(ELS '13)

Proceedings of ELS 2013

6th European Lisp Symposium

June 3 – 4 2013
Madrid, Spain

EPITA
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

clozure
associates

inria
informatiques mathématiques

Franz Inc.
"Web 3.0's Database"

LispWorks LW

ALU.ORG

Organization

Programme Chairs

- Christian Queinnec, UPMC, France
- Manuel Serrano, INRIA, France

Local Chair

- Juan José Garcia-Ripoll, IFF, Madrid

Programme Committee

- Pascal Costanza, Intel, Belgium
- Ludovic Courtès, Inria, France
- Theo D'Hondt, Vrije Universiteit Brussel, Belgium
- Erick Gallesio, University of Nice-Sophia Antipolis
- Florian Loitsch, Google, Denmark
- Kurt Noermark, Aalborg University, Denmark
- Christian Queinnec, UPMC, France
- Olin Shivers, Northeastern University, USA
- Manuel Serrano, Inria, France
- Didier Verna, Epita Research Lab, France

Sponsors

EPITA
14-16 rue Voltaire
FR-94276 Le Kremlin-Bicêtre CEDEX
France
`www.epita.fr`

LispWorks Ltd.
St John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England
`www.lispworks.com`

Franz Inc.
2201 Broadway, Suite 715
Oakland, CA 94612
`www.franz.com`

Clozure Associates
Boston, MA 02205-5071
USA
`www.clozure.com`

INRIA
Domaine de Voluceau
Rocquencourt - BP 105
78153 Le Chesnay Cedex
France
`www.inria.fr`

Association of Lisp Users
USA
`www.alu.org`

Contents

Invited Talks

Asynchronous Programming in Dart

Florian Loitsch, Google

Florian Loitsch has a passion for dynamic languages, like Scheme, JavaScript and now Dart. He wrote a Scheme-to-JavaScript compiler during his thesis, and then completed a JavaScript-to-Scheme compiler in his spare time.
In 2010 he joined Google's team in Aarhus (Denmark) where he worked on V8 and later Dart. Being part of the Dart team Florian has helped specifying the language, which was presented in late 2011. In 2012 he became the Tech Lead for the Dart libraries where, among other tasks, he participated on the creation of a new asynchronous library that is based on Streams and Futures (aka promises).

Lisp and Music Research

Gérard Assayag, IRCAM

Lisp has long been and still is a privileged language for building "experiments in musical intelligence", to quote the title of a book by David Cope, a famous researcher in the field. Thus its use in several "Computer Assisted Composition" systems or more generally for environments oriented towards the modelling, representation, analysis and generation of music. Although the choice of Lisp has been generally reserved to high level, symbolic and formal representations, it is also possible to benefit from this powerful functional language paradigm and from its characteristic data / program duality (think of the musical duality between structures and processes) in complex setups, where the whole range of representations and time scales is invoked from the acoustic signal to the formal organization. Some interesting past and present systems will be presented in this perspective, including OpenMusic and OMax, designed by the author with his team at IRCAM.
Gerard Assayag is the founder of the Music Representations Team at IRCAM, where he has designed with his collaborators OpenMusic and OMax, two lisp based environments which have become international standards for computer assisted music composition / analysis and for music improvisation. He is head of the IRCAM STMS research Lab since Jan 2011. IRCAM is the biggest joint facility for music research and production in the world, where many leading technologies and software have been created.

Streams-Based, Multi-Threaded News Classification

Jason Cornez, RavenPack

Streams are a way of organizing indefinite collections of data such that each item can naturally flow through a network of computations. Using some simple abstractions, we construct a computation network that operates on streams with each node being handled by a separate computation thread. The result is efficient, maintainable and all done in Common Lisp.

The financial industry is hungry to trade on news, but often ill-equipped to do so. The continuous publication of news is like big data in real time. RavenPack processes this flow and produces actionable News Analytics for any industry with an appetite for news. Jason Cornez joined RavenPack in 2003 to solve real-world problems using Common Lisp.

Session I

Functional Package Management with Guix

Ludovic Courtès
Bordeaux, France
ludo@gnu.org

ABSTRACT

We describe the design and implementation of GNU Guix, a purely functional package manager designed to support a complete GNU/Linux distribution. Guix supports transactional upgrades and roll-backs, unprivileged package management, per-user profiles, and garbage collection. It builds upon the low-level build and deployment layer of the Nix package manager. Guix uses Scheme as its programming interface. In particular, we devise an embedded domain-specific language (EDSL) to describe and compose packages. We demonstrate how it allows us to benefit from the host general-purpose programming language while not compromising on expressiveness. Second, we show the use of Scheme to write build programs, leading to a "two-tier" programming system.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.4.5 [**Operating Systems**]: System Programs and Utilities; D.1.1 [**Software**]: Applicative (Functional) Programming

General Terms

Languages, Management, Reliability

Keywords

Functional package management, Scheme, Embedded domain-specific language

1. INTRODUCTION

GNU Guix[1] is a *purely functional* package manager for the GNU system [20], and in particular GNU/Linux. Package management consists in all the activities that relate to building packages from source, honoring the build-time and run-time dependencies on packages, installing, removing, and upgrading packages in user environments. In addition to these standard features, Guix supports transactional upgrades and roll-backs, unprivileged package management, per-user profiles, and garbage collection. Guix comes with a distribution of user-land free software packages.

Guix seeks to empower users in several ways: by offering the uncommon features listed above, by providing the tools that allow users to formally correlate a binary package and the "recipes" and source code that led to it—furthering the spirit of the GNU General Public License—, by allowing them to customize the distribution, and by lowering the barrier to entry in distribution development.

The keys toward these goals are the implementation of a purely functional package management paradigm, and the use of both declarative and lower-level programming interfaces (APIs) embedded in Scheme. To that end, Guix reuses the package storage and deployment model implemented by the Nix functional package manager [8]. On top of that, it provides Scheme APIs, and in particular embedded domain-specific languages (EDSLs) to describe software packages and their build system. Guix also uses Scheme for programs and libraries that implement the actual package build processes, leading to a "two-tier" system.

This paper focuses on the programming techniques implemented by Guix. Our contribution is twofold: we demonstrate that use of Scheme and EDSLs achieves expressiveness comparable to that of Nix's DSL while providing a richer and extensible programming environment; we further show that Scheme is a profitable alternative to shell tools when it comes to package build programs. Section 2 first gives some background on functional package management and its implementation in Nix. Section 3 describes the design and implementation of Guix's programming and packaging interfaces. Section 4 provides an evaluation and discussion of the current status of Guix. Section 5 presents related work, and Section 6 concludes.

[1]`http://www.gnu.org/software/guix/`

2. BACKGROUND

This section describes the functional package management paradigm and its implementation in Nix. It then shows how Guix differs, and what the rationale is.

2.1 Functional Package Management

Functional package management is a paradigm whereby the build and installation process of a package is considered as a pure function, without any side effects. This is in contrast with widespread approaches to package build and installation where the build process usually has access to all the software installed on the machine, regardless of what its declared inputs are, and where installation modifies files in place.

Functional package management was pioneered by the Nix package manager [8], which has since matured to the point of managing a complete GNU/Linux distribution [9]. To allow build processes to be faithfully regarded as pure functions, Nix can run them in a `chroot` environment that only contains the inputs it explicitly declared; thus, it becomes impossible for a build process to use, say, Perl, if that package was not explicitly declared as an input of the build process. In addition, Nix maps the list of inputs of a build process to a statistically unique file system name; that file name is used to identify the output of the build process. For instance, a particular build of GNU Emacs may be installed in `/nix/store/-v9zic07iar8w90zcy398r745w78a7lqs-emacs-24.2`, based on a cryptographic hash of all the inputs to that build process; changing the compiler, configuration options, build scripts, or any other inputs to the build process of Emacs yields a different name. This is a form of *on-disk memoization*, with the `/nix/store` directory acting as a cache of "function results"—i.e., a cache of installed packages. Directories under `/nix/store` are immutable.

This direct mapping from build inputs to the result's directory name is basis of the most important properties of a functional package manager. It means that build processes are regarded as *referentially transparent*. To put it differently, instead of merely providing pre-built binaries and/or build recipes, functional package managers provide binaries, build recipes, and in effect a *guarantee* that a given binary matches a given build recipe.

