Packaging O'Caml Code Using GODIVA

Owen Gunden
ogunden@stwing.upenn.edu

William Lovas
wlovas@stwing.upenn.edu

Advised by E Lewis
eclewis@cis.upenn.edu

April 9, 2004

Abstract

The O'Caml programming language is in dire need of a standard package management framework for reusable code distribution. GODI has been developed to fill this niche, but has not yet attained widespread popularity due to the complexity involved in GODI package creation. This paper describes GODIVA, a tool which vastly simplifies the process of creating GODI packages.

1 Introduction

The Objective Caml programming language (O'Caml) has experienced explosive growth over the past few years [4], even though there has never been a standard system for packaging, distributing, and installing software. It is generally agreed that such a system is necessary to accommodate the growing community and to facilitate further growth. An emerging project called GODI is beginning to establish itself as the standard distribution framework for O'Caml packages. GODI presents a simple interface to users for installing, upgrading, and removing packages, as shown in Figure 1(a).

Surprisingly, however, GODI has not received the same enthusiastic response as O'Caml in general. Why not? Being based on NetBSD pkgsrc [10], GODI is a highly flexible package management system. Unfortunately this flexibility is not without cost: making packages for GODI is extraordinarily complex and unsafe. Traditionally, making a GODI package entails understanding a lot of internal GODI and NetBSD details and writing a significant amount of BSD Makefile code. Figure 1(b) demonstrates the problem pictorially — making GODI packages is frightfully difficult! This is the primary reason, we believe, for GODI's relatively slow growth in popularity.

Our work is intended to simplify GODI package creation. To this end, we have written a tool called GODIVA (GODI Verpacken Assistant) which handles all of the details of GODI package creation. Packagers provide GODIVA with a simple specification for a package, and it then translates

What is GODI?

GODI (Gerd's O'Caml Distribution) is a source-based package manager based on the NetBSD pkgsrc framework [10]. To O'Caml users, GODI presents a simple and easy to use interface for installing, upgrading, and removing packages. Why source-based? Any viable O'Caml package management system must be source-based, for two reasons: (1) library code may only be used with the exact version of the O'Caml compiler that was used to compile the library, and (2) in general, code may only be used on the platform for which it was compiled. (Incidentally, the first reason alone demonstrates the importance of package management — when upgrading the compiler, all libraries must be recompiled. Imagine the hassle of upgrading a GODI-less system!) GODI's use of NetBSD pkgsrc has allowed it to mature rapidly from an "experiment" not intended for widespread use in July 2003 [13] to its present status a viable solution with a small but significant community [14].

the specification into a package that can be installed immediately in the Godi framework. As in Figure 1(c), software packagers are happy once again.

In this paper, we present in Godiva in detail. Section 2 covers the core of our work: Godiva's design. Section 3 discusses techniques used in the current implementation of Godiva. Section 4 presents several case studies and demonstrates how in each case, Godiva significantly simplifies the problem of Godi package generation. Section 5 comments on other work related to O'Caml package management or distribution. Finally, section 6 outlines a few ideas for future work on Godiva.

Godiva has already been released to the community as an open source project so that it can grow and change with the needs of its users.

2 Design

Godiva acts like a compiler: It takes a high-level human-readable specification and translates it to a low-level Godi-recognizable data format. Figure 2 illustrates Godiva's basic design and dataflow, from invocation to completion. The information about a package goes through the following five phases as it passes through Godiva:

1. Godiva *spec:* High level specification of how the software package is put together. This stage may be skipped by specifying the `-camlsyntax` flag.
2. *O'Caml record:* O'Caml data structure representing the specification. Optional components are represented by O'Caml `option` datatypes.
3. *After semantic analyzer:* Error-checked and fully-specified O'Caml data structure.
4. *After translator:* Translated data, in abstract Godi package format.
5. Godi *package:* Deposited directly into the Godi build tree.

2.1 User interface

Godiva runs at the command line. Its usage is simple:

```
Usage: godiva [OPTIONS] SPEC GODI_LOCALBASE
  -patch <patch>  includes the given patch
  -filesdir <dir>  includes the given hierarchy of files
  -replace  replaces an old package that is in the way
  -camlsyntax  use the ocaml syntax (devel only)
```

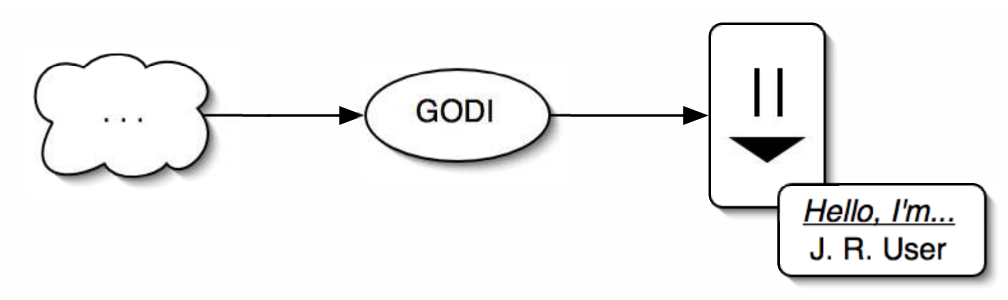
The required arguments are the two endpoints of the Godiva dataflow. `SPEC` is the specification file to be translated, which is where Godiva begins processing. `GODI_LOCALBASE` is the location of the Godi installation into which the final resulting package is to be placed. Using `GODI_LOCALBASE` allows Godiva to be used with multiple Godi installations, which is an especially common situation at this early stage in Godi's development.

