An Adaptive Package Management System for Scheme

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
http://www.inria.fr/mimosa/Manuel.Serrano

Erick Gallesio
Université de Nice – Inria Sophia Antipolis
930 route des Colles, BP 145, F-06903 Sophia Antipolis, Cedex, France
http://www.essi.fr/~eg

Abstract

This paper presents a package management system for the Scheme programming language. It is inspired by the *Comprehensive Perl Archive Network* (CPAN) and various GNU/Linux distributions. It downloads, installs, and prepares source codes for execution. It manages the dependencies between packages. The main characteristic of this system is its neutrality with respect to the various Scheme implementations. It is neutral with respect to the language extensions that each Scheme implementation proposes *and* with respect to the execution environment of these implementations. This allows the programmer to blend, within the same program, independent components which have been developed and tested within different Scheme implementations. ScmPkg is available at:

`http://hop.inria.fr/hop/scmpkg`

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages; D.2.12 [*Software Engineering*]: Interoperability—Interface definition languages; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Design, Languages

Keywords Functional programming

1. Introduction

Scheme is a functional programming language which was created in the 1970s. Amongst the specific features of Scheme, we can cite its Lisp like syntax, its (hygienic) macro system which provides a powerful self-extension mechanism, its support for continuation management with it `call/cc` operator which allows the expression of complex flow control structures, or its support for proper tail-recursion.

Another characteristic of Scheme is its size: it is incredibly small and compact. The, still official, report [3] describing the language requires no more than 50 pages, and it includes a formal semantics! This is sometimes an advantage, for instance, for teaching. This is also a drawback. The price to pay for this compactness is a severe lack of libraries. Scheme does not provide the minimal set of primitives required for writing any *modern applications*, such as applications involving network communication, graphical interfaces, or multiples threads.

Implementing a naive interpreter or compiler is an easy task since Scheme is small. Hence, it is no surprise that Scheme has so many different implementations. They all have different flavors and tastes. Some promote a read-eval-print loop interaction. Some promote batch compilations. Some emphasize easy access to C or Java. etc. Because the language is unrealistically minimalist, each of its main implementations provides numerous extensions. As we can see, Scheme is not a language, but a family of languages. There are as many dialects of Scheme as implementations of Scheme and it is uncertain to execute a Scheme program within a different environment from the one it has been implemented with. Scheme is yet another tower of Babel!

The Scheme language is minimalist but its main implementations generally are not. In effect, current Scheme implementations offer means to program the *modern applications* depicted above since most of them provide an object system, libraries for network programming, multiple threads, etc. As a consequence, the Scheme community is in fact fragmented in several implementation communities.

Today it is hardly possible to develop, maintain, and release Scheme codes that could directly be used by several implementations although porting such codes from one implementation to another is generally easy since, as we have seen before, implementations tend to offer similar features with slightly different interfaces. To our knowledge three endeavors, including the present study, are currently ongoing to tackle the development of Scheme codes that can be run

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS'07, October 22, 2007, Montréal, Québec, Canada.

Copyright © 2007 ACM 978-1-59593-868-8/07/0010. . . $5.00

on various implementations. They either follow a *centrifugal* approach or *centripetal* approach.

- The *centrifugal* approach consists in *attracting* users by enriching the language with enough extensions in order to bootstrap a *portable* package system. That is, if it is possible to extend the language with the libraries required to implement modern applications then the community could start contributing with new APIs and libraries.
- The *centripetal* approach consists in accepting the differences and idiosyncrasies of each implementation by proposing a framework that lets pristine Scheme codes coming from various implementations be integrated inside a single application or library.

ScmPkg, our proposal, belongs to the centripetal family. It is not a language but a mere package repository, a couple of web services, and a set of conventions and rules. It rests on a pragmatic approach that could be described as a gentle *bazaar* as described in a famous paper by E. Raymond [5]. The general idea is not to try to elaborate a system that solves all problems that could be raised in all situations. By opposition, it is a system that tries to solve the problems that are raised by common practices. The ideas governing the design of ScmPkg are:

- Let users (i.e., Scheme programmers) continue to maintain their private practices. In particular, let users keep writing their code using the language and *its extensions* they like or they are used to.
- Let users keep developing using the environment they like and when the code is ready, let them transform it into a package by automatic tools.
- Make the distance from a package to plain Scheme code so short that it could be *almost* possible for programming environments to use packages as regular source files.

ScmPkg does not even pretend to address all the programmers problems of portability of Scheme code. It only claims to permit one to write *almost portable* packages that can be used by several implementations, not by enforcing rules or by imposing language constructions, but by gently begging for *fair use*. So, ScmPkg goal does not consist in permitting the execution of all Scheme codes that are floating around on particular implementations. Codes which use features bound to one particular implementation are really unportable and ScmPkg is helpless for these codes.

We have evidence that the ScmPkg packaging system can be successful for reusing Scheme codes. Quite easily, we have been able to feed our packaging system with more than a hundred packages within a couple of weeks, by simply *grabbing* code here and there. The system is operational for three Scheme implementations: Bigloo, STklos, and, partially, MzScheme. It accepts source code coming from seven Scheme dialects. Using ScmPkg, we have successfully built applications blending packages implemented in several of these dialects.

Reusing code written for a given Scheme dialect may sometimes lead to poor performances on another implementation. Sometimes, it may use dialect features that are unavailable or behave incorrectly on this implementation. In such cases, it is suitable to *tune* the code for this implementation. ScmPkg permits one to develop, when needed, implementation specific adaptations that will be applied when a package is installed on a given Scheme dialect. Hopefully, not all the packages need to be tuned but this mechanism, which will be lenghtly decribed in Section 3, ensures a correct behaviour and correct performances of ScmPkg packages for each Scheme implementation.

The next sections present the technical aspects of ScmPkg. First, in Section 2, we present the system *per se* and a small application blending two dialects. Then, in Section 3, we present the *Host Adaptation*, that is the machinery that is deployed to let code written for one implementation be used within another one. Section 4 briefly states the material which must be developed to offer ScmPkg on a new implementation. Finally, Section 5 presents the current status of ScmPkg.

2. ScmPkg

ScmPkg is a loose architecture that consists of a web server, a simple Interface Desciption Language (IDL) for describing packages, and ad-hoc package managers that are provided by the Scheme implementations (that we henceforth denote as *Scheme hosts*, *hosts*, or *dialects*) that support ScmPkg.