2.2 Nix

The idea of *purely functional* package started by making an analogy between programming language paradigms and software deployment techniques [8]. The authors observed that, in essence, package management tools typically used on free operating systems, such as RPM and Debian's APT, implement an *imperative* software deployment paradigm. Package installation, removal, and upgrade are all done in-place, by mutating the operating system's state. Likewise, changes to the operating system's configuration are done in-place by changing configuration files.

This imperative approach has several drawbacks. First, it makes it hard to reproduce or otherwise describe the OS state. Knowing the list of installed packages and their version is not enough, because the installation procedure of packages may trigger hooks to change global system configuration files [4, 7], and of course users may have done additional modifications. Second, installation, removal, and upgrade are not transactional; interrupting them may leave the system in an undefined, or even unusable state, where some of the files have been altered. Third, rolling back to a previous system configuration is practically impossible, due to the absence of a mechanism to formally describe the system's configuration.

Nix attempts to address these shortcomings through the functional software deployment paradigm: installed packages are immutable, and build processes are regarded as pure functions, as explained before. Thanks to this property, it implements *transparent source/binary deployment*: the directory name of a build result encodes all the inputs of its build process, so if a trusted server provides that directory, then it can be directly downloaded from there, avoiding the need for a local build.

Each user has their own *profile*, which contains symbolic links to the `/nix/store` directories of installed packages. Thus, users can install packages independently, and the actual storage is shared when several users install the very same package in their profile. Nix comes with a *garbage collector*, which has two main functions: with conservative scanning, it can determine what packages a build output refers to; and upon user request, it can delete any packages not referenced *via* any user profile.

To describe and compose build processes, Nix implements its own domain-specific language (DSL), which provides a convenient interface to the build and storage mechanisms described above. The Nix language is purely functional, lazy, and dynamically typed; it is similar to that of the Vesta software configuration system [11]. It comes with a handful of built-in data types, and around 50 primitives. The primitive to describe a build process is `derivation`.

```
derivation {
  name = "example-1.0";
  builder = "${./static-bash}";
  args = [ "-c" "echo hello > $out" ];
  system = "x86_64-linux";
}
```

Figure 1: Call to the `derivation` primitive in the Nix language.

Figure 1 shows code that calls the `derivation` function with one argument, which is a dictionary. It expects at least the four key/value pairs shown above; together, they define the build process and its inputs. The result is a *derivation*, which is essentially the *promise* of a build. The derivation has a low-level on-disk representation independent of the Nix language—in other words, derivations are to the Nix language what assembly is to higher-level programming languages. When this derivation is instantiated—i.e., built—, it runs the command `static-bash -c "echo hello > $out"` in a chroot that contains nothing but the `static-bash` file; in addition, each key/value pair of the `derivation` argument is reified in the build process as an environment variable, and the `out` environment variable is defined to point to the output `/nix/store` file name.

Before the build starts, the file `static-bash` is imported under `/nix/store/...-static-bash`, and the value associated with `builder` is substituted with that file name. This `${...}` form on line 3 for string interpolation makes it easy to insert Nix-language values, and in particular computed file names, in the contents of build scripts. The Nix-based GNU/Linux distribution, NixOS, has most of its build scripts written in Bash, and makes heavy use of string interpolation on the Nix-language side.

All the files referenced by derivations live under `/nix/store`, called *the store*. In a multi-user setup, users have read-only access to the store, and all other accesses to the store are mediated by a daemon running as `root`. Operations such as importing files in the store, computing a derivation, building a derivation, or running the garbage collector are all implemented as remote procedure calls (RPCs) to the daemon. This guarantees that the store is kept in a consistent state—e.g., that referenced files and directories are not garbage-collected, and that the contents of files and directories are genuine build results of the inputs hashed in their name.

The implementation of the Nix language is an interpreter written in C++. In terms of performance, it does not compete with typical general-purpose language implementations; that is often not a problem given its specific use case, but sometimes requires rewriting functions, such as list-processing tools, as language primitives in C++. The language itself is not extensible: it has no macros, a fixed set of data types, and no foreign function interface.

2.3 From Nix to Guix

Our main contribution with GNU Guix is the use of Scheme for both the composition and description of build processes, and the implementation of build scripts. In other words, Guix builds upon the build and deployment primitives of Nix, but replaces the Nix language by Scheme with embedded domain-specific languages (EDSLs), and promotes Scheme as a replacement for Bash in build scripts. Guix is implemented using GNU Guile 2.0[2], a rich implementation of Scheme based on a compiler and bytecode interpreter that supports the R5RS and R6RS standards. It reuses the build primitives of Nix by making remote procedure calls (RPCs) to the Nix build daemon.

We claim that using an *embedded* DSL has numerous practical benefits over an independent DSL: tooling (use of Guile's compiler, debugger, and REPL, Unicode support, etc.), libraries (SRFIs, internationalization support, etc.), and seamless integration in larger programs. To illustrate this last point, consider an application that traverses the list of available packages and processes it—for instance to filter packages whose name matches a pattern, or to render it as HTML. A Scheme program can readily and efficiently do it with Guix, where packages are first-class Scheme objects; conversely, writing such an implementation with an external DSL such as Nix requires either extending the language implementation with the necessary functionality, or interfacing with it *via* an external representation such as XML, which is often inefficient and lossy.

We show that use of Scheme in build scripts is natural, and can achieve conciseness comparable to that of shell scripts, but with improved expressivity and clearer semantics.

[2]`http://www.gnu.org/software/guile/`

The next section describes the main programming interfaces of Guix, with a focus on its high-level package description language and "shell programming" substitutes provided to builder-side code.

3. BUILD EXPRESSIONS AND PACKAGE DESCRIPTIONS

Our goal when designing Guix was to provide interfaces ranging from Nix's low-level primitives such as `derivation` to high-level package declarations. The declarative interface is a requirement to help grow and maintain a large software distribution. This section describes the three level of abstractions implemented in Guix, and illustrates how Scheme's homoiconicity and extensibility were instrumental.

3.1 Low-Level Store Operations

As seen above, *derivations* are the central concept in Nix. A derivation bundles together a *builder* and its execution environment: command-line arguments, environment variable definitions, as well as a list of input derivations whose result should be accessible to the builder. Builders are typically executed in a `chroot` environment where only those inputs explicitly listed are visible. Guix transposes Nix's `derivation` primitive literally to its Scheme interface.

```
 1: (let* ((store (open-connection))
 2:        (bash  (add-to-store store "static-bash"
 3:                             #t "sha256"
 4:                             "./static-bash")))
 5:   (derivation store "example-1.0"
 6:               "x86_64-linux"
 7:               bash
 8:               '("-c" "echo hello > $out")
 9:               '() '()))
10:
11: ⇒
12: "/nix/store/nsswy...-example-1.0.drv"
13: #<derivation "example-1.0" ...>
```

Figure 2: Using the `derivation` primitive in Scheme with Guix.

Figure 2 shows the example of Figure 1 rewritten to use Guix's low-level Scheme API. Notice how the former makes explicit several operations not visible in the latter. First, line 1 establishes a connection to the build daemon; line 2 explicitly asks the daemon to "intern" file `static-bash` into the store; finally, the `derivation` call instructs the daemon to compute the given derivation. The two arguments on line 9 are a set of environment variable definitions to be set in the build environment (here, it's just the empty list), and a set of *inputs*—other derivations depended on, and whose result must be available to the build process. Two values are returned (line 11): the file name of the on-disk representation of the derivation, and its in-memory representation as a Scheme record.

The build actions represented by this derivation can then be performed by passing it to the `build-derivations` RPC.

Again, its build result is a single file reading `hello`, and its build is performed in an environment where the only visible file is a copy of `static-bash` under `/nix/store`.

3.2 Build Expressions

The Nix language heavily relies on string interpolation to allow users to insert references to build results, while hiding the underlying `add-to-store` or `build-derivations` operations that appear explicitly in Figure 2. Scheme has no support for string interpolation; adding it to the underlying Scheme implementation is certainly feasible, but it's also unnatural.