Why the command line?

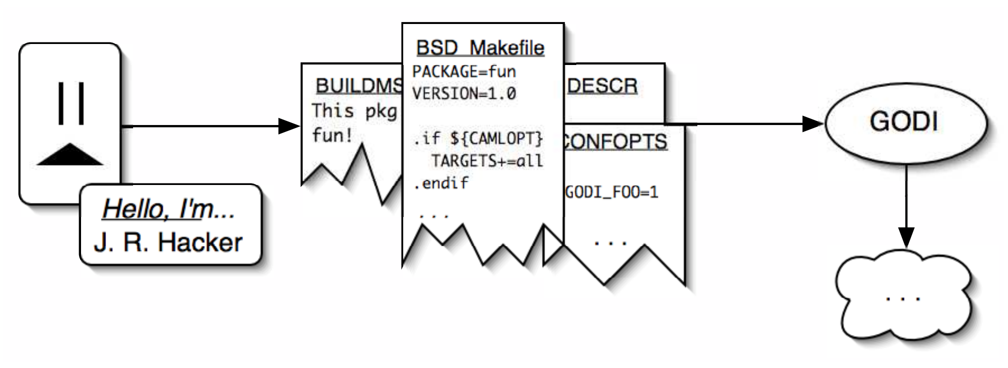
Godi runs on a wide variety of platforms, including various flavors of Unix, Mac OS X, Windows (via Cygwin), Linux, etc. The simplest way to be platform-independent is to make a command line tool. Furthermore, Godiva's operation is very simple, so the potential benefits of a graphical tool are questionable.

From time to time, a package has files that need to be patched so that it will build under Godi. When this happens, patches are given at the command line through the `-patch` option. Patches are given names like `patch-NN-hint`, where `NN` is a number used to indicate the order in which the patches should be applied, and `hint` helps tell the system which file needs to be patched. Similarly, if a package needs extra files before it is built, they can be added with the `-filesdir` option.

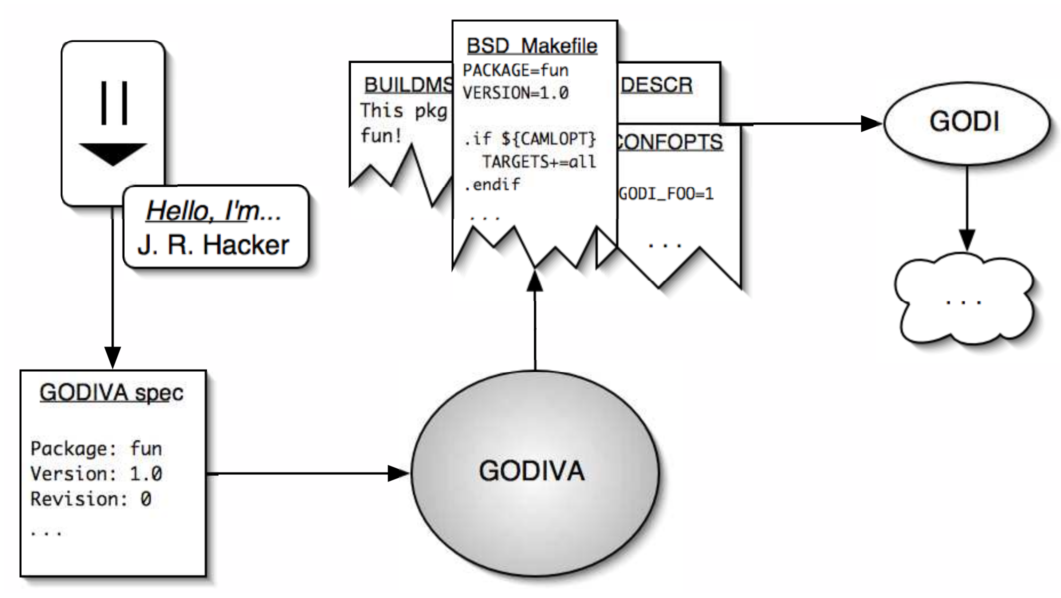
Godiva outputs packages directly into the Godi build directory. Sometimes the user needs to update a package which is already in the build directory; when this happens it is convenient to have the tool replace



(a) The world according to Godi.



(b) The rest of the story.



(c) The world is sweeter with Godiva.

Figure 1: The relationship among Godi, Godiva, and the rest of the world.