- The web server implements a graphical user interface for browsing the packages, their documentation, and their source code. It also supports two services that are used for installing the packages. The first one provides the whole list of packages and the second implements downloading of individual packages.
- An IDL describes the material exposed by a package along with some additional meta information. This IDL is described in Section 2.3.
- Each Scheme host that supports ScmPkg must provide a *package manager*. This is a simple program that is in charge of preparing a package for its host. This, in general, involves *downloading*, *preparing* and *installing* the code. Optionally it may also *compile* the code. The package manager is the equivalent to the `apt-get` command of GNU/Linux Debian distributions.

In contrast to other approaches, ScmPkg does not rest on a unified framework for installing packages. This design choice is based on the observation that the process of installing source code is highly dependent on the nature of the host. For instance, installing a package for a batch compiler could mean to download it and its dependencies and to generate a Makefile for building a library out of these

packages. This task can be assumed by an external tool. The same operation for a system based on a read-eval-print loop, could be totally different. According to this setting, a convenient approach could consist in overriding the `require` command (assuming that this form loads a package in the system) for automatically downloading missing packages. In this context, no external tool is required. Since the objective of ScmPkg is to provide a package-management system that is as transparent as possible, we think that letting implementors of Scheme hosts choose their own "right way" for installing packaging is the wisest approach.

After this informal presentation and before digging into the technical aspects of ScmPkg, we present a complete example in the next section.

2.1 An example

```
1: (module gtranslate
2:    (import http-utils html-parse)
3:    (export (trans text from to)))
4: (define (trans text from to)
5:   (html-parse
6:     (http-parse-body
7:      (http-post
8:       "http://translate.google.com/translate_t"
9:       :text text
10:      :langpair (format "~a|~a" from to)))
11:    (lambda (markup attrs body)
12:       (if (and (eq? markup 'div)
13:                (equal? (assq 'id attrs)
14:                        '(id "result_box")))
15:           (car body)))))
```

Figure 1. `gtranslate.scm`: A Bigloo module

We consider here a simple package named `gtranslate` that exports a single function: `trans`. This function translates texts written in a natural language into another natural language. It takes three parameters: the text to be translated, the name of the source language, and the name of the output language. This package is implemented in the Bigloo dialect [6]. The source code adapted to fit the size constraint of this paper is presented in Figure 1. It may be used as:

```
(trans (trans "Il pleut" "fr" "en") "en" "es")
   ==> Llueve
```

The function `trans` is a simple wrapper to the Google translation tools. It merely opens an HTTP connection (line *8*), parses the HTML result (line *5*) and extracts the result that is contained in the box (a HTML `div` element) named `result_box` (line *14*). This implementation uses several of Bigloo's features. First, it is organized as a module that is declared on line *1*.

Regardless of Bigloo's specific features that are used in this code, transforming this module into a ScmPkg package is straightforward. The only mandatory effort is to provide an *interface* file, suffixed with `.spi` and to bundle the interface and the implementation files in a tarball file named `gtranslate-0.0.2.tar.gz`. The interface file is given Figure 2.

```
1: (interface gtranslate
2:    (version "0.0.2")
3:    (language bigloo)
4:    (import http-utils html-parse)
5:    (export (trans text from to)))
```

Figure 2. `gtranslate.spi`: The interface

Since the interface file is a mere export list, it can be automatically generated from the original source code. This even holds if, contrary to this example, the original source code is implemented for a dialect that does not provide modules. In our example, the interface mirrors the Bigloo module but in addition it gives important information: line *3* tells in which language (which Scheme dialect) the package is implemented. In Section 3 we present the meaning of this statement and how it is handled. For now, it is sufficient to know that each package is implemented in a Scheme dialect (which defaults to *Scheme R^{5}RS*) and that this information is used by the package managers to *adapt* the source file to its host. The interface may also contain optional meta information such as the the author address or a description of the package. This information is mainly used by the ScmPkg Web server.

The implementation of `translate` is as compact because it uses various pre-existing packages for dealing with web programming. These packages are not required to be all implemented in Bigloo. Indeed, in this particular example, the package `html-parse` that provides functions for dealing with URLs is implemented in the STklos dialect [1]. Its implementation is sketched in Figure 3. It may be observed that the structure of the STklos module is fundamentally different from the structure of the Bigloo module. *i)* STklos modules are closed, i.e., the declarations and definitions are lexically nested inside the module declaration. *ii)* Import and export clauses are *floating* inside the module. That is, they don't have the syntactic obligation to follow the module declaration. *iii)* Export clauses only specify names, by contrast with Bigloo where they specify full prototypes. However, in spite of these differences, it is as easy to make a ScmPkg out of this STklos code. It is given in Figure 4. Provided with this interface declaration, the package can be used by any Scheme host, regardless of its actual implementation language.

```
1: (define-module html-parse
2:   (import xml-parse)
3:   (define (html-parse port constructor) ...)
4:   (export html-parse))
```

Figure 3. `html-parse.stk`: A STklos source code

For the sake of the example, let us show how this package can be used in Bigloo. This host comes with a command named `bglpkg` that installs packages. It may be used as:

```
1: (interface html-parse
2:    (language stklos)
3:    (import xml-parse)
4:    (export (html-parse port constructor)))
```

Figure 4. `html-parse.spi`: An interface

```
$ bglpkg -v gtranslate  # download pkg and its dependencies
http-utils 0.0.2:
@stklos 1.0.0:
html-parse 1.0.0:
google-translate 0.0.2:
xml-parse 1.0.1:
$ make                  # build gtranslate library
bigloo -c -spi -O3 http-utils.spi -o http-utils.o
bigloo -c -spi -O3 @stklos.spi -o @stklos.o
bigloo -c -spi -O3 html-parse.spi -o html-parse.o
bigloo -c -spi -O3 gtranslate.spi -o gtranslate.o
libgtranslate.so, libgtranslate.a done
$ bigloo foo.scm -lgtranslate -o a.out # compile
$ a.out "il pleut" fr en               # run
it rains
```

The `bglpkg` tool that is shipped with the Bigloo distribution, installs packages and it manages dependencies between packages. Hence, it has downloaded the package `gtranslate` and all the packages it recursively depends on. By convention the packages whose name starts with the `@` sign denote languages implementations. All Scheme hosts that want to use ScmPkg have to provide facilities for installing packages.