The obvious strategy here is to instead leverage Scheme's homoiconicity. This leads us to the definition of `build-expression->derivation`, which works similarly to `derivation`, except that it expects a *build expression* as an S-expression instead of a builder. Figure 3 shows the same derivation as before but rewritten to use this new interface.

```
 1: (let ((store   (open-connection))
 2:       (builder '(call-with-output-file %output
 3:                   (lambda ()
 4:                     (display "hello")))))
 5:   (build-expression->derivation store
 6:                                 "example-1.0"
 7:                                 "x86_64-linux"
 8:                                 builder '()))
 9:
10: ⇒
11: "/nix/store/zv3b3...-example-1.0.drv"
12: #<derivation "example-1.0" ...>
```

Figure 3: Build expression written in Scheme.

This time the builder on line 2 is purely a Scheme expression. That expression will be evaluated when the derivation is built, in the specified build environment with no inputs. The environment implicitly includes a copy of Guile, which is used to evaluate the `builder` expression. By default this is a stand-alone, statically-linked Guile, but users can also specify a derivation denoting a different Guile variant.

Remember that this expression is run by a separate Guile process than the one that calls `build-expression->derivation`: it is run by a Guile process launched by the build daemon, in a `chroot`. So, while there is a single language for both the "host" and the "build" side, there are really two *strata* of code, or *tiers*: the host-side, and the build-side code[3].

Notice how the output file name is reified *via* the `%output` variable automatically added to `builder`'s scope. Input file names are similarly reified through the `%build-inputs` variable (not shown here). Both variables are non-hygienically introduced in the build expression by `build-expression->derivation`.

Sometimes the build expression needs to use functionality from other modules. For modules that come with Guile, the expression just needs to be augmented with the needed `(use-modules ...)` clause. Conversely, external modules first need to be imported into the derivation's build environment so the build expression can use them. To that end, the `build-expression->derivation` procedure has an optional `#:modules` keyword parameter, allowing additional modules to be imported into the expression's environment.

When `#:modules` specifies a non-empty module list, an auxiliary derivation is created and added as an input to the initial derivation. That auxiliary derivation copies the module source and compiled files in the store. This mechanism allows build expressions to easily use helper modules, as described in Section 3.4.

3.3 Package Declarations

The interfaces described above remain fairly low-level. In particular, they explicitly manipulate the store, pass around the system type, and are very distant from the abstract notion of a software package that we want to focus on. To address this, Guix provides a high-level package definition interface. It is designed to be *purely declarative* in common cases, while allowing users to customize the underlying build process. That way, it should be intelligible and directly usable by packagers will little or no experience with Scheme. As an additional constraint, this extra layer should be efficient in space and time: package management tools need to be able to load and traverse a distribution consisting of thousands of packages.

Figure 4 shows the definition of the GNU Hello package, a typical GNU package written in C and using the GNU build system—i.e., a `configure` script that generates a makefile supporting standardized targets such as `check` and `install`. It is a direct mapping of the abstract notion of a software package and should be rather self-descriptive.

The `inputs` field specifies additional dependencies of the package. Here line 16 means that Hello has a dependency labeled `"gawk"` on GNU Awk, whose value is that of the `gawk` global variable; `gawk` is bound to a similar `package` declaration, omitted for conciseness.

The `arguments` field specifies arguments to be passed to the build system. Here `#:configure-flags`, unsurprisingly, specifies flags for the `configure` script. Its value is quoted because it will be evaluated in the build stratum—i.e., in the build process, when the derivation is built. It refers to the `%build-inputs` global variable introduced in the build stratum by `build-expression->derivation`, as seen before. That variable is bound to an association list that maps input names, like `"gawk"`, to their actual directory name on disk, like `/nix/store/...-gawk-4.0.2`.

The code in Figure 4 demonstrates Guix's use of embedded domain-specific languages (EDSLs). The `package` form, the `origin` form (line 5), and the `base32` form (line 9) are expanded at macro-expansion time. The `package` and `origin` forms expand to a call to Guile's `make-struct` primitive, which instantiates a record of the given type and with the given field values[4]; these macros look up the mapping of

[3]The term "stratum" is this context was coined by Manuel Serrano et al. for their work on Hop where a similar situation arises [17].

[4]The `make-struct` instantiates SRFI-9-style flat records,

```
 1: (define hello
 2:   (package
 3:     (name "hello")
 4:     (version "2.8")
 5:     (source (origin
 6:               (method url-fetch)
 7:               (uri (string-append "mirror://gnu/hello/hello-"
 8:                                   version ".tar.gz"))
 9:               (sha256 (base32 "0wqd8..."))))
10:     (build-system gnu-build-system)
11:     (arguments
12:       '(#:configure-flags
13:         `("-disable-color"
14:           ,(string-append "-with-gawk="
15:                           (assoc-ref %build-inputs "gawk")))))
16:     (inputs `(("gawk" ,gawk)))
17:     (synopsis "GNU Hello")
18:     (description "An illustration of GNU's engineering practices.")
19:     (home-page "http://www.gnu.org/software/hello/")
20:     (license gpl3+)))
```

Figure 4: A package definition using the high-level interface.

field names to field indexes, such that that mapping incurs no run-time overhead, in a way similar to SRFI-35 records [14]. They also bind fields as per `letrec*`, allowing them to refer to one another, as on line 8 of Figure 4. The `base32` macro simply converts a literal string containing a base-32 representation into a bytevector literal, again allowing the conversion and error-checking to be done at expansion time rather than at run-time.

```
(define-record-type* <package>
  package make-package
  package?

  (name package-name)
  (version package-version)
  (source package-source)
  (build-system package-build-system)
  (arguments package-arguments
             (default '()) (thunked))

  (inputs package-inputs
          (default '()) (thunked))
  (propagated-inputs package-propagated-inputs
                     (default '()))

  (synopsis package-synopsis)
  (description package-description)
  (license package-license)
  (home-page package-home-page)

  (location package-location
    (default (current-source-location))))
```

Figure 5: Definition of the `package` record type.

which are essentially vectors of a disjoint type. In Guile they are lightweight compared to CLOS-style objects, both in terms of run time and memory footprint. Furthermore, `make-struct` is subject to inlining.

The `package` and `origin` macros are generated by a `syntax-case` hygienic macro [19], `define-record-type*`, which is layered above SRFI-9's syntactic record layer [13]. Figure 5 shows the definition of the `<package>` record type (the `<origin>` record type, not shown here, is defined similarly.) In addition to the name of a procedural constructor, `make-package`, as with SRFI-9, the name of a *syntactic* constructor, `package`, is given (likewise, `origin` is the syntactic constructor of `<origin>`.) Fields may have a default value, introduced with the `default` keyword. An interesting use of default values is the `location` field: its default value is the result of `current-source-location`, which is itself a built-in macro that expands to the source file location of the `package` form. Thus, records defined with the `package` macro automatically have a `location` field denoting their source file location. This allows the user interface to report source file location in error messages and in package search results, thereby making it easier for users to "jump into" the distribution's source, which is one of our goals.

```
(package (inherit hello)
  (version "2.7")
  (source
    (origin
      (method url-fetch)
      (uri
       "mirror://gnu/hello/hello-2.7.tar.gz")
      (sha256
        (base32 "7dqw3...")))))
```

Figure 6: Creating a variant of the `hello` package.

The syntactic constructors generated by `define-record-type*` additionally support a form of *functional setters* (sometimes referred to as "lenses" [15]), *via* the `inherit` keyword. It allows programmers to create new instances that differ from an existing instance by one or more field values. A typical use case is shown in Figure 6: the expression shown evaluates to a new `<package>` instance whose fields all have the same value as the hello variable of Figure 4, except for the version and source fields. Under the hood, again, this expands to a single make-struct call with struct-ref calls for fields whose value is reused.

The inherit feature supports a very useful idiom. It allows new package variants to be created programmatically, concisely, and in a purely functional way. It is notably used to bootstrap the software distribution, where bootstrap variants of packages such as GCC or the GNU libc are built with different inputs and configuration flags than the final versions. Users can similarly define customized variants of the packages found in the distribution. This feature also allows high-level transformations to be implemented as pure functions. For instance, the static-package procedure takes a <package> instance, and returns a variant of that package that is statically linked. It operates by just adding the relevant configure flags, and recursively applying itself to the package's inputs.