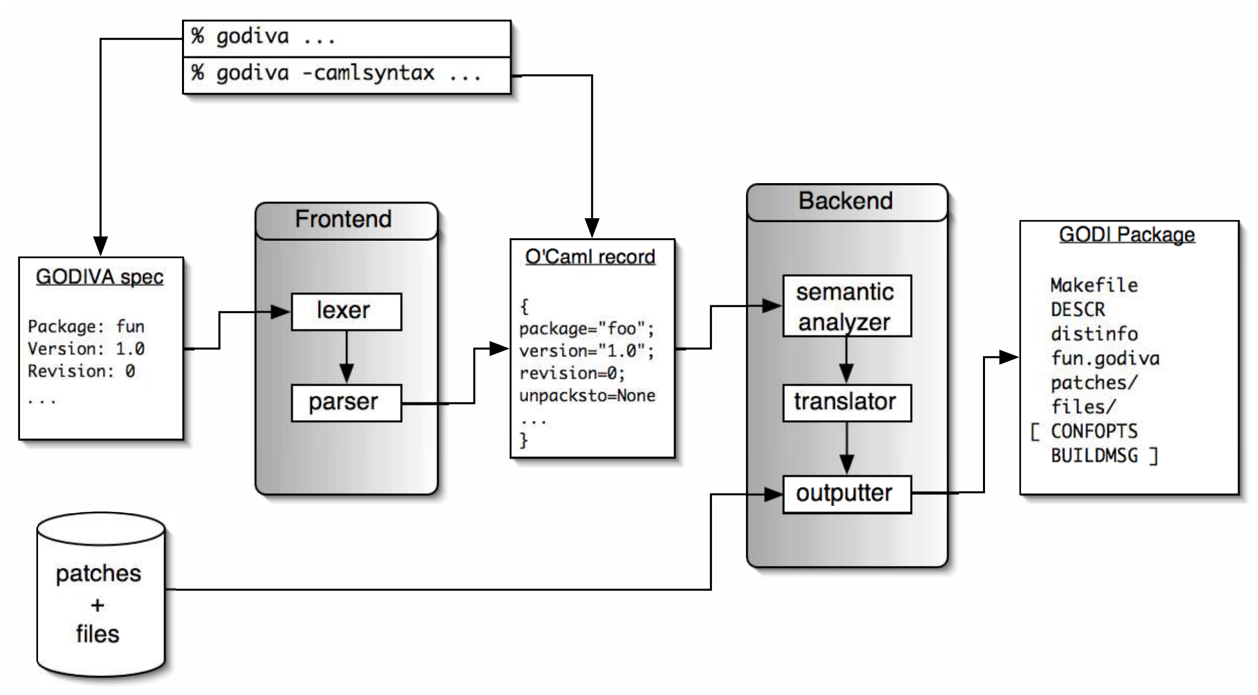


Figure 2: GODIVA's components and dataflow.

the old package with the new one, instead of failing with an "already exists" message. The `-replace` flag is used to indicate that replacement behavior is desired.

Finally, the frontend to GODIVA can be bypassed altogether by using the `-camlsyntax` flag. This functionality is primarily for developers, and is motivated further in Section 3.3.

2.2 Specification file

Our specification file format is a simple human-readable text description of various important pieces of information, including the software's name and version, the location of the source archive, the package's dependencies, short and long descriptions, etc. The information included in the specification file has been motivated by our experiences making a variety of packages. A sample file is shown in Figure 3. For comparison purposes, the corresponding GODI Makefile produced is given in Appendix A.2. Manual generation of the Makefile is significantly more complex and error-prone than manual generation of our specification file.

2.3 Deployment

A package produced by GODIVA is placed directly into the GODI build root specified on the command line. The following files are included in the GODI package:

```
Package: godi-xml-light
Version: 2.01
Revision: 0
Category: godi
Depends: godi-ocaml (>= 3.06)
Build depends: godi-findlib (>= 0.8.1)
Sources: http://www.phauna.org/~ogunden/code/xml-light2.tar.gz
Homepage: http://tech.motion-twin.com/?&node=28
Maintainer: ogunden@phauna.org
Options: opt
Docfiles: README
Description: A minimal XML parser & printer for OCaml
XML Light is a minimal XML parser & printer for OCaml. It provides functions to
parse an XML document into an OCaml data structure, work with it, and print it
back to an XML document.

Since version 2, XML Light also adds support for DTDs and PCDATA.
.
```

Figure 3: Specification file for the xml-light package.

	•	`Makefile`	the BSD Makefile used by Godi
	•	`DESCR`	the long description of the package
	•	`distinfo`	checksums, produced by Godi
	•	`package.godiva`	a copy of the spec file, for ease of maintenance
	•	`patches/`	any patches required
	•	`files/`	any extra files required
(optional)	•	`CONFOPTS`	specification of conf opts
(optional)	•	`BUILDMSG`	documentation of conf opts

After creating the Godi package, the package maintainer synchronizes his local Godi tree with the global tree, using the `subversion` revision control system [15]. Godi users may then synchronize their local trees and install the software with ease using the `godi_console` program.

2.4 Policy

In any source-based package manager, there is inherent tension between imposing restrictions on software developers and imposing complexity on package maintainers. If developers are unrestricted, packaging their software can be complicated, since package maintainers must understand and communicate to the packaging framework all the details of the building and installation process for each individual piece of software. If packages are too difficult to create, nobody will volunteer to create them. The more restrictions there are on the developer's building and installation process, the easier it becomes to package their software.