In conclusion, in this Section, we have presented a complete example of a package. We have shown that a package contains a specification of the dialect of Scheme it is implemented in. We have shown that packages may depend on other packages that are not required to share the same implementation dialect. That is, it is possible to blend packages regardless of their implementation language. In particular, we have shown that the Bigloo compiler is able to compile packages implemented in its own dialect or in the STklos dialect granted the code is associated with an interface declaration.

After this introductory example, let us now present the actual organization of a package and the actual syntax of the ScmPkg IDL used for describing the interfaces.

2.2 Package architecture

In this section we briefly present the organization of the files inside a package. As we have already seen, a package `pkg` must, at least, contain an interface description located inside a file named `pkg.spi`. Generally, there is an implementation file associated with the package. This is not a strict requirement because a package may simply re-export functions, variables, or macros that are already implemented elsewhere. We will also see that, with package adaptation, it may happen that a package has no portable implementation. The specification of packages does not enforce constraint on the name of the implementation file but in general, it is named `pkg.scm`.

By convention, the tools composing the ScmPkg systems search for some extra files.

- The file `doc/pkg.wiki`, `doc/pkg.txt`, and `doc/pkg.-html` are considered as documentation. According to its general principle, ScmPkg does not impose any particular file format for the documentation. It encourages the adoption of the wiki syntax because it is compact and unobtrusive but it also supports plain HTML and text.
- The files `test/pkg-test.spi` and `test/pkg-test.-scm`, if present, implement a test suite for the package. These can be used to automatically test a package on a Scheme host. If a package provides tests, they can be run by the ScmPkg host *package manager*.

2.3 ScmPkg Interface Description Language

In order to facilitate code reuse, ScmPkg package interface descriptions located in a file different from the actual implementation source code. This permits users to grab an existing Scheme source file and to augment it with the information required by ScmPkg *without editing the initial file*. The role of the package interface consists mainly in providing five pieces of information: *i)* the name of the package, *ii)* the packages it depends on, *iii)* the variables, functions, and macros it exports, *iv)* the dialect it is implemented in, *v)* the location of the actual source file.

The main syntax of a package thus described by the following grammar:

```
<intf> ⟶ ( interface <pkg-name> <intf-clause>* )
<intf-clause> ⟶ <language-clause>
     | <meta-clause>
     | <import-clause>
     | <export-clause>
     | <extension-clause>
<extension-clause> ⟶ ( <symbol> ... )
```

The language clause specifies in which language a package is implemented but it also gives information on the subset of that language that is actually needed. For instance, the following declaration:

```
(interface pkg (language bigloo parser match))
```

introduces a package implemented in the Bigloo language that, in addition to the basic support, also requires the Bigloo parsing and pattern matching facilities. Section 2.4 will focus on languages and language features. We only present here their syntax which is defined by:

```
<language-clause> ⟶ ( language <name> <feature>* )
```

The *meta-clause* gives information on the package itself such as the version number or the name of the implementation file. It is important in the context of ScmPkg not to make assumptions on the names of the files. Since we want to reuse code as is and since various hosts use various conventions (e.g., Bigloo assumes that source files are suf-

fixed with either `.scm` or `.bgl`, MzScheme assumes a suffix `.ss`, STklos uses `.stk`, etc.) it is necessary to introduce customization of the file names.

```
<meta-clause> ⟶ ( suffix <string> )
     | ( source <file> )
     | ( version <version> )
     | ...
<file> ⟶ <string>
<version> ⟶ <string>
```

It is possible to import a whole package (and use all its exports) or to restrict the import to some bindings. The syntax of an import clause is:

```
<import-clause> ⟶ ( import <import>* )
<import> ⟶ <pkg-name>
     | ( <pkg-name> <symbol>+ )
```

ScmPkg supports two export forms. A *direct* export that exports bindings defined in a package. An *indirect* export that lets a package export bindings provided by another package.

```
<export-clause> ⟶ ( export <export>* )
<export> ⟶ <direct-export>
     | <indirect-export>
<direct-export> ⟶ <variable>
     | <function>
     | <macro>
     | <syntax>
<indirect-export> ⟶ ( from <pkg-name> <export>* )
```

A binding is exported with its prototype. That is, the export clause distinguishes between variables, functions, and macros. In the case of a function, the export specifies the full prototype of the function. This gives opportunities to optimizing compilers.

```
<variable> ⟶ <ident>
<function> ⟶ ( <ident> <r5rs-formals> )
     | ( <ident> <srfi89-formals> )
<macro> ⟶ ( macro ( <ident> <r5rs-formals> ) )
<syntax> ⟶ ( syntax <ident> )
```

R^{5}RS formal parameter list corresponds to traditional Scheme code. It supports fixed arity functions or variable arity functions. SRFI89 formal parameter list is an extension to R^{5}RS that correspond to the DSSSL [2] language with a slightly diverging syntax. It supports named and optional parameters for functions.

2.4 The Languages

A Package is implemented in a *language*. A Language is implemented by a package that, by convention, is named after the language name prefixed with the `@` sign. Specifying that a package `pkg` is implemented in the language `lang` is roughly equivalent to specifying that `pkg` imports `@lang`. In addition, the `language` interface clause gives information about the nature of the implementation. For instance, it sets a default suffix for the source files. ScmPkg also offers opportunities to apply transformations to a source code. Due to space constraints, this is not presented in this paper. For now, we can consider a language declaration as a mere shorthand for an import and a suffix declaration. So we can consider that the interface `intf1` and `intf2` given in Figure 5 are equivalent.

```
(interface intf1
   (language stklos)
   (export (fun1 a b)))
```

```
(interface intf2
   (import @stklos)
   (suffix "stk")
   (export (fun1 a b)))
```

Figure 5. Two equivalent interfaces

Languages are used in ScmPkg for ensuring portability between packages. They implement the minimal glue that is needed for reusing code from one dialect into another one. In Section 2.1, we have seen an example using a package implemented in the `stklos` language. Let us study now the implementation of that language. As presented, STklos provides modules that are introduced by the `define-module` form. These modules embed the definitions of functions, variables, and macros. All Scheme dialects but STklos must simply extract these definitions and ignore the module declaration itself. This can be easily implemented with a macro. Hence, the package interface exports the macros needed for *erasing* the STklos module idiosyncrasies. Its interface is presented in Figure 6. An implementation of that package

```
(interface @stklos
   (export (macro (define-module name . body))
           (macro (export . rest))
           (macro (import . rest))))
```

