView with buttons to press to change the options. A simple mechanism for this is provided by AbstractTranslator and can be redefined by subclasses. ClosTranslator provides many different options which can be changed in this way, for example the option to chose how to represent booleans (see section 2.2.2).

5 ProgramFinder

To translate an application program from Smalltalk into another language, the methods required for the application program need to be translated. In order to make the translated code as compact as possible, *only* those methods which are required should be translated. The methods which are required are those which *could* be invoked during the running of the application program. To determine the minimum collection of methods which are required, and none others is, in general, impossible due to the reflective nature of Smalltalk.

ProgramFinder provides methods that are useful for determining everything that needs to be translated for an application. ProgramFinder is cautious, in that it produces a collection which, in addition to those methods which are required, includes many methods which are not required. This is inevitable because of the dynamic binding of Smalltalk.

ProgramFinder finds the possible message sends recursively; this can take a long time, and may produce a surprisingly large list of messages.

To give an indication of the performance of the ProgramFinder, the messages for a test program, which simply created a Set, added a few Integers to it and then iterated over the Set, were worked out by hand and filed-out as 70k of source code. The ProgramFinder produced just over twice this amount of filed-out code. Translating the whole of each class that the ProgramFinder considered, produced a file-out about twice as large again.

In order to check whether all the necessary classes have been specified, another method is provided to check that every message in a list of messages is implemented by at least one of those classes. It prints a message to Transcript showing any messages which are not implemented, and a list of possible implementors.

The output of the ProgramFinder is a list of messages which need translating. A method is provided by ClosTranslator which uses this, and a list of classes, to specify the translation of a complete application. In practice, this is currently the best way of translating a complete application, rather than using the file-out facility of the LanguageBrowser.

6 Testing

The translator has been tested on two programs; one which implements an algorithm to compute the longest common subsequence of two sequences[7] [Hunt 77] (similar to that used by the Unix “diff” command but more general as the sequences can be of any type of object), and another which imple-

[7] Available by anonymous ftp from the Manchester goodies library.

ments the "Richards Benchmark"[8]. Both of these programs are atypical Smalltalk, in that they do not require the user interface. The translated source code of both programs was compiled and executed using two different compilers, which demonstrates its portability.

6.1 The "diff" program

This program uses many of the Smalltalk Collection classes, as well as arithmetic, Strings, Characters and many other Smalltalk classes. Although it is a small program, it requires a very large amount of code to be translated because it uses (directly and indirectly) many different classes and results in the calling of many methods.

Using the ProgramFinder to work out what needs to be translated results in about 330k of translated source code.

The performance of Babel in terms of speed of translation is good; it takes between 50 seconds to 6 minutes to translate this test code on a Sun SPARC ELC, depending on the ClosTranslator options used.

Using the fastest options of ClosTranslator, the translated code of this program is 10 times slower than the original Smalltalk. It is important to note that this translation required no intervention by the programmer, no changes to the original code, and no additional hand translation. By including a few type declarations and hand writing a small number of functions after having profiled the code, the performance was improved to within 6.5 times slower than the original Smalltalk. This is disappointing, but we believe is due to the nature of this particular program. In particular, this program uses many collection classes, which translated automatically are relatively slow. An improvement could be made by using hand written replacement classes exploiting the features of CLOS, for example, using `hash-tables` to implement Dictionaries.

6.2 The "Richards Benchmark"

This program is even less typical of Smalltalk than the "diff" program, in that it is almost self contained; calling only 7 non arithmetic messages defined outside the benchmark. The code produced by Babel is 4.5 times slower than the original Smalltalk. Hand editing this code increased its performance to 2 times slower than the original Smalltalk. A totally hand written version of this program[9] was twice as fast as the Smalltalk version. This hand written version was significantly different to a direct translation of the Smalltalk code, and so should not be taken as an accurate guide of the relative speed of CLOS compared to Smalltalk, but does indicate that the performance of code produced by Babel can be expected to be improved. Note that only a small amount of hand translation was required; this was for Transcript and Time.

7 Conclusions

Babel has potentially fewer restrictions and requires less programmer intervention than other Smalltalk translators.

Providing the user with options to choose between either fewer restrictions or improved performance, increases the flexibility of the translator.