NetBSD pkgsrc must deal with a wide variety of packages, so it necessarily must place very few restrictions on developers. Godi inherits pkgsrc's philosophy. Consequently, packages are difficult to create, and Godi has not attracted many package maintainers. Godi deals with a much smaller variety of software than pkgsrc, though, so it is reasonable to move some burden from the package maintainers to the software developers. By making this change, packages will become easier to create, more people will volunteer to create them, and Godi will become a more attractive option for package management.

The burden we place on software developers takes the form of a build policy for software. Given a piece of software that adheres to the build policy, a potential package maintainer has merely to write a small specification with some details about the package and run it through GODIVA. GODIVA creates a GODI package which is immediately placed into the GODI tree where it can be used by a GODI user.

The policy is a collection of requirements ("must" rules) and recommendations ("should" rules) that serve not only to make automatic package creation possible, but also to maximize compatibility with end users' expectations. We took great care to keep our policy simple for two reasons: one, so that developers will be encouraged to adhere to it, and two, so that even if developers do not adhere to it, it will still be easy for volunteers to patch software to adhere to it. The entire policy is listed in Appendix A.1. (The policy makes reference to "findlib" [11], which is a piece of software related to package management also written by Gerd Stolpmann but much earlier than GODI. Findlib handles dependencies related to library linking. It is not a full package manager, though — it does not record any information about non-linkable files [13], for example. Nevertheless, it is an important and useful tool and GODI depends on it.)

3 Implementation

3.1 Automatic PLIST generation

Pkgsrc requires a package to list its installed files in a "packing list" file called `PLIST`. GODI inherits this requirement, but in principle, there is no reason the `PLIST` file could not be generated automatically just before installation (the `PLIST` is used to record which files and directories should be removed when a package is uninstalled). We extended GODI to do just that.

What's wrong with a PLIST?

If a package installs a large number of files, or has a confusing `make install` target, it may prove to be very difficult to create an accurate PLIST. Creating a PLIST by hand is usually the most difficult and complex part of making a GODI package. Having it done automatically means GODIVA can operate on the specification file alone — thus reducing the packager's workload enormously.

Automatic `PLIST` generation takes advantage of our build policy requirement that the `make install` step respects the setting of the `$(PREFIX)` variable. When software is being built we set `$(PREFIX)` to the final installation location of the software, in case the build step requires knowledge of this final location (for example, if it needs to know where its configuration file will be installed). However, just before we invoke the `make install` step, we set `$(PREFIX)` to a new, temporary location that is initially empty. Then, we may examine this location to determine exactly what files a package installs and make note of these in the `PLIST`. Finally, we can move the files to their final location and remove the temporary location.

Since the installed files of a package could change from platform to platform or based on user configuration options, we regenerate the `PLIST` every time a package is installed. In order to distinguish between packages with a manually generated `PLIST` and packages with an automatically generated `PLIST`, we add a configuration variable to the GODI Makefile, `AUTOGENERATE_PLIST`. Since this variable will not be defined in old-style packages, we maintain backward compatibility.

3.2 Frontend

The command line options to GODIVA are implemented using the `Arg` module in the O'Caml standard library, which provides a standard command line syntax and a powerful programming API. For lexing and parsing the specification files, we generate a tokenizer using lexing states in ocamllex and an LALR parser using ocamlyacc. The lexer and parser are very careful to return helpful error messages with exact positions when something is wrong with the syntax of the specification file.

3.3 Backend

3.3.1 Components

The backend is made up of three basic components which we call the *semantic analyzer*, the *translator*, and the *outputter*. The semantic analyzer takes an O'Caml record representing the specification and performs several checks to ensure that it is correct. For example, the `unpacksto` field should only contain one path element, and it should certainly never be something like '`../../foo`.' The semantic analyzer is also responsible for filling in defaults for any optional values, such as the make targets for building bytecode and optimized code, which default to '`all`' and '`opt`', respectively. After the specification passes through the semantic analyzer, it should not contain any semantic errors, and it should be complete.

The second component translates each field of the specification into a string that is suitable for direct inclusion into one or more files in the resulting GODI package. Often this is a trivial translation; for example the translation of the package version is accomplished with the following code:

```
translate_version v = ("VERSION = " ^ v)
```

At other times the translation is more involved. For example, each conf opt is translated into (1) a line in the resulting BSD Makefile that sets a default value for the conf opt, (2) one or more lines in the Makefile for its implementation, (3) a line in `CONFOPTS` declaring the conf opt, and (4) one or more lines in `BUILDMSG` describing it.

The final component receives the specification as a collection of strings from the translator and runs them through a template engine to create the final set of files making up the GODI package.

3.3.2 Development model

In order to rapidly implement and test new features, it is convenient to be able to skip updating the frontend of GODIVA with new syntax. Adding features to the backend is much less work. To facilitate a development model in which potential extensions may be first implemented in the backend, GODIVA provides the `-camlsyntax` option, which allows O'Caml record syntax to be used for the specification. (Section 4.1.3 contains a complete example of this syntax.) Once the extension has been well tested and its necessity and form are relatively plain, a concrete syntax is solidified in the frontend. Using this development model streamlines the implementation of new features.

3.4 Releases

Several versions of GODIVA have been released to the community. GODIVA is licensed under the GNU General Public License (Version 2) because it has always intended to be a resource that belongs to the community. GODIVA source code, documentation, news, and other resources can be found at the project webpage: `http://projects.phauna.org/godiva/`.