Another application is the *on-line auto-updater*: when installing a GNU package defined in the distribution, the guix package command automatically checks whether a newer version is available upstream from ftp.gnu.org, and offers the option to substitute the package's source with a fresh download of the new upstream version—all at run time.This kind of feature is hardly accessible to an external DSL implementation. Among other things, this feature requires networking primitives (for the FTP client), which are typically unavailable in an external DSL such as the Nix language. The feature could be implemented in a language other than the DSL—for instance, Nix can export its abstract syntax tree as XML to external programs. However, this approach is often inefficient, due to the format conversion, and lossy: the exported representation may be either be too distant from the source code, or too distant from the preferred abstraction level. The author's own experience writing an off-line auto-updater for Nix revealed other specific issues; for instance, the Nix language is lazily evaluated, but to make use of its XML output, one has to force strict evaluation, which in turn may generate more data than needed. In Guix, <package> instances have the expected level of abstraction, and they are readily accessible as first-class Scheme objects.

Sometimes it is desirable for the value of a field to depend on the system type targeted. For instance, for bootstrapping purposes, MIT/GNU Scheme's build system depends on pre-compiled binaries, which are architecture-dependent; its input field must be able to select the right binaries depending on the architecture. To allow field values to refer to the target system type, we resort to *thunked* fields, as shown on line 13 of Figure 5. These fields have their value automatically wrapped in a thunk (a zero-argument procedure); when accessing them with the associated accessor, the thunk is transparently invoked. Thus, the values of thunked fields are computed lazily; more to the point, they can refer to *dynamic state* in place at their invocation point. In particular, the package-derivation procedure (shortly introduced) sets up a current-system dynamically-scoped parameter, which allows field values to know what the target system is.

Finally, both <package> and <origin> records have an associated "compiler" that turns them into a derivation. origin-derivation takes an <origin> instance and returns a derivation that downloads it, according to its method field. Likewise, package-derivation takes a package and returns a derivation that builds it, according to its build-system and associated arguments (more on that in Section 3.4). As we have seen on Figure 4, the inputs field lists dependencies of a package, which are themselves <package> objects; the package-derivation procedure recursively applies to those inputs, such that their derivation is computed and passed as the inputs argument of the lower-level build-expression->derivation.

Guix essentially implements *deep embedding* of DSLs, where the semantics of the packaging DSL is interpreted by a dedicated compiler [12]. Of course the DSLs defined here are simple, but they illustrate how Scheme's primitive mechanisms, in particular macros, make it easy to implement such DSLs without requiring any special support from the Scheme implementation.

3.4 Build Programs

The value of the build-system field, as shown on Figure 4, must be a build-system object, which is essentially a wrapper around two procedure: one procedure to do a native build, and one to do a cross-build. When the aforementioned package-derivation (or package-cross-derivation, when cross-building) is called, it invokes the build system's build procedure, passing it a connection to the build daemon, the system type, derivation name, and inputs. It is the build system's responsibility to return a derivation that actually builds the software.

```
(define* (gnu-build #:key (phases %standard-phases)
                    #:allow-other-keys
                    #:rest args)
  ;; Run all the PHASES in order, passing them ARGS.
  ;; Return true on success.
  (every (match-lambda
          ((name . proc)
           (format #t "starting phase ‘~a’~%" name)
           (let ((result (apply proc args)))
             (format #t "phase ‘~a’ done~%" name)
             result)))
         phases))
```

Figure 7: Entry point of the builder side code of gnu-build-system.

The gnu-build-system object (line 10 of Figure 4) provides procedures to build and cross-build software that uses the GNU build system or similar. In a nutshell, it runs the following phases by default:

1. unpack the source tarball, and change the current directory to the resulting directory;
2. patch shebangs on installed files—e.g., replace #!/-bin/sh by #!/nix/store/...-bash-4.2/bin/sh; this is required to allow scripts to work with our unusual file system layout;
3. run ./configure --prefix=/nix/store/..., followed by make and make check

4. run `make install` and patch shebangs in installed files.

Of course, that is all implemented in Scheme, *via* `build-expression->derivation`. Supporting code is available as a build-side module that `gnu-build-system` automatically adds as an input to its build scripts. The default build programs just call the procedure of that module that runs the above phases.

The `(guix build gnu-build-system)` module contains the implementation of the above phases; it is imported on the builder side. The phases are modeled as follows: each phase is a procedure accepting several keyword arguments, and ignoring any keyword arguments it does not recognize[5]. For instance the `configure` procedure is in charge of running the package's `./configure` script; that procedure honors the `#:configure-flags` keyword parameter seen on Figure 4. Similarly, the `build`, `check`, and `install` procedures run the `make` command, and all honor the `#:make-flags` keyword parameter.

All the procedures implementing the standard phases of the GNU build system are listed in the `%standard-phases` builder-side variable, in the form of a list of phase name-/procedure pairs. The entry point of the builder-side code of `gnu-build-system` is shown on Figure 7. It calls all the phase procedures in order, by default those listed in the `%standard-phases` association list, passing them all the arguments it got; its return value is true when every procedure's return value is true.

```
(define howdy
  (package (inherit hello)
    (arguments
      '(#:phases
        (alist-cons-after
          'configure 'change-hello
          (lambda* (#:key system #:allow-other-keys)
            (substitute* "src/hello.c"
              (("Hello, world!")
               (string-append "Howdy! Running on "
                              system "."))))
          %standard-phases)))))
```

Figure 8: Package specification with custom build phases.

The `arguments` field, shown on Figure 4, allows users to pass keyword arguments to the builder-side code. In addition to the `#:configure-flags` argument shown on the figure, users may use the `#:phases` argument to specify a different set of phases. The value of the `#:phases` must be a list of phase name/procedure pairs, as discussed above. This allows users to arbitrarily extend or modify the behavior of the build system. Figure 8 shows a variant of the definition in Figure 4 that adds a custom build phase. The `alist-cons-after` procedure is used to add a pair with `change-hello` as its first item and the `lambda*` as its second item right after the pair in `%standard-phases` whose first item is `configure`; in other words, it reuses the standard build phases, but with an additional `change-hello` phase right after the `configure` phase. The whole `alist-cons-after` expression is evaluated on the builder side.

[5]Like many Scheme implementations, Guile supports *named* or *keyword* arguments as an extension to the R5 and R6RS. In addition, procedure definitions whose formal argument list contains the `#:allow-other-keys` keyword ignore any unrecognized keyword arguments that they are passed.

This approach was inspired by that of NixOS, which uses Bash for its build scripts. Even with "advanced" Bash features such as functions, arrays, and associative arrays, the phases mechanism in NixOS remains limited and fragile, often leading to string escaping issues and obscure error reports due to the use of `eval`. Again, using Scheme instead of Bash unsurprisingly allows for better code structuring, and improves flexibility.

Other build systems are provided. For instance, the standard build procedure for Perl packages is slightly different: mainly, the configuration phase consists in running `perl Makefile.-PL`, and test suites are run with `make test` instead of `make check`. To accommodate that, Guix provides `perl-build-system`. Its companion build-side module essentially calls out to that of `gnu-build-system`, only with appropriate `configure` and `check` phases. This mechanism is similarly used for other build systems such as CMake and Python's build system.

```
(substitute* (find-files "gcc/config"
                         "^gnu-user(64)?\\.h$")
  (("#define LIB_SPEC (.*)$" _ suffix)
   (string-append "#define LIB_SPEC \"-L" libc
                  "/lib \" " suffix "\n"))
  (("#define STARTFILE_SPEC.*$" line)
   (string-append "#define STARTFILE_PREFIX_1 \""
                  libc "/lib\"\n" line)))
```

Figure 9: The `substitute*` macro for `sed`-like substitutions.

Build programs often need to traverse file trees, modify files according to a given pattern, etc. One example is the "patch shebang" phase mentioned above: all the source files must be traversed, and those starting with `#!` are candidate to patching. This kind of task is usually associated with "shell programming"—as is the case with the build scripts found in NixOS, which are written in Bash, and resort to `sed`, `find`, etc. In Guix, a build-side Scheme module provides the necessary tools, built on top of Guile's operating system interface. For instance, `find-files` returns a list of files whose names matches a given pattern; `patch-shebang` performs the `#!` adjustment described above; `copy-recursively` and `delete-recursively` are the equivalent, respectively, of the shell `cp -r` and `rm -rf` commands; etc.