Tools such as LanguageBrowser and ProgramFinder are important for improving the use of a translator. In particular, the ability to browse and edit translated code is very useful.

Translating a prototype automatically produces a working system with little additional effort. However, the system produced is still, essentially, a prototype. Improving a system that is already

[8] The "Richards Benchmark" is an operating system simulator. It was originally written in BCPL by Mark Richards. The Smalltalk version translated by Babel was written by L. Peter Deutsch.

[9] Many thanks to Urs Hölzle of Sun Microsystems Laboratories for providing this code.

working has its advantages and disadvantages. The risk is that a working system will be left as it is, with no improvement.

7.1 Future Work

An improved version of ImplementorFinder, possibly working on a complete application[Palsberg 91] rather than a method at a time, could improve both the performance of the translated code by further reducing the amount of dynamic binding, but also could improve a ProgramFinder thus significantly reducing not only the size of a translated application, but also of a "stripped" Smalltalk application.

In order for Babel to become really useful, CLOS versions of the user interface primitives are needed.

Typically, Lisp implementations do not provide a suitable mechanism for producing stand-alone executables. This may appear to be a severe limitation, however, we believe that Babel could be modified relatively easily to translate into Dylan[Apple 92], which is specifically designed to provide facilities for delivering applications.

Acknowledgements

The authors would like to thank Brian Monahan for his help with CLOS, and Jon L. White for his help with the CLOS meta-object protocol. We would also like to thank Jon Taylor and Glenn Maughan for their useful comments on the draft of this paper.

References

[Adams 79] Douglas Adams
The Hitch-Hiker's Guide to the Galaxy (page 50)
Pan Books (1979)

[Apple 92] Dylan Interim Reference Manual
(mosaic) http://legend.gwydion.cs.cmu.edu:8001
Apple Computer Inc, Cupertino, California, USA. (1992)

[Bible] The Bible, Genesis 11:1-8

[Bracha 93] Gilad Bracha, David Griswold
Strongtalk: Typechecking Smalltalk in a Production Environment
OOPSLA (1993) SIGPLAN Notices Volume 28 Number 10

[Cope 94] Neil Cope
A Translator from Smalltalk to Eiffel
MSc thesis, University of Manchester (to appear 1994)

[Cox 87] Brad J. Cox, Kurt Schmucker
Producer - A tool for translating Smalltalk-80 to Objective C
OOPSLA (1987) SIGPLAN Notices Volume 22 Number 12

[Goldberg 89] Adele Goldberg, David Robson
Smalltalk-80 : The Language
Addison-Wesley (1989)

[Hunt 77] J. W. Hunt, T. G. Szymanski
A fast algorithm for computing longest common subsequences
CACM Volume 20 Number 5 pp 350-353 (1977)

[Keene 89] Sonya E. Keene
Object-Oriented Programming in Common Lisp
A Programmer's Guide to CLOS
Addison-Wesley (1989)

[Kiczales 91] Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow
The Art of the Metaobject Protocol
The MIT Press (1991)

[Moore 93] Ivan R. Moore
Constructing stand-alone applications by translating Smalltalk into CLOS
MSc thesis, University of Manchester (1993)

[Nash 91] C. Nash, W. Haebich
An "Accidental" Translator from Smalltalk to ANSI C
OOPS Messenger Volume 2 Number 3 (1991)

[Palsberg 91] Jens Palsberg, Michael I. Schwartzbach
Object-Oriented Type Inference
OOPSLA (1991) SIGPLAN Notices Volume 26 Number 11

[ParcPlace 92] Objectworks\Smalltalk User's Guide (second edition)
ParcPlace Systems, Sunnyvale, California, USA. (1992)

[Steele 90] Guy L. Steele Jr.
Common Lisp - The Language (second edition)
Digital Press (1990)

[TNI 90] OpenTalk Reference Manual B2
Techniques Nouvelles d'Informatique
Plouzain, Brest, France. (1990)

[Vokach-Brodsky 90] Borek Vokach-Brodsky
Smalltalk Application Compilers
MSc thesis, University of Manchester (1990)

[Vokach-Brodsky(2) 90] Borek Vokach-Brodsky, Mario Wolczko
Smalltalk Application Compilers
Proc. TOOLS Pacific (1990)