4 Evaluation

In this section, we present several case studies which demonstrate GODIVA's effectiveness at packaging a variety of software projects.

4.1 SpamOracle

We begin our exploration of GODIVA's merits by packaging a spam filtering program written by Xavier Leroy called SpamOracle [6].

4.1.1 The Source

Downloading and unpacking the source is straightforward, because Xavier has named his tarball `spamoracle-1.4.tar.gz` and it extracts to `spamoracle-1.4`, just as is suggested in the GODIVA policy (see Section A.1). The following line in our specification file sums this up:

```
Sources: http://pauillac.inria.fr/~xleroy/software/spamoracle-1.4.tar.gz
```

4.1.2 Observations & Challenges

A glance at the `README` file reveals that SpamOracle has no dependencies beyond the O'Caml compiler. The version is clearly 1.4, and this is an application so it belongs in the `apps` category. Our specification file now contains all the basics:

```
Package: apps-spamoracle
Version: 1.4
Revision: 0
Depends: godi-ocaml (>= 3.06)
Build-Depends:
Sources: http://pauillac.inria.fr/~xleroy/software/spamoracle-1.4.tar.gz
Homepage: http://pauillac.inria.fr/~xleroy/software.html#spamoracle
Maintainer: Owen Gunden <ogunden@phauna.org>
Options:
Docfiles: README
Description: Bayesian spam filtering software that works with procmail.
SpamOracle, a.k.a. "Saint Peter",  is a tool to help detect and filter away
"spam"  (unsolicited commercial e-mail). It proceeds by statistical analysis of
the words that appear in the e-mail, comparing the frequencies of  words with
those found in a user-provided corpus of known spam and  known legitimate
e-mail. The classification algorithm is based on  Bayes' formula, and is
described in Paul Graham's paper, A plan for spam.
.
```

The only tricky part appears to be the three configuration options which are discussed in the `README`. The installation instructions read as follows:

```
  Edit the Makefile and change the definitions of the following variables
  at the top of the file:
    LANGUAGES    the languages you're interested in besides English
    CPP          how to invoke the C preprocessor
    BINDIR       where to install the executable
```

We need to allow the user to specify values for `LANGUAGES` and `CPP` — we need conf opts here. Fortunately, GODIVA has support for translating conf opts to make variables. However, this experimental feature is not supported by the frontend, so we have to make use of `-camlsyntax`. The O'Caml syntax for conf opts is a list of records:

```
    { name = "GODI_SPAMORACLE_LANGUAGES";
      default = "";
      description = "The languages you're interested in besides English";
      implementation = `makevar ("LANGUAGES");
    };
    { name = "GODI_SPAMORACLE_CPP";
      default = "gcc -E -P \\$$(LANGUAGES) -";
      description = "How to invoke the C preprocessor";
      implementation = ‘makevar ("CPP");
    }
```

The specification of each conf opt has four components.

name is the name of the conf opt as it will be displayed by the GODI interface.

default is the default value; in this case no extra languages and a `CPP` string copied directly from the default found in SpamOracle's Makefile.

description is a string that is displayed near the conf opt when the user is setting its value.

implementation indicates how the conf opt is implemented. In this case, we want to implement both of our conf opts by overriding the make variables "`LANGUAGES`" and "`CPP`," just as the `README` told us to do.

All we have left to deal with is `BINDIR`, which tells SpamOracle where to install. Since this should be the same for everyone, we don't want to bother the user with another conf opt. So we have to patch the Makefile to set `BINDIR` to `$PREFIX/bin`. This tiny patch (`patch-00-Makefile`) will do:

```
--- Makefile.old>-------2004-03-24 22:52:41.000000000 -0500
+++ Makefile>---2004-03-24 22:53:04.000000000 -0500
@@ -7,10 +7,10 @@
 CPP=gcc -E -P $(LANGUAGES) -

 # Where to install the binary
-BINDIR=/usr/local/bin
+BINDIR=$(PREFIX)/bin

 # Where to install the man pages
-MANDIR=/usr/local/man
+MANDIR=$(PREFIX)/man

 ### End of configuration section
```

(It turns out the `README` wasn't telling the whole truth because it never mentioned `MANDIR`. We noticed this because we were good package creators and carefully studied how the "make install" target worked.)

4.1.3 Solution

In O'Caml syntax, we now have (saved to `spamoracle.godiva.ml`):

```
let spec = {
  package = "spamoracle";
  version = "1.4";
  revision = 0;
  category = ‘apps;
  depends = [(‘godi, "ocaml", (Some (‘ge, "3.06")))];
  build_depends = [];
  sources_site = "http://pauillac.inria.fr/~xleroy/software/";
  sources_basename = "spamoracle-1.4";
  sources_extension = ".tar.gz";
  sources_unpacksto = None;  (* the default unpacking location is fine *)
  homepage = "http://pauillac.inria.fr/~xleroy/software.html#spamoracle";
  maintainer = "ogunden@phauna.org";
  short_desc = "Bayesian spam filtering software that works with procmail.";
  long_desc = ("SpamOracle, a.k.a. \"Saint Peter\", is ...");
  options = [];
  confopts = [
    { name = "GODI_SPAMORACLE_LANGUAGES";
      default = "";
      description = "The languages you're interested in besides English";
      implementation = `makevar ("LANGUAGES");
    };
    { name = "GODI_SPAMORACLE_CPP";
      default = "gcc -E -P \\$$(LANGUAGES) -";
      description = "How to invoke the C preprocessor";
      implementation = `makevar ("CPP");
    }
  ];
  docfiles = ["README"];
  all_target = None; (* default target (``make all'') is fine *)
  opt_target = None; (* don't bother with opt for this package *)
}
```