An interesting example is the `substitute*` macro, which does `sed`-style substitution on files. Figure 9 illustrates its use to patch a series of files returned by `find-files`. There are two clauses, each with a pattern in the form of a POSIX regular expression; each clause's body returns a string, which is the substitution for any matching line in the given files. In the first clause's body, `suffix` is bound to the submatch corresponding to `(.*)` in the regexp; in the second clause, `line` is bound to the whole match for that regexp. This snippet is nearly as concise than equivalent shell code using find and sed, and it is much easier to work with.

Build-side modules also include support for fetching files over HTTP (using Guile's web client module) and FTP, as needed to realize the derivation of origins (line 5 of Figure 4). TLS support is available when needed through the Guile bindings of the GnuTLS library.

4. EVALUATION AND DISCUSSION

This section discusses the current status of Guix and its associated GNU/Linux distribution, and outlines key aspects of their development.

4.1 Status

Guix is still a young project. Its main features as a package manager are already available. This includes the APIs discussed in Section 3, as well as command-line interfaces. The development of Guix's interfaces was facilitated by the reuse of Nix's build daemon as the storage and deployment layer.

The guix package command is the main user interface: it allows packages to be browsed, installed, removed, and upgraded. The command takes care of maintaining meta-data about installed packages, as well as a per-user tree of symlinks pointing to the actual package files in /nix/store, called the *user profile*. It has a simple interface. For instance, the following command installs Guile and removes Bigloo from the user's profile, as a single transaction:

```
$ guix package --install guile --remove bigloo
```

The transaction can be rolled back with the following command:

```
$ guix package --roll-back
```

The following command upgrades all the installed packages whose name starts with a 'g':

```
$ guix package --upgrade '^g.*'
```

The --list-installed and --list-available options can be used to list the installed or available packages.

As of this writing, Guix comes with a user-land distribution of GNU/Linux. That is, it allows users to install packages on top of a running GNU/Linux system. The distribution is self-contained, as explained in Section 4.3, and available on x86_64 and i686. It provides more than 400 packages, including core GNU packages such as the GNU C Library, GCC, Binutils, and Coreutils, as well as the Xorg software stack and applications such as Emacs, TeX Live, and several Scheme implementations. This is roughly a tenth of the number of packages found in mature free software distributions such as Debian. Experience with NixOS suggests that the functional model, coupled with continuous integration, allows the distribution to grow relatively quickly, because it is always possible to precisely monitor the status of the whole distribution and the effect of a change—unlike with imperative distributions, where the upgrade of a single package can affect many applications in many unpredictable ways [7].

From a programming point of view, packages are exposed as first-class global variables. For instance, the (gnu packages guile) module exports two variables, guile-1.8 and guile-2.0, each bound to a <package> variable corresponding to the legacy and current stable series of Guile. In turn, this module imports (gnu packages multiprecision), which exports a gmp global variable, among other things; that gmp variable is listed in the inputs field of guile and guile-2.0. The package manager *and* the distribution are just a set of "normal" modules that any program or library can use.

Packages carry meta-data, as shown in Figure 4. Synopses and descriptions are internationalized using GNU Gettext—that is, they can be translated in the user's native language, a feature that comes for free when embedding the DSL in a mature environment like Guile. We are in the process of implementing mechanisms to synchronize part of that meta-data, such as synopses, with other databases of the GNU Project.

While the distribution is not bootable yet, it already includes a set of tools to build bootable GNU/Linux images for the QEMU emulator. This includes a package for the kernel itself, as well as procedures to build QEMU images, and Linux "initrd"—the "initial RAM disk" used by Linux when booting, and which is responsible for loading essential kernel modules and mounting the root file system, among other things. For example, we provide the expression->derivation-in-linux-vm: it works in a way similar to build-expression->derivation, except that the given expression is evaluated in a virtual machine that mounts the host's store over CIFS. As a demonstration, we implemented a derivation that builds a "boot-to-Guile" QEMU image, where the initrd contains a statically-linked Guile that directly runs a boot program written in Scheme [5].

The performance-critical parts are the derivation primitives discussed in Section 3. For instance, the computation of Emacs's derivation involves that of 292 other derivations—that is, 292 invocations of the derivation primitive—corresponding to 582 RPCs[6]. The wall time of evaluating that derivation is 1.1 second on average on a 2.6 GHz x86_64 machine. This is acceptable as a user, but 5 times slower than Nix's clients for a similar derivation written in the Nix language. Profiling shows that Guix spends most of its time in its derivation serialization code and RPCs. We interpret this as a consequence of Guix's unoptimized code, as well as the difference between native C++ code and our interpreted bytecode.

4.2 Purity

Providing pure build environments that do not honor the "standard" file system layout turned out not to be a problem, as already evidenced in NixOS [8]. This is largely thanks to the ubiquity of the GNU build system, which strives to provide users with ways to customize the layout of installed packages and to adjust to the user's file locations.

The only directories visible in the build chroot environment are /dev, /proc, and the subset of /nix/store that is ex-

[6]The number of derivation calls and add-to-store RPCs is reduced thanks to the use of client-side memoization.

plicitly declared in the derivation being built. NixOS makes one exception: it relies on the availability of `/bin/sh` in the `chroot` [9]. We remove that exception, and instead automatically patch script "shebangs" in the package's source, as noted in Section 3.4. This turned out to be more than just a theoretical quest for "purity". First, some GNU/Linux distributions use Dash as the implementation of `/bin/sh`, while others use Bash; these are two variants of the Bourne shell, with different extensions, and in general different behavior. Second, `/bin/sh` is typically a dynamically-linked executable. So adding `/bin` to the `chroot` is not enough; one typically needs to also add `/lib*` and `/lib/*-linux-gnu` to the chroot. At that point, there are many impurities, and a great potential for non-reproducibility—which defeats the purpose of the `chroot`.

Several packages had to be adjusted for proper function in the absence of `/bin/sh` [6]. In particular, libc's `system` and `popen` functions had to be changed to refer to "our" Bash instance. Likewise, GNU Make, GNU Awk, GNU Guile, and Python needed adjustment. Occasionally, occurrences of `/bin/sh` are not be handled automatically, for instance in test suites; these have to be patched manually in the package's recipe.

4.3 Bootstrapping

Bootstrapping in our context refers to how the distribution gets built "from nothing". Remember that the build environment of a derivation contains nothing but its declared inputs. So there's an obvious chicken-and-egg problem: how does the first package get built? How does the first compiler get compiled?

The GNU system we are building is primarily made of C code, with libc at its core. The GNU build system itself assumes the availability of a Bourne shell, traditional Unix tools provided by GNU Coreutils, Awk, Findutils, sed, and grep. Furthermore, our build programs are written in Guile Scheme. Consequently, we rely on pre-built statically-linked binaries of GCC, Binutils, libc, and the other packages mentioned above to get started.

Figure 10 shows the very beginning of the dependency graph of our distribution. At this level of detail, things are slightly more complex. First, Guile itself consists of an ELF executable, along with many source and compiled Scheme files that are dynamically loaded when it runs. This gets stored in the `guile-2.0.7.tar.xz` tarball shown in this graph. This tarball is part of Guix's "source" distribution, and gets inserted into the store with `add-to-store`.

But how do we write a derivation that unpacks this tarball and adds it to the store? To solve this problem, the `guile-bootstrap-2.0.drv` derivation—the first one that gets built—uses `bash` as its builder, which runs `build-bootstrap-guile.sh`, which in turn calls `tar` to unpack the tarball. Thus, `bash`, `tar`, `xz`, and `mkdir` are statically-linked binaries, also part of the Guix source distribution, whose sole purpose is to allow the Guile tarball to be unpacked.

Once `guile-bootstrap-2.0.drv` is built, we have a functioning Guile that can be used to run subsequent build programs. Its first task is to download tarballs containing the other pre-built binaries—this is what the `.tar.xz.drv` derivations do. Guix modules such as `ftp-client.scm` are used for this purpose. The `module-import.drv` derivations import those modules in a directory in the store, using the original layout[7]. The `module-import-compiled.drv` derivations compile those modules, and write them in an output directory with the right layout. This corresponds to the `#:module` argument of `build-expression->derivation` mentioned in Section 3.2.