Figure 6. `@stklos.spi`: The interface of the STklos language

is straightforward and quite standard. It is given in Figure 7.

```
(define-macro (define-module name . body) body)
(define-macro (export . rest) #f)
(define-macro (import . rest) #f)
```

Figure 7. `@stklos.scm`: An implementation of the STklos language

We have found it useful to split language specifications in small pieces. This has two advantages. *i)* It allows a package to use only the relevant parts of the library of the system it is implemented in. *ii)* It allows us to provide incremental support for each language. First we can provide a minimal support for each system and then, on demand, we add new *features* to that language by the mean of new packages. For instance, as mentioned in Section 2.3 Bigloo supports parsing facilities. Not all packages implemented in Bigloo need them. So they are not specified in the package implemented the `bigloo` language. They are specified in a package named `@bigloo-parser` that is developed and maintained independently of the first one. Language features may be specified on the `language` interface specification. They are handled as a shorthand for an export. That is, the two interfaces presented in Figure 8 are equivalent.

```
(interface intf-avec      (interface intf-sans
 (language bigloo parser)) (language bigloo)
                           (import @bigloo-parser))
```

Figure 8. Using language features

In Section 3.1, we will see that *host adapters* can eliminate the performance penalty that could be associated with the language abstraction promoted by ScmPkg. We will see that, in general, a package can be optimally compiled or interpreted by a host even if that package was not initially intended to be used with that system.

3. Host Adaptation

Before being compiled or interpreted a package may have to be *adapted* to a Scheme host. That is, some parts of its interface and its implementation may have to be rewritten. It is handled automatically by the host package manager. This process is at the heart of the whole system. The general idea of adaptation is that Scheme hosts are compatible to a large extent. They all support a common set of extensions even though they don't share the same interfaces. Sometimes, *reusing* a source code implemented for a host `H1` within a host `H2` requires us to provide glue that is made of additional functions and macros. It may also require us to override some definitions of the original source code (either because `H2` allows a faster implementation or because the `H1` implementation uses features that have no counterpart in `H2`). ScmPkg proposes a methodology for automatically processing *host adaptation*. This is the subject of this section.

3.1 The Adapters

In order to proceed to package adaptation, a ScmPkg package manager searches for an *adapter*. An adapter is a bundle of files which is generally stored along the package itself on the ScmPkg server. Each file of an adapter contains *rewriting* rules of the initial package interface and implementation. The name of an adapter bundle is obtained by concatenating the name of the package, the `_` character, the name of the host, and the package version number. So, the name of the adapter for the STklos host of the package `pkg` version 0.0.2 is `pkg_stklos-0.0.2.tar.gz`.

An adapter is a set of files that are used to rewrite the original interface and implementation. Each package manager is free to implement and support its own rules but we also propose a set of rules that covers all the possible needs for rewriting we have met so far. These rules are shared by Bigloo and STklos. They are presented in the rest of this Section. In this whole section we assume a package `pkg` whose interface file is named `pkg/pkg.spi` and its implementation lies in `pkg/pkg.scm`.

We have found of premium importance to release adapters independently of packages. This allows a lightweight development process that is impossible with conditional compilation. A programmer may implement and maintain one package on a host `H1`, totally ignoring that his package is adapted to other hosts. Independently, a user of the host `H2` could be interested in that package. If this package needs to be adapted for `H2`, he can write an adapter and release it independently of the package. The two authors don't have to synchronize. The developer of the package does not have to integrate in his development tree the modifications required for hosts that he may never install on his machine. The developer of the adapter does not have to wait until his modifications have been integrated in the main development tree of the package to offer it to the `H2` community.

3.2 Implementation Adaptation

In this Section we present the adaptation of the implementations. In the next section we will present the adaptation of the interfaces. For the sake of simplicity, we will only present the adaptation in the context of the Bigloo system. The very same principles apply to other systems.

3.2.1 Rewriting the whole package

Bigloo compiles modules. Its future version will also support natively the compilation of interfaces. This is simply implemented inside the compiler by a macro that expands the `interface` form as defined in Section 2.3 into a plain module clause declaration. This expansion is so simple and direct that it does not deserve any presentation here.

A common adaptation framework for Bigloo consists in simply removing the entire interface and implementation of the ScmPkg package. This, for instance, corresponds to situations where Bigloo natively supports the features exposed by a package (e.g., Bigloo native SRFIs). This first rewriting is specified by the rule 1 defined as:

> *Rule 1: If the file `pkg/bigloo/pkg.bgl` exists in the adapter, remove the files `pkg.spi` and `pkg.scm` and use that file instead.*

Even if this rule is extremely simple, let us apply it to an obvious example. Each language package is natively supported by one host. For instance, the `@bigloo` package that implements a minimal Bigloo runtime is *de facto* supported by the Bigloo compiler. So, when Bigloo compiles a package that depends on the `@bigloo` package (i.e., a package that is implemented in the `bigloo` language) it must then also compile that module! Since, by construction, Bigloo natively supports the `bigloo` language, the package `@bigloo` has to be replaced with an empty specification. Hence the Bigloo adapter `@bigloo/bigloo/@bigloo.bgl` is defined as:

```
(module @bigloo)
```

Of course, some situations are not as simple and the substituted file contains actual code.

3.2.2 Adding extra code

Sometimes, it may be necessary to inject code in the implementation before or after `pkg.scm`.

Rule 2: If the file `pkg/bigloo/pkg-before.scm` exists in the adapter, it is inserted in front of the initial file `pkg.scm`. If the file `pkg/bigloo/pkg-after.scm` exists, it is appended to the source file.