We're done! Now we can run GODIVA to finish up. Our GODI installation is at `/opt/godi`.

```
% godiva -patch patch-00-Makefile -camlsyntax spamoracle.godiva.ml
This is GODIVA version 0.2.
Parsing spec file..
Using alternate syntax
ocamlfind ocamlc -package godiva -c spamoracle.godiva.ml
No filesdir supplied, skipping
Generating checksums..
Package written to /opt/godi/build/apps/apps-spamoracle.
```

At this point, our apps-spamoracle package is sitting in our GODI installation. We can install and test it, then upload it to the GODI repository using subversion.

4.2 Unison

Next we present some brief notes on our experience packaging a much larger piece of software, namely the Unison file synchronizer primarily written by Benjamin C. Pierce [9].

4.2.1 Routine Work

Like SpamOracle, Unison uses conf opts and requires a relatively minor patch. The details of both of these will be omitted in the interest of brevity.

4.2.2 What's New

Unison is distributed in a non-standard way. The distribution archive is named `src.tar.gz`, but it unpacks to `unison-2.9.20`. For such cases, GODIVA provides the optional "unpacksto" field. In the O'Caml syntax, the field is "sources_unpacksto" and it uses an option type.

4.2.3 The Solution

Our O'Caml syntax spec file begins like this:

```
let spec = {
  package = "unison";
  version = "2.9.20";
  revision = 0;
  category = ‘apps;
  depends = [(‘godi, "ocaml", (Some (‘ge, "3.06")))];
  build_depends = [];
  sources_site = "http://www.cis.upenn.edu/~bcpierce/unison/download/beta-test/unison-2.9.20/";
  sources_basename = "src";
  sources_extension = ".tar.gz";
  sources_unpacksto = Some "unison-2.9.20";
  homepage = "http://www.cis.upenn.edu/~bcpierce/unison/";
  ...
}
```

Running GODIVA on the Unison package works exactly as it did for SpamOracle.

4.3 OcamlConf

We next explore the installation of a library package, OcamlConf, an `autoconf`-like utility written especially for O'Caml projects [5].

4.3.1 Routine Work

A small patch is necessary to fix an incompatibility with OcamlConf under GODI. As before, we elide the details to simplify things — there's nothing here we haven't seen before.

4.3.2 What's New

OcamlConf is a library, so it requires an extra dependency — findlib. Furthermore, it uses a `configure` script that takes a prefix argument.

4.3.3 The Solution

This package doesn't make use of any experimental features, so we can use a normal human-readable spec file. Here it is:

```
Package: godi-ocamlconf
Version: 0.3
Revision: 0
Depends: godi-ocaml (>= 3.06), godi-findlib
Build-Depends:
Sources: http://kenn.frap.net/ocamlconf/ocamlconf-0.3.tar.bz2
Homepage: http://kenn.frap.net/ocamlconf/
Maintainer: William Lovas <wlovas@stwing.upenn.edu>
Options: configure[prefix_option="--prefix "]
Docfiles: README, COPYING
Description: A simple O'Caml build tool
[... long description elided ...]
.
```

The interesting points are the `Depends:` line and the `Options:` line.

When we run GODIVA on this spec file, it produces a GODI package that installs flawlessly.

What's so different about this as compared to our previous packaging case studies? Not much, from the Godiva user's perspective! But under the hood, the generated `Makefile` has several extra variable definitions, and Godi has been made aware of all the findlib-installed library files thanks to the automatic `PLIST` generation discussed in Section 3.1. All these things are taken care of without the packager ever having to be aware of them.

4.4 Godiva

4.4.1 Self Hosting

Of course, Godiva can be used to generate a package for itself. Because we created Godiva from the ground up, it was easy to make it adhere strictly to the Godiva policy. Thus packaging Godiva is extremely simple — no patches and no camlsyntax features are required. This demonstrates that if a new piece of code is written and the author expends a near-negligible amount of energy minding the policy along the way, packaging is trivial.

Godiva's primary mode of distribution is as a Godi package. This is a bold step that sets a good example for other new projects, and further encourages the adoption of Godi.

4.4.2 Updates

Since we frequently release new versions of Godiva as Godi packages, we have to update our old versions. To facilitate this, Godiva leaves a copy of the original `.godiva` specification file in every outputted Godi package. Updating is often as simple as copying the old specification and changing the version number, then re-running Godiva.

5 Related Work

Our work builds on Godi [12], the emerging standard for O'Caml source code distribution. Godi is a source-based package management system, like NetBSD pkgsrc [10] and the Gentoo Linux distribution's Portage [7] among others. Both pkgsrc and Portage are highly flexible package management systems used on a large corpus of packages. Godi borrows ideas from both.