Finally, the various tarballs are unpacked by the derivations `gcc-bootstrap-0.drv`, `glibc-bootstrap-0.drv`, etc., at which point we have a working C GNU tool chain. The first tool that gets built with these tools (not shown here) is GNU Make, which is a prerequisite for all the following packages.

Bootstrapping is complete when we have a full tool chain that does not depend on the pre-built bootstrap tools shown in Figure 10. Ways to achieve this are known, and notably documented by the *Linux From Scratch* project [1]. We can formally verify this no-dependency requirement by checking whether the files of the final tool chain contain references to the `/nix/store` directories of the bootstrap inputs.

Obviously, Guix contains `package` declarations to build the bootstrap binaries shown in Figure 10. Because the final tool chain does not depend on those tools, they rarely need to be updated. Having a way to do that automatically proves to be useful, though. Coupled with Guix's nascent support for cross-compilation, porting to a new architecture will boil down to cross-building all these bootstrap tools.

5. RELATED WORK

Numerous package managers for Scheme programs and libraries have been developed, including Racket's PLaneT, Dorodango for R6RS implementations, Chicken Scheme's "Eggs", Guildhall for Guile, and ScmPkg [16]. Unlike GNU Guix, they are typically limited to Scheme-only code, and take the core operating system software for granted. To our knowledge, they implement the *imperative* package management paradigm, and do not attempt to support features such as transactional upgrades and rollbacks. Unsurprisingly, these tools rely on package descriptions that more or less resemble those described in Section 3.3; however, in the case of at least ScmPkg, Dorodango, and Guildhall, package descriptions are written in an *external* DSL, which happens to use s-expression syntax.

In [21], the authors illustrate how the *units* mechanism of MzScheme modules could be leveraged to improve operating system packaging systems. The examples therein focus on OS services, and multiple instantiation thereof, rather than on package builds and composition.

The Nix package manager is the primary source of inspiration for Guix [8, 9]. As noted in Section 2.3, Guix reuses the low-level build and deployment mechanisms of Nix, but differs in its programming interface and preferred implementation language for build scripts. While the Nix language relies on

[7]In Guile, module names are a list of symbols, such as `(guix ftp-client)`, which map directly to file names, such as `guix/ftp-client.scm`.

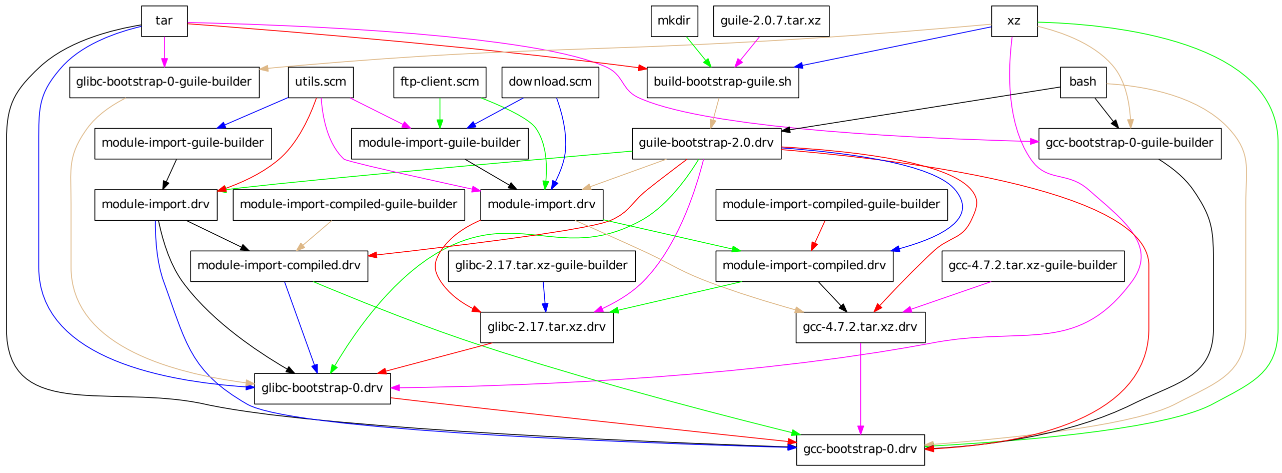

tar
mkdir
guile-2.0.7.tar.xz
xz
glibc-bootstrap-0-guile-builder
utils.scm
ftp-client.scm
download.scm
build-bootstrap-guile.sh
bash
module-import-guile-builder
module-import-guile-builder
guile-bootstrap-2.0.drv
gcc-bootstrap-0-guile-builder
module-import.drv
module-import-compiled-guile-builder
module-import.drv
module-import-compiled-guile-builder
module-import-compiled.drv
glibc-2.17.tar.xz-guile-builder
module-import-compiled.drv
gcc-4.7.2.tar.xz-guile-builder
glibc-2.17.tar.xz.drv
gcc-4.7.2.tar.xz.drv
glibc-bootstrap-0.drv
gcc-bootstrap-0.drv

Figure 10: Dependency graph of the software distribution bootstrap.

laziness to ensure that only packages needed are built [9], we instead support *ad hoc* laziness with the `package` form. Nix and Guix have the same application: packaging of a complete GNU/Linux distribution.

Before Nix, the idea of installing each package in a directory of its own and then managing symlinks pointing to those was already present in a number of systems. In particular, the Depot [3], Store [2], and then GNU Stow [10] have long supported this approach. GNU's now defunct package management project called 'stut', ca. 2005, used that approach, with Stow as a back-end. A "Stow file system", or `stowfs`, has been available in the GNU Hurd operating system core to offer a dynamic and more elegant approach to user profiles, compared to symlink trees. The storage model of Nix/Guix can be thought of as a formalization of Stow's idea.

Like Guix and Nix, Vesta is a purely functional build system [11]. It uses an external DSL close to the Nix language. However, the primary application of Vesta is fine-grain software build operations, such as compiling a single C file. It is a developer tool, and does not address deployment to end-user machines. Unlike Guix and Nix, Vesta tries hard to support the standard Unix file system layout, relying on a virtual file system to "map" files to their right location in the build environment.

Hop defines a *multi-tier* extension of Scheme to program client/server web applications [17]. It allows client code to be introduced ("quoted") in server code, and server code to be invoked from client code. There's a parallel between the former and Guix's use of Scheme in two different strata, depicted in Section 3.2.

Scsh provides a complete interface to substitute Scheme in "shell programming" tasks [18]. Since it spans a wide range of applications, it goes beyond the tools discussed in Section 3.4 some ways, notably by providing a concise *process notation* similar to that of typical Unix shells, and S-expression regular expressions (SREs). However, we chose not to use it as its port to Guile had been unmaintained for some time, and Guile has since grown a rich operating system interface on top of which it was easy to build the few additional tools we needed.

6. CONCLUSION

GNU Guix is a contribution to package management of free operating systems. It builds on the functional paradigm pioneered by the Nix package manager [8], and benefits from its unprecedented feature set—transactional upgrades and roll-back, per-user unprivileged package management, garbage collection, and referentially-transparent build processes, among others.

We presented Guix's two main contributions from a programming point of view. First, Guix *embeds* a declarative domain-specific language in Scheme, allowing it to benefit from its associated tool set. Embedding in a general-purpose language has allowed us to easily support internationalization of package descriptions, and to write a fast keyword search mechanism; it has also permitted novel features, such as an on-line auto-updater. Second, its build programs and libraries are also written in Scheme, leading to a unified programming environment made of two strata of code.

We hope to make Guix a good vehicle for an innovative free software distribution. The GNU system distribution we envision will give Scheme an important role just above the operating system interface.

Acknowledgments

The author would like to thank the Guix contributors for their work growing the system: Andreas Enge, Nikita Karetnikov, Cyril Roelandt, and Mark H. Weaver. We are also grateful to the Nix community, and in particular to Eelco Dolstra for his inspiring PhD work that led to Nix. Lastly, thanks to the anonymous reviewer whose insight has helped improve this document.

7. REFERENCES

[1] G. Beekmans, M. Burgess, B. Dubbs. Linux From Scratch. 2013. *http://www.linuxfromscratch.org/lfs/*.

[2] A. Christensen, T. Egge. Store—a system for handling third-party applications in a heterogeneous computer environment. Springer Berlin Heidelberg, 1995, pp. 263–276.

[3] S. N. Clark, W. R. Nist. The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries. In *In Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90)*, pp. 37–46, 1990.