Let us illustrate this adaptation with one example extracted from the `kishi` package which is a simple chess game, implemented in the Chicken Scheme dialect. Its interface is:

```
;; the source file is implemented in Chicken and it
;; uses several standard functions of that dialect
(interface kishi
   (language chicken level2)
   (import srfi6 srfi13 srfi26)
   (export (kishi args)))
```

The function `kishi` implements `And/Or` trees and extensively uses continuations to abort computations. Bigloo support for continuations is extremely inefficient. However, it supports fast exceptions. So, in order to improve the performance for Bigloo, it is valuable to replace the continuations with exceptions. This can be easily implemented by adding a simple macro definition in front of the initial source code. This is the role of the file `kishi/bigloo/kishi-before.scm`:

```
(define-expander call/cc
   (lambda (x e)
      (match-case x
         ((?- (lambda (?var) . ?body))
          (e `(bind-exit (,var) ,@body) e)))))
```

This defines an expander named `call/cc` that actually overrides the definition of the standard function. It rewrites any calls to `call/cc` into exception blocks (`bind-exit` is comparable to the Java's `try` block). Of course, this transformation is, in general, incorrect. It can be applied to the `kishi` package only because continuations are used in their dynamic extend to abort computations. Note also, that this overriding only applies to the `kishi` package and an application using this package can still use the `call/cc` primitive in all its glory in other packages, if needed.

With this example we have demonstrated that host adaptation can be used to tune an implementation for improving the performance on a host, without modifying the original source code. This is another important property of ScmPkg: with the framework we propose, in general, the packages can be tuned for specific hosts and specific applications. Hence, they are efficient on all platforms. We think that this is a fundamental property. It is mandatory to guarantee that the package organization does not impede the performance of the application. Otherwise, users would be reluctant, with reason, to use such a system.

3.2.3 Replacing the implementation

In rare situations, it might be necessary to replace an entire implementation. This corresponds to situations where a package implementation uses a specific feature that is only available on some hosts. This rewriting is the purpose of the following rule:

Rule 3: If the file `pkg/bigloo/pkg.scm` exists, it replaces the original package implementation.

This situation is of course the least satisfying, since if forces a complete rewrite. However, the adaptated (or rewritten) package can transparently be integrated with the other packages of an application.

A more common case of adaptation consists of slighly modifying the initial source code for changing not the entire implementation but only a couple of definitions. This is the purpose of our fourth adaptation rule.

Rule 4: If the file `pkg/bigloo/pkg-override.scm` exists in the adapter, all the definitions found in this file shadow the corresponding ones in the original source code.

Generally, only a couple of definitions are present in the override file; all other definitions of the package implementation are left unchanged.

We illustrate this adaptation with the `match` package whose implementation consists of a code provided by A. Wright that gave to the community a mostly portable implementation of pattern matching in Scheme. The interface for that package is:

```
(interface match
   (export (match-error obj)
           (macro (match . args))
           (macro (match-lambda . args))))
```

As most Scheme code, the pattern matcher has to deal with errors. Unfortunately, Scheme does not provide any official mean for raising errors. Hence, *portable* Scheme codes have to simulate raising an error. In the case of the `match` package, it is implemented as:

```
(define (match-error obj)
   (display (format "*** MATCH FAILED: ~a" obj))
   (/ 1 0))
```

This implementation displays an error message and raises a native error. This trick is common because there is no better way to implement *portable errors*. However, it is also quite inconvenient because it does not leave opportunity to the rest of the program to intercept the error and to make something useful out of it. The system will think a floating point exception has occured instead of a pattern matching error! This situation is very common because it is in the very nature of library functions to raise exceptions that are expected to be managed by the main program. Bigloo, as most Scheme hosts, provides functions for raising and catching exceptions. So we can adapt the definition of `match-error` to Bigloo by replacing the original definition with a better one that raises a true exception:

```
(define (match-error obj)
   (raise (instantiate::&match-error
             (proc 'match)
             (msg "match failed")
             (obj obj))))
```

With this example we have demonstrated that using the ScmPkg host adaptation it is possible to *naturally* blend

packages implemented in different dialects. In particular, we have shown how the errors and exceptions of the packages can be mapped to the native exception mechanism of the Scheme host.

3.3 Interface Adaptation

As it may be required to modify the implementation of a package, it may also be required to change its interface. For instance, it can be required to *erase* an export for a host that provides natively a function or a macro or it can be useful to add *ad hoc* annotations to an export in order to give opportunities to an optimizing compiler. This adaptation of the interface is described in this section.

3.3.1 Adapting the interface

The rules for adapting a package interface are similar to the rules for adapting an implementation. The simplest one, replaces a whole package interface.

Rule 5: If the file `pkg/bigloo/pkg.spi` is present, it replaces the whole package interface.

This is a coarse-grain rule that, hopefully, is not often used. The most useful rule is the following:

Rule 6: If the file `pkg/bigloo/pkg-after.spi` is present, its content is appended to the package interface.

This rule allows one to *inject* extra clauses to a package interface. Combined with the host interface extensions this rule permits one to precisely tune a package interface. In particular, it permits one to add annotations that opens opportunities for optimizations.

3.3.2 Host interface extensions

As we have presented in Section 2.3 the syntax of the interface provides an open door to extensions. This is used for adapting the interfaces. This can either be required if the package contains incompatibilities with a given host or simply for improving the performance. In order to simplify the current presentation we will keep focusing on the extensions provided for the Bigloo system. Adapting the rules we are going to present to another Scheme host is, however, straightforward. One of the goals of this section is to demonstrate that the portability model proposed by ScmPkg does not impact the performance. This section does not contain any material essential to the understanding of the overall architecture of ScmPkg. Hence, it can be skip for a first read.

The general idea of the `<extension-clause>` rule of the syntax of the interface is to permit annotations that are ignored by all hosts but one. In the case of Bigloo, this extension is marked by the `bigloo` keyword. This is denoted by a new rule added to the syntax of the IDL:

```
<extension-clause> ⟶ ( bigloo <bigloo-module>+ )
     | ( <symbol> ... )
<bigloo-module> ⟶ (module-override)
     | (export-override <bigloo-export>+)
     | (export-replace <replace-clause>+)
<replace-clause> ⟶ ( <ident> <bigloo-export> )
<bigloo-export> ⟶ See the bigloo user manual
```

Bigloo handles these extensions as follows:

Bigloo Rule 1: If `module-override` is present in the `bigloo` extension clause, the Bigloo module declaration found in the implementation file replaces the whole package interface.

This rule is frequently used for transforming an existing Bigloo module into a ScmPkg package.