Well-tested and widely-used reusable code distribution frameworks exist for the Perl programming language and the $\TeX$ typesetting language; they are CPAN [2] and CTAN [17], respectively. Many of the techniques underlying their implementations are specific to their respective target languages, but their relative ease-of-use inspired much of the philosophy underlying Godiva. Furthermore, their popularity has given us an provably attainable goal to aim for in our work.

There are a variety of O'Caml projects that help standardize compilation and distribution of source code. These include OcamlConf [5], a tool for writing Makefile-generating configure scripts in O'Caml; OCamlMakefile [8], an `include`-able Makefile that vastly simplifies the Makefile for many types of O'Caml projects; OCamake [1], an "automatic compiler" for O'Caml that unifies `ocamldep` (the standard ocaml module dependency analyzer) with the compilers, `ocamlc` and `ocamlopt`; Findlib [11], another tool by Gerd Stolpmann to standardize the installation of O'Caml libraries; and the Caml Humps [16], a searchable index of many O'Caml projects available freely on the internet. OcamlConf, OCamlMakefile, and OCamake provided significant inspiration and guidance while we were delineating the Godiva build policy. Findlib is an important underlying component of Godi that solves the problem of *using* a library once it has been installed. The Caml Humps site contains myriad O'Caml applications and libraries ripe to become test cases for Godiva — all of our case studies came from the Humps.

The Debian O'Caml Maintainers Task Force [3] is a group of people who have maintained high quality Debian Linux packages for many O'Caml-related programs. Their experience in package maintenance has been an invaluable resource for understanding the intricacies of the problem.

6 Future work

While Godiva greatly simplifies the creation of packages in its current incarnation, it might be improved in a few ways.

- Godiva has experimental support for conf opts, but this functionality is accessible only through the `-camlsyntax` hook into the backend. As soon as the backend design of conf opts is sufficiently stabilized and well-understood, we will design a concrete spec file syntax for them and implement lexing and parsing for this syntax in the frontend.
- The O'Caml compiler suite includes both a bytecode compiler and runtime, and an optimizing native code compiler. Some packages make an effort to allow the user to select which compiler is used, but the interface for choosing is not standard and varies widely. Some even let the user install both bytecode and native versions of the package. Currently, Godiva plays it safe and compiles with whichever compiler is default in the package's makefiles.
- The policy of requiring `PREFIX` to be used for installation is not enforced. One idea which has been discussed in the Godi community involves wrapping all obvious filesystem modifying utilities to catch any modifications that are made to files with roots other than `PREFIX`. While this is not completely safe, it should catch the vast majority of cases. It may be possible to implement an airtight safety mechanism as well, but it is currently unclear how such a mechanism would function.
- Godiva has only been tested on x86 hardware running Linux. Since it is 100% O'Caml, it should run on other platforms with no problems. Testing on other platforms is nonetheless important.
- Godi's current model for adding packages to the official distribution is to upload via subversion and modify a few registration files. This could be simplified in a few minor ways.

Perhaps the most important work to do with Godiva is simply to use it. There are hundreds of O'Caml projects available on the Caml Humps [16] alone. At a rate of just a few minutes per package, attainable only with the help of Godiva, all of these packages could be added to Godi in a few short hours.

Acknowledgements

We thank E Lewis, our advisor who never let us lose excitement. We also appreciate the help and encouragement we got from Gerd Stolpmann, creator of Godi. Special thanks go to the ASCII character 2, for being such a good sport in Figure 1. Finally, we are indebted to Benjamin Pierce for the idea to name our tool "Godiva," although the acronym (including *Verpacken*, German for the verb "to pack" or "to package") was our own creation.

References

[1] Nicholas Cannasse. OCamake. `http://tech.motion-twin.com/html/ocamake.html`, 2002–2003.

[2] CPAN. `http://www.cpan.org/`.

[3] Debian OCaml Maintainers Task Force. `http://pkg-ocaml-maint.alioth.debian.org/`, February 2004.

[4] Valery A. Khamenya. OCaml mail list activity. `http://khamenya.ru/ocaml/activity/`, September 2003.

[5] Kenn Knowles. OcamlConf: A simple Ocaml build tool. `http://kenn.frap.net/ocamlconf/`, 2004.

[6] Xavier Leroy. SpamOracle: detection of spam by statistical analysis of e-mail contents. `http://pauillac.inria.fr/~xleroy/software.html#spamoracle`.

[7] Bruce A. Lock, Carl Anderson, Sven Vermeulen, and Jorge Paulo. Gentoo Linux Documentation – Portage Manual. `http://www.gentoo.org/doc/en/portage-manual.xml`, September 2003.

[8] Markus Mottl. OCamlMakefile. `http://www.ai.univie.ac.at/~markus/home/ocaml_sources.html#OCamlMakefil%e`, 1999–2003.

[9] Benjamin C. Pierce. Unison File Synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison/index.html`.

[10] pkgsrc: The NetBSD Packages Collection. `http://www.netbsd.org/Documentation/software/packages.html`.

[11] Gerd Stolpmann. Findlib. `http://www.ocaml-programming.de/programming/findlib.html`.