[4] R. D. Cosmo, D. D. Ruscio, P. Pelliccione, A. Pierantonio, S. Zacchiroli. Supporting software evolution in component-based FOSS systems. In *Sci. Comput. Program.*, 76(12) , Amsterdam, The Netherlands, December 2011, pp. 1144–1160.

[5] L. Courtès. Boot-to-Guile!. February 2013. *http://lists.gnu.org/archive/html/bug-guix/2013-02/msg00173.html*.

[6] L. Courtès. Down with /bin/sh!. January 2013. *https://lists.gnu.org/archive/html/bug-guix/2013-01/msg00041.html*.

[7] O. Crameri, R. Bianchini, W. Zwaenepoel, D. Kostić. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *In Proceedings of the Symposium on Operating Systems Principles*, 2007.

[8] E. Dolstra, M. d. Jonge, E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA '04)*, pp. 79–92, USENIX, November 2004.

[9] E. Dolstra, A. Löh, N. Pierron. NixOS: A Purely Functional Linux Distribution. In *Journal of Functional Programming*, (5-6) , New York, NY, USA, November 2010, pp. 577–615.

[10] B. Glickstein, K. Hodgson. Stow—Managing the Installation of Software Packages. 2012. *http://www.gnu.org/software/stow/*.

[11] A. Heydon, R. Levin, Y. Yu. Caching Function Calls Using Precise Dependencies. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, PLDI '00, pp. 311–320, ACM, 2000.

[12] P. Hudak. Building domain-specific embedded languages. In *ACM Computing Surveys*, 28(4es) , New York, NY, USA, December 1996, .

[13] R. Kelsey. Defining Record Types. 1999. *http://srfi.schemers.org/srfi-9/srfi-9.html*.

[14] R. Kelsey, M. Sperber. Conditions. 2002. *http://srfi.schemers.org/srfi-35/srfi-35.html*.

[15] T. Morris. Asymmetric Lenses in Scala. 2012. *http://days2012.scala-lang.org/*.

[16] M. Serrano, É. Gallesio. An Adaptive Package Management System for Scheme. In *Proceedings of the 2007 Symposium on Dynamic languages*, DLS '07, pp. 65–76, ACM, 2007.

[17] M. Serrano, G. Berry. Multitier Programming in Hop. In *Queue*, 10(7) , New York, NY, USA, July 2012, pp. 10:10–10:22.

[18] O. Shivers, B. D. Carlstrom, M. Gasbichler, M. Sperber. Scsh Reference Manual. 2006. *http://www.scsh.net/*.

[19] M. Sperber, R. K. Dybvig, M. Flatt, A. V. Straaten, R. B. Findler, J. Matthews. Revised6 Report on the Algorithmic Language Scheme. In *Journal of Functional Programming*, 19, 7 2009, pp. 1–301.

[20] R. M. Stallman. The GNU Manifesto. 1983. *http://www.gnu.org/gnu/manifesto.html*.

[21] D. B. Tucker, S. Krishnamurthi. Applying Module System Research to Package Management. In *Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.

Data-transformer: an example of data-centered tool set

Michael A. Raskin[*]
Moscow Center for Continuous Mathematical Education
119002 Moscow
Bolshoy Vlasyevskiy 11
Russia
raskin@mccme.ru

ABSTRACT

This paper describes the data-transformer library, which provides various input and output routines for data based on a unified schema. Currently, the areas of the library's use include storage and retrieval of data via CLSQL; processing CSV and similar tabular files; interaction with user via web forms. Using the supplied schema, the data-transformer library can validate the input, process it and prepare it for output. A data schema may also include channel-specific details, e.g. one may specify a default HTML textarea size to use when generating the forms.

1. INTRODUCTION

Processing medium and large arrays of data often encounters the problem of the poor data quality. Use of human input may even lead to incorrect data formats. The same problem may occur when automated systems written by independent teams for completely unrelated tasks have to interact. Usage of wrong data formats may happen completely unexpectedly: spreadsheet processors sometimes convert 555-01-55 to 555-01-1955 just in case.

Finding and fixing such mistakes (both in the manually entered data and in the exchange format handling) usually relies on the automated verification. Of course, verification is implemented differently depending on the needs of the application in question.

Many software products use formally defined data schemas for codifying the structure of the data being handled. We could name XML Schemas and SQL database schemas among the examples of such formal schemas. XML schemas are declarative and SQL schemas are usually understood in a declarative way; this improves portability but sometimes restricts functionality.

[*]This work was partially supported by RFBR grant 12-01-00864-a

The data-transformers library is a library with the opposite approach: it tries to provide consistent handling with many small features without any hope for portability. Simple things are defined declaratively, but in many cases pieces of imperative code are included inside the schema. Moreover, declarative schemas for the separate data exchange interfaces can be generated out of a single data-transformer schema.

2. SCOPE

Initially this library has been written to support complex validations when parsing CSV files. So the focus of this specific library is on validating simple records one-by-one; validating hierarchical data sets is not in the scope. Handling of more complex data structures is done by wrappers which use the data-transformer library functions to handle each specific level of the hierarchy.

Interface of the data-transformer library supports many various operations, but all of them are applied to a single record at a time (although the record fields may contain complex data types for many of the operations).

3. DATA MODEL

The data transformers are defined by the schemas, usually written as s-expressions. For actual data processing a data-transformer class instance is created. It holds the data format definition, the data entry currently processed, and the errors found during processing.

The record format is defined as an array of fields; each field description (instance of the field-description class) holds the specified parameters from the schema and a cache of the computed default parameter values. All supported field parameters and the rules for deducing default values when necessary are defined inside the data-transformer library, although it is easy to add additional field parameters after loading the main data-transformer library.

The default values for the field parameters usually depend on the values of the other parameters.

The data held inside a data-transformer instance is an array of the values (in the same order as the fields); this array is supposed to contain the current representation of data according to the current needs of the application. A data-transformer instance is not meant to store the data for a long period of time, it only keeps the performed conversions uniform.

4. INPUT VERIFICATION

Let us consider a typical data flow. The data-transformer library is used to validate the data in a CSV file and put it into an SQL database. The process goes as follows. Check that the text looks like something meaningful, then parse it into the correct data type, check that this value doesn't violate any constraints, check that there are no broken cross-field constraints, combine the fields when the input requirements and the storage requirements are different (think of the date formats), generate an SQL statement to save the data or pass the list of errors to the UI code.

For example, a birth date is usually split into three columns in our CSV forms. The text format validation ensures that the day, the month and the year represent three integer numbers; parsing is done by parse-integer; the single-field validation ensures that year is in a generally reasonable range; the cross-field validation ensures that such a date exists (i.e. 2013, 04 and 31 are legal values for a year, a month, and a day number, but 2013-04-31 is not a legal date); and finally the data is formatted into the YYYY-MM-DD ISO format for the insertion into the database.

5. EXAMPLE DEFINITION

Example piece of a schema:

```
(defparameter *basic-schema*
  `(((:code-name :captcha-answer)
     (:display-name "Task answer")
     (:type :int)
     (:string-verification-error
     "Please enter a number")
     (:data-verification-error "Wrong answer")
     (:string-export ,(constantly "")))
    ((:code-name :email)
     (:display-name "Email")
     (:type :string)
     , (matcher "^(.+@.+[.].+|)$")
         (:string-verification-error
   "Email is specified but it doesn't
   look like a valid email address"))))
(let
  ((schema (transformer-schema-edit-field
              *basic-schema* :captcha-answer
      (lambda (x)
        (set-> x :data-verification
  (lambda (y)
    (and y (= y captcha-answer))))))))

  ; some code using the schema
  )
```

This description (a mangled piece of a registration form) illustrates the following attributes:
1) A code name for generating HTML, SQL and similar field identifiers and a human readable name used for generating the labels for HTML forms, CSV tables etc.
2) Types of the individual record fields. In our system the types are used mainly for generating the SQL table definitions.
3) Verification procedures. For the integer fields checking against the regular expression **" *[+-]?[0-9]+[.]? *"** is the default text format check, so it is not specified. Although the default check is used, a custom error message is specified for use when the format requirements are not met. The data verification is added to the schema later, right before the actual use. Note that the second verification step may rely on the previous ones to ensure that it is passed nil or a valid number.
4) Data formatting procedure. In this case, if the user entered a wrong CAPTCHA answer, there is no point in showing them their old answer, so we clear the field, instead.