The package interface is less precise than the Bigloo module declaration. In particular, because the latter contains type annotations and other information useful for the compiler. In order not to impede the performance when a Bigloo module is transformed into a package, it is thus important not to lose this extra information. For the sake of the example, let us assume the following Bigloo module:

```
(module xml-parse
 (export (xml-parse::list in::input-port k::proc)))
```

This module exports one function that accepts an input port and a procedure as parameters and returns a list. This module can be translated in a package whose interface could be:

```
(interface xml-parse
   (bigloo (module-override))
   (export (xml-parse in k)))
```

Without the `(bigloo (module-override))` clause, the type annotations would be lost and the compiler would be no longer able to statically type check a package importing `xml-parse`. With the `module-override` clause, the compiler ignores the package interface and actually uses the more precise module declaration.

Bigloo Rule 2: An `export-override` clause overrides the export clause of the package interface with a more specific Bigloo one. The rest of the package interface is left unchanged.

This rule is used to change the prototype of an export. It can be used for adding type annotations or compiler annotations such as inlining information. For instance, in Section 2.1, Figure 4 we have presented a package implemented in STklos for parsing HTML. In order to improve the performance and the error messages produced by the compiler it could be useful to add type annotations to the function `html-parse`. This can be accomplished with a `export-override` clause such as:

```
(bigloo
 (export-override
  (html-parse::list in::input-port k::procedure)))
```

This clause takes place in the Bigloo interface adapter, that is in the file `html-utils/bigloo/html-parse-after.spi`.

Bigloo Rule 3: An `export-replace` clause replaces an export clause with a different one or it simply erases an export clause.

The `export-replace` Bigloo extension clause is a generalization of the `export-override` clause. By contrast with the former, it allows a clause to remove an export while the previous is only able to change it. Let us illustrate this new functionality with an example. The package `@mzscheme` implementing a subset of the MzScheme dialect in ScmPkg exports a random number generator. Bigloo already provides such a facility with the very same interface. The Bigloo adapter for the `@mzscheme` must then *erase* that export. This is accomplished the using a `export-replace` rule in the file `@mzscheme/bigloo/@mzscheme-after.spi`:

```
(bigloo
  (export-replace
    (random #f)))
```

All the others functions, macros, and variables exported by the `@mzscheme` module are left unchanged.

In this section, we have shown how package interfaces can be adapted to Scheme hosts. In particular, we have demonstrated that the abstraction layer introduced by the package interfaces does not jeopardize the performance of a system because the interface adaptation permits a host to add the annotations that are required by its optimizer.

4. ScmPkg Cost of Entry

Up to now, we have been able to blend packages written in the following Scheme dialects: Bigloo, Chicken, Gambit, Mzscheme, R^{5}RS, Snow, and STklos. However, currently, the ScmPkg packaging system is fully operational on only two hosts, Bigloo and STklos, and some partial support is also available for Mzcheme. This is clearly not sufficient and we know that the system must be adapted to other hosts in order to achieve the goal it has been assigned to. In this section, we expose the different tasks that need to be completed in order to offer ScmPkg on a new host.

The first task consists in supporting the ScmPkg IDL, that is the `interface` form, in the new host. This can be implemented by a macro which rewrites the `interface` form into directives for the underlying host. If this host provides a module system or namespaces, this macro is simple to write. For instance, the whole support of the ScmPkg IDL for STklos, is less than 250 lines long. For systems which don't have a module system or namespaces, this task can be tedious because an *ad hoc* mechanism for isolating identifiers inside interfaces has to be invented. Probably ScmPkg is probably not well adapted to such systems. However, amongst the current mainstream Scheme implementations, SCM seems to be the only one which is in this case.

Native support of the ScmPkg IDL permits us to make small experiments with the system, but we have seen that the package manager plays a key role for adapting a package to the host. The complexity of the package manager depends on the level of integration envisioned for this tool. However, the rules shown before for package adaptation are simple to implement since they consist mainly in file copying and renaming.

Finally, to support all the packages provided by the ScmPkg repository, a final task consists in writing adaptations for the languages supported by ScmPkg. These adaptations are generally straightforward to implement since they only involve simple macros such as the ones shown Figure 7. To give an idea, the average length of the language adaptations we have written so far is around 200 lines per host.

As we can see, adding support for ScmPkg to another Scheme system does not require a tremendous work to implementors because there is no need to modify the packages individually. **Only the packages implementing languages need per-host attention**. We hope that it is sufficiently low to convince the Scheme dialects authors, or advanced users of their community, to invest in the implementation of adapted ScmPkg tools.

5. ScmPkg Status and Related Work

ScmPkg is still young and until recently each new version was deeply incompatible with previous versions. The mechanics for host adaptation as presented in the paper have only been designed after several endeavors that have shown to be inconvenient and tedious to use. Hence, ScmPkg has not been yet publicly announced so we don't have yet external users contributions. The only packages that populate the actual system have all been "assembled" by the authors. However, we have paid attention to exercise as much as possible the portability supported by ScmPkg. We have ported existing Scheme code that was coming from different sources.

Currently, ScmPkg contains approximately 100 packages amongst which we have ported code coming from seven different hosts. For each package, we have refrained from manually tuning the original source code. We have pushed the host adaptation to its limits, in order to validate our approach. We have had varying degrees of success depending on the source of the packages. For each of these system we have implemented corresponding ScmPkg language and we have been able to automatically import packages.

5.1 SRFI

Several attempts have already attempted to build a community of users for Scheme. The *Scheme Request For Implementation* process is of one them. It has been initiated at the end of the 90's. It claims to be an "*approach to helping Scheme users to write portable and yet useful code. It is a forum for people interested in coordinating libraries and other additions to the Scheme language between implementations*". Today, there are approximatively ninety SRFIs. Few of them provide libraries and the ones that do, actually provide libraries for classical manipulation of data structures

(e.g., list and string manipulation, hash tables, vectors of all kinds, etc.). The majority of SRFIs specifies syntactic extensions to the language. While this can be useful, it does not help for writing the applications we want/need to write today.

The SRFI process has proven to be useful for normalizing the usages for the extensions that are present in the major Scheme implementations (e.g., hash tables or string ports). However, nine years after the first SRFI submission, we think that the whole SRFI process has failed at being a repository of useful APIs for Scheme.