[12] Gerd Stolpmann. GODI. `http://www.ocaml-programming.de/programming/godi.html`.

[13] Gerd Stolpmann. Re: [Caml-list] GODI (was: CTAN/CPAN for Caml (COCAN ...?)). Caml-list message, archived at `http://caml.inria.fr/archives/200307/msg00193.html`.

[14] Gerd Stolpmann. Personal communication, October 2003.

[15] subversion Project home. `http://subversion.tigris.org/`.

[16] The Caml Humps. `http://caml.inria.fr/humps/`.

[17] Welcome to ctan.org. `http://www.ctan.org/`.

A Appendices

A.1 Build policy

1. *Distribution.* Source code must come in a compressed (gzip or bzip2) tar archive. The archive should [1] have a name like `package-version.tar.gz` or `package-version.tar.bz2`, and it should unpack into a directory called `package-version`.

2. *Configuration.* If a package requires a configuration step, it must be performed by an executable called `configure`.

3. *Compilation.* A package must have one or more 'make' targets for building software and documentation. These should have the following names[1]:

(required)	`make all`	builds bytecode versions of the software.
(optional)	`make opt`	builds native code versions of the software.
(optional)	`make htdoc`	builds html documentation for the code and places it in `doc/html`.

   Compilation must work with either GNU make or BSD make, at the developer's option. If compilation requires knowledge of the final location of the software or documentation, it should consult the `$(PREFIX)` variable.

4. *Installation.* A package must have one `make` target for installation:

(required)	`make install`	installs files into `$(PREFIX)`, and/or installs libraries using findlib. Libraries must be installed with findlib (an O'Caml tool for managing libraries and linking).

   Note that this target must respect the `$(PREFIX)` setting, even in the case that it differs from the `$(PREFIX)` setting from a previous `make` invocation.

   File locations should follow these rules:

   (a) Binaries should be installed in `$(PREFIX)/bin`.

   (b) Man pages should be installed in `$(PREFIX)/man/mman`*n*. Man pages must not be compressed.

   (c) Info pages should be installed in `$(PREFIX)/info`.

   (d) HTML documentation should be installed in `$(PREFIX)/doc/<pkgname>/html`.

   (e) Other documentation should be installed in `$(PREFIX)/doc/<pkgname>`.

   (f) Other files related to the package should be installed in `$(PREFIX)/share/<pkgname>`.

A.2 Sample GODI Makefile specification

The following is the Godi Makefile specification for the xml-light package corresponding to the specification for our package creation tool shown in Figure 3. (**Note:** the Makefile does not include the long description of the package — in Godi, this is stored in a separate file.)

```
.include "../../mk/bsd.prefs.mk"
.include "../../mk/godi.pkg.mk"

VERSION=        2.01
PKGNAME=        godi-xml-light-${VERSION}
#PKGREVISION=        XXX
DISTNAME=       xml-light-${VERSION}
DISTFILES=      xml-light-${VERSION}.tar.gz
```

[1]With versions 0.02 and earlier of Godiva, this is a "must" rule.

```
CATEGORIES=     godi # godi, apps, or conf
MASTER_SITES=   http://www.phauna.org/~ogunden/code/
MAINTAINER=     ogunden@phauna.org
HOMEPAGE=       http://tech.motion-twin.com/?&node=28
COMMENT=        A minimal XML parser & printer for OCaml.

AUTOGENERATE_PLIST = yes
PKG := ${PKGNAME:S/-${VERSION}//}

MAKE_FLAGS=     PREFIX=${PREFIX}

DEPENDS+=       godi-ocaml>=3.06:../../godi/godi-ocaml
BUILD_DEPENDS+= godi-findlib>=0.8.1:../../godi/godi-findlib

# Adjust PATH such that version of godi is prepended.  This has
# only an effect on commands where PATH is explicitly passed (e.g.
# configure).
PATH:=          ${LOCALBASE}/bin:${PATH}

#HAS_CONFIGURE = yes
# If commented out, the configure script will not be called

USE_GMAKE=      yes
# If commented out, bmake will be used for compilation

CONFIGURE_ENV+= ${BUILD_OCAMLFIND_ENV}
MAKE_ENV+=      ${BUILD_OCAMLFIND_ENV}

pre-configure:
.        if exists(files)
           cp files/* ${WRKSRC}
.        endif

pre-install-mkdirs:
.        for d in bin lib/ocaml/pkg-lib doc share man etc info sbin include
           ${_PKG_SILENT}${_PKG_DEBUG}mkdir -p ${PREFIX}/${d}
.        endfor
.        for n in 1 2 3 4 5 6 7 8 9
           ${_PKG_SILENT}${_PKG_DEBUG}mkdir -p ${PREFIX}/man/man${n}
.        endfor

ALL_TARGET=     all
.if ${GODI_HAVE_OCAMLOPT} == "yes"
ALL_TARGET+=    opt
.endif

pre-install: pre-install-mkdirs

post-install:
        mkdir -p ${PREFIX}/doc/${PKG}
.        for DOC in README
```

```
                install ${WRKSRC}/${DOC} ${PREFIX}/doc/${PKG}
.       endfor

.include "../../mk/bsd.pkg.mk"
```