The SRFI process was probably developed too late. Scheme was more than twenty years old when the SRFI process started. As a consequence, most of the major features necessary to build *modern applications* were already present in the main implementations under a form or another. As a consequence, it is very difficult to ask implementors to change features that are central for their implementation. In this respect, we think that this is enlightening to see that the only proposition for defining an object system (SRFI-20) was withdrawn after a discussion involving four comments (none being really negative), whereas the SRFI (SRFI-48) which define the `format` function was accepted after more than 60 comments. To date, nobody was courageous enough to propose a SRFI for important aspects such as modules or network programming which are really primordial for writing *modern applications*. This enforces our opinion that it is very difficult to impose new programming habits to Scheme users and that a solution built around existing systems is preferable.

SRFIs exposes variables, functions and syntaxes that can handled by ScmPkg interfaces. Since the SRFI process requests a reference implementation, adapting a given SRFI to ScmPkg is straightforward: it simply consists in bundling this implementation with an interface declaration file (a `.spi` file).

5.2 Dedicated Package Management Systems

Several implementations of Scheme propose their own package management systems. Chicken with its Eggs Unlimited has been one of the first systems to propose a packaging system. It contains many useful packages for modern programming. For several years, it has been imitated by PLaneT [4] a deployment system for PLT Scheme. This system too contains numerous valuable packages. The problem with these systems is that while they definitively add extra value to their respective implementation and to their own community of users, they do not unify the Scheme community, in general. The packages they deliver are intended exclusively for their systems.

5.2.1 Eggs Unlimited

From Chicken's Eggs Unlimited we have been able to reuse the packages implementing network protocols (ftp, irc, pop3, ...). We have focused on these packages because they implement useful non-trivial facilities and because they are fully implemented in Scheme while many Chicken packages are actually implemented, partially or integrally, in C.

In order to reuse these packages we have implemented a small subset of the Chicken's runtime system inside the `@chicken` language. Since these packages deal with network programming, we have also exported the Chicken network facilities in the `@chicken-net` package. Once this has been accomplished, creating the ScmPkg packages has only required a couple of minutes.

5.2.2 PLaneT

PLaneT is a repository of packages for PLT Scheme. For this system, we have created an extension to the package interface syntax. We have created a new entry in the syntax of the interface that allows interfaces to refer to a planet from a ScmPkg package exactly as the PLaneT users are used to. For instance, we write:

```
(interface password
  (planet
   (require
    (planet "password.ss"
            ("schematics" "password.plt" 1 0))))
  (language mzscheme)
  (import srfi13 planet-macros random leet)
  (export max-length min-length
          (string->password string)
          (make-passwords . args)))
```

This `planet` extension tells the host package managers to download the package from the PLaneT web site and install it locally. We have been able to reuse several PLaneT packages without even editing them and even though they use MzScheme extensions such as its infix syntax. Currently our implementation of the MzScheme language is a little bit too restricted to import many packages. We have promising results but we also must put more energy for reusing the majority of the packages proposed by this system.

5.3 Centrifugal approaches

Two main projects have adopted the centrifugal approach: the Snow project and the *Revised ^{6} Report on Scheme*. They are discussed in this Section.

5.3.1 Snow

Snow is a portable packaging system for Scheme. It relies on an extension of the Scheme programming language for providing portable packages. Snow is an entire package management system. In particular, it proposes its own portable way for installing and using packages. The idea governing the system is to make the Scheme host as transparent as possible. In other words, one should use Snow packages in the very same way, whatever the Scheme host. As a consequence, if a user adheres to the Snow's conventions and restricts himself to only use them, it is likely that his program can run on another host running Snow. This goes in the exact opposite direction from ScmPkg, where the user is

free to use the features of his host system and let the tools offered by ScmPkg do the necessary adaptation for him.

Snow does not propose host adaptations. It relies extensively on conditional compilation directives. We think that this approach is error-prone and hard to maintain. It enforces a centralized heavyweight architecture for packaging and releasing codes that are avoided with host adaptation.

Since Snow offers, by nature, a well formalized system and since it presents interesting packages, we have implemented a simple tool (entirely with ScmPkg packages) that automatically generates ScmPkg interfaces from Snow descriptions. Hence, via ScmPkg we deliver *all* the Snow packages. That is, we have successfully bootstrapped Snow in ScmPkg. In particular, we are able to natively compile all the Snow packages with Bigloo and STklos, via ScmPkg.

5.3.2 R6RS

R^{6}RS is the next evolution of Scheme. It should be completed mid 2007. It extends the language in two directions: a new core language and a new standard set of libraries. While no R^{6}RS implementation exists yet, we have started to implement partial support within ScmPkg.

First, we have implemented a `@r6rs` package that contains the new function and syntax definitions of the R^{6}RS core language. We plan to implement the additional libraries with language features. For instance, we have already implemented the `(r6rs case-lambda)` library as the `@r6rs-case-lambda` package. This approach has some limitations.

- The R^{6}RS `library` form is a more powerful than the ScmPkg `interface`. It supports renaming, partial import and automatic prefixing. These have no direct counterparts in ScmPkg. but they can still be encoded in that system, at the price of tedious manual adaption.
- The R^{6}RS *import and export levels* have no counterpart in the ScmPkg's IDL and we don't plan to support them in a near future. This implies that the language used for implementing ScmPkg macros is restricted to the language natively supported by hosts. So, writing a portable ScmPkg macro requires attention but, on the other hand, the macro system for ScmPkg is simpler than the R^{6}RS one.
- Unicode and the full numerical tower require native support from hosts.

Consequently, ScmPkg will probably not permit one to use any (future) R^{6}RS code within a R^{5}RS implementation, but, at least, it will allow one to use the whole R^{6}RS standard library set.

5.4 Centrifugal vs centripetal approach

In this Section, we show how the systems based on the centrifugal approach are unable to tackle some subtle problems.

The study of the last Snow released packages available on the Snow repository unveils difficult issues. These packages are due to external contributors, that is people not involved in the development of Snow nor ScmPkg. These packages have been built on top of the Gambit implementation. Even if these packages are disguised in Snow packages they, for now, remain un-portable and they are only operational on Gambit, since they use functions that are specific to that host. Fixing this problem with Snow is not so easy. There are at least three solutions.

- Modify the implementation of the packages for adding conditional compilation directives. This is painful and puts an heavy burden on the shoulders of the implementors of these packages. These users must now be turned into specialists of all possible Scheme implementations because they must know now precisely the boundaries of Scheme R^{5}RS and they must know how to rewrite Gambit specificities for each of the systems Snow supports. We think that this is unrealistic.
- Another solution could consist in statically detecting that a package uses variables not defined in Scheme R^{5}RS nor the Snow corpus and to reject it. This requires a non trivial program analysis.
- A last possibility consists in adding new features to Snow for abstracting these Gambit specificities and then, rewriting the faulty packages. This requires a lot of energy and a long development cycle.

On the other hands, solving this problem with ScmPkg has been straightforward: we have created a new language, `@gambit`, and the adaptors for Bigloo and STklos. This task has taken a couple of minutes because the adaptors actually only contain aliasing. That is the specificities of Gambit used in the packages already exist in Bigloo and STklos under different names. If a new host adheres to ScmPkg, it will have to implement its own adaptor for the Gambit language but this will be developed independently of the packages themselves at a pace chosen by the maintainer of the new host.

We now exhibit a second problem of the centrifugal approach that we think is harder to deal with: some R^{5}RS codes are not portable. That is, two R^{5}RS implementation may deliver different results when evaluating the same expression. When populating the ScmPkg repository with Bigloo modules, we have found the following piece of code:

```
(quotient (inexact->exact a) 10)
```

This code uses the standard R^{5}RS primitives `quotient` and `inexact->exact`. It seems to be valid but in fact it has a flaw. This code uses a Bigloo idiosyncrasy where the function `inexact->exact` returns the integer obtained by truncating the real number `a`. On Scheme implementations provided with rational numbers, this function returns a rational which cannot be passed to the function `quotient` whose both arguments must be integers. So, in spite of its inoffen-

sive appearance, this code is not portable. This rather subtle problem is difficult to detect statically. A possible correction for this module could be to replace the faulty code by:

```
(quotient (truncate (inexact->exact a)) 10)
```

However, this is not practical for two reasons. *i)* It requires us to change the original source code. *ii)* It is very likely that this pattern is used several times and all occurrences must be fixed too. A better solution consists in adding the following definition to the package `@bigloo`, which is in charge of the Bigloo language:

```
(define inexact->exact
  (let ((i->e inexact->exact))
    (lambda (real)
      (i->e (truncate real)))))
```

This new definition, of `inexact->exact` will be used only by hosts that support rationals for packages written in Bigloo. A host not supporting rationals could also adapt this definition in its private adaptation of the `@bigloo` package. This approach definitively ensures that this function, when it appears in a Bigloo package, will have the correct behaviour when used on a host which is not Bigloo. By extension, one could imagine to use host adaptation for *mimicking* a bug of a Scheme host. That is, here again, we think that the centripetal approach used by ScmPkg gives pragmatic tools to fix the kind of difficult issues that the centrifugal approach, by construction, refuses to address.

6. Future Work

Currently, ScmPkg is supported by two Scheme hosts: Bigloo and STklos. They have sufficiently different characteristics for demonstrating that the system can be used in various contexts. Bigloo is a batch compiler. STklos is based on an interpreter and it relies on a read-eval-print loop. However, in order to demonstrate the validity of the approach, we have to make additional experiments. In particular, we are developing a prototype for MzScheme. This ScmPkg immersion will use an approach different to the one chosen for Bigloo and STklos. In order to fit into the MzScheme philosophy, we are implementing ScmPkg by overriding its `require` form that is used to import a module. That is, we are adding a new syntax for `require` that automatically downloads and adapts a package if not present in the cache. From a MzScheme point of view, using a ScmPkg will be strictly equivalent to using a PLaneT package. We already have an operational version of that port but it still need to be polished before being released.

In addition to porting ScmPkg to new Scheme hosts, we also have to add an additional feature that is not tackled by the first version: currently ScmPkg does not allow the programmer to specify packages that export types! That is, currently an interface may only export functions, variables, and syntax. This is enough for coping with Scheme R^5RS and many extensions. In particular, this is sufficient for exporting records because most of the time they come with a functional interface. Hence, exporting a record is equivalent to exporting its accessors, creators, and predicate.

7. Conclusion

In this paper we have presented ScmPkg, an *adaptive package management system for Scheme*. Contrary to other approaches this system is agnostic with respect to linguistic extensions and to execution environments. It proposes a framework for specifying *host adapters*. That is, when a package is installed, before being compiled or loaded, it is first *adapted* to a specific Scheme implementation. Using this methodology, it is now possible to blend, within a single application, source codes that have been developed using different Scheme systems. We have successfully compiled applications blending Bigloo, Chicken, MzScheme, and STklos codes.

In the paper, we have shown that the *adaptation* also allows one to tune packages for performance. That is, it enables us to take benefit of the specificities of each Scheme hosts. We have also shown that it gives a framework for coherent handling of exceptions and errors.

Currently, ScmPkg has been ported to Bigloo, an optimizing compiler, and to STklos, a byte-code interpreter. A prototypical version for MzScheme is under development.

At first glance, the system may look overly simple. It relies on a small IDL for describing packages and a set of loose rules for organizing them and for implementing adapters. However, it took us several monthes and many endeavors to reach that simplicity. In particular, the first version of the IDL was much richer and it was then difficult to describe harmoniously Scheme codes coming from different systems. It took us a lot of time to understand that language neutrality was absolutely required and that it was mandatory to impoverish the IDL as much as possible.

8. References

[1] Erick Gallesio – **STklos Reference Manual** – I3S/RR-2004-41-FR, I3S CNRS / Université de Nice - Sophia Antipolis, 2004, pp. 160.

[2] ISO/IEC – **Information technology, Processing Languages, Document Style Semantics and Specification Languages (DSSSL)** – 10179:1996(E), ISO, 1996.

[3] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – Higher-Order and Symbolic Computation, 11(1), Sep, 1998.

[4] Matthews, J. – **Component Deployment with PLanet – You Want it Where?** – Proceedings of the Seventh Workshop on Scheme and Function Programming, (University of Chicago Technical Report TR-2006-06), Oregon, USA, Sep, 2006.

[5] Raymond, E. – **The Cathedral and the Bazaar** – 2000.

[6] Serrano, M. – **Bee: an Integrated Development Environment for the Scheme Programming Language** – Journal of Functional Programming, 10(2), May, 2000, pp. 1–43.