6. CURRENTLY USED INPUT AND OUTPUT CHANNELS

Initial scope: CSV and SQL.

The only feature which is mostly specific for the CSV support is support for specifying a single date field and getting separate fields for day, month and year with verification and aggregation specified correctly by default. This is used in some of our web forms, too.

SQL-specific features are more numerous. A record definition has to contain the list of the relevant fields in the database; there is support for adding extra parameters and specifying the WHERE-conditions and the source tables as well. To simplify generating the SQL table definitions and the queries, one may specify the foreign keys as such where appropriate.

Creating a nice HTML output is quite similar to exporting data in the CSV format from the data viewpoint, one just needs a template. However, validating the web forms has some specifics. The data-transformer library supports generating the input fields for web forms, getting the field values from a Hunchentoot request object, handling the uploaded files (they are stored in a separate directory and the file paths are put into the database), preparing the data for use by the CL-Emb templates etc. Some of this functionality is also used for generating printable PDF documents (CL-Emb supports generating the TEXcode just as well as generating HTML).

7. WHAT DOES AND DOESN'T WORK FOR US

The data-transformer library has been started to avoid a mismatch between two data format descriptions (the code that validates the CSV files and the SQL schema) and unify validation. It grows organically, and therefore it is sometimes inconsistent.

When we started the project that includes the data-transformer library, we were looking for a good validation library designed to support many data exchange channels and have not found any that met our needs. I guess I shall look better if I have to start next big project from scratch. Or maybe I will just take this code.

It is nice to have the field definitions always in sync, of course. Although it is still possible that data storage and data loading procedures are slightly mismatched, this problem almost never occurs.

As one can see, we freely use Lisp code inside the data format definition. This means that we don't care about portability. On the bright side, this means that we can easily perform any needed check. For example, some of our online registrations for various events can check whether there is enough room for one more participant via a DB query. It is done inside the verification procedure.

It turned out that the lack of portability means that the data schemas are tied not only to the Common Lisp language itself. The code is also coupled with the tools we use (Hunchentoot, CLSQL, CL-Emb etc.) as well.

The excessive flexibility helps in an understaffed project. For example, there is some code for the event registrations. The main procedures of this code are almost never changed; the changes are mostly restricted to the page templates and fields lists, where all the definitions are relatively short.

The main problem is the frequent lack of time to find out a way to remove small annoyances. Sometimes some code repeatedly uses the library almost in the same way multiple times, and it is hard to find a good way to express these patterns. But it is probably not specific to our library.

Some functionality is still implemented in a wasteful way. For example, currently validating web form submission iterates over the field list and checks whether a validation procedure is defined for each field. It would be nice to allow generating a progn with all the verifying code included to remove the unnecessary branching (and the iteration over the fields lacking the verification procedures as well).

8. SOURCE

Data-transformer library is a part of the MCCME-helpers library (which serves as a core of one of our sites). Code may be found at
`http://mtn-host.prjek.net/viewmtn/mccme-helpers/`
`/branch/changes/ru.mccme.dev.lisp.mccme-helpers`

The Leonardo System and Software Individuals

Erik Sandewall
Computer and Information Science
Linköping University
58183 Linköping, Sweden
erisa@ida.liu.se

ABSTRACT

In this article we describe a software platform for cognitive intelligent agents, called Leonardo, which has been implemented in Lisp and which extends the principles underlying the design of Lisp in several important ways. We shall first describe Leonardo from the point of view of cognitive agents, in order to give the motivation for the design, and then discuss how it relates to the salient characteristics of the Lisp family of languages.

1. LEONARDO SYSTEM PHILOSOPHY

Leonardo is a Lisp-based system that has been developed as a software platform for cognitive intelligent agents. It is also very appropriate for other knowledge-based systems. The present section introduces the purpose and approach for the development of this system.

1.1 Definitions and Requirements

We define am *autonomous agent* as a software system that maintains an explicit representation of actions whereby it can not merely execute actions, but also reason about them, plan future actions, analyze the effects of past actions, and maintain and use an agenda. An autonomous agent is *cognitive* if it maintains a qualitative model of its environment and uses it for its action-related computations, for example for the purpose of planning and plan execution. A cognitive autonomous agent is called *intelligent* to the extent that its performance of actions qualifies for this description, so that it can be said to act intelligently.

Two characteristic properties of cognitive intelligent agents are of particular interest for our topic, namely the capability of *learning* and of *delegation.* One would like such an agent to be able to learn from its own past experience, and to be able to delegate tasks to other agents as well as to accept delegation of tasks from others.

The learning requirement implies a requirement for *longevity.* It is not interesting to have a system that can learn something during a short demo session and that loses what it has learnt when the session ends. Meaningful learning in practical environments, robotic ones or otherwise, will only make sense if the learning process can extend over months and years, so that many things can be learnt successively, results of learning can be used for further learning, and the learning process itself can be improved by learning. Long-time learning implies that the cognitive intelligent agent itself must be long-lived.

The delegation requirement implies that there must be a notion of *individuality* [2] . Suppose you have a software agent in your computer and you make a copy of it: is the copy then the same agent or another one for the purpose of delegation? Will both copies be responsible for completing the delegated task? Questions like this suggest the need for a concept of *software individuals,* ie. software artifacts for which it is well defined what are the separate individuals. If you make a copy of an individual then the copy shall be considered as a *separate* individual, in particular for the purpose of entering into contracts, for example when it comes to delegation of tasks.

As a cognitive intelligent agent needs both the capability of learning and the capability of delegation and cooperation, it follows that it should be organized as a *long-lived software individual* [3, 4] . This is the basic concept for the Leonardo software platform. We shall now outline what a session with a Leonardo individual is like. This outline corresponds to the demo that we use in talks describing our approach.

1.2 Starting a Session with a Leonardo Individual

A Leonardo Individual is technically a file directory with a particular structure of subdirectories and files, and dependencies between them. This directory can be moved from one memory device to another, which is how it can have a much longer life than the particular computer where it is located at some point in time. The individual is active when a computational session is started with respect to this directory.

Each individual consists of a number of *agents,* some of which may be cognitive or even intelligent, but some are not. Each agent can perform computational *sessions,* but not more than one session at a time. Agents can perform *actions* during these sessions, and these are then also actions of the individual as a whole. Such actions may change the individual's environment, but also its own internal structure, so that the individual can change over time as a result of its own actions.

The most natural way of starting a session is to right-click the icon representing the individual, ie. the particular directory in the file system that *is* the individual. This will start a command-line dialog whereby the user can request information from the agent. It will also (optionally) open a window for a *Session GUI* whereby the user can inspect current information in the agent and the individual.

If the user right-clicks the same icon in order to start another, concurrent session with the same agent, then the system rejects the attempt. If the user has *copied* the entire directory for the individual and then starts a session with the copy, then the system will recognize that a copy has been made, confirm this to the user, and reorganize its contents so as to represent that the copy is indeed another individual. On the other hand, if the user has just *moved* the directory to another place (this can only been done when no session is in progress), then the system will not consider the individual in question to be a new one.

The reorganization for a new individual involves obtaining a new name for the purpose of communication between individuals, removing the individual's memory of its own past actions, removing the individual's current agenda, and other similar changes. The way for it to obtain a name is to send a *registration request* to a Registrar, which is of course also a Leonardo individual. The Registrar serves a 'society' of Leonardo individuals and keeps them informed about new society members, their whereabouts and other network-level information.

Whenever a session is started, the Leonardo software identifies its environment: what is the name of the computer host where it is executing, what is the name of the Local Area Network where this host is located, who is the current owner, and so forth. The agent also creates a persistent data object (in the form of a file in the individual's directory structure) where it can store information about its own actions, observed external events, and other things that pertain to the session in question. For this purpose it maintains a sequential numbering of its own sessions.

To conclude, a newly started session with a Leonardo agent resembles a session with Lisp interpreter in the sense that it can receive commands in its command-line executive loop, but it differs in the following ways.

- The session has a lot of information about its own individual and agent, other individuals, the owner, the computer network structure, and so forth. All this information is represented in the knowledge representation language that is used for Leonardo, so that it is easily available for reasoning and other processing.
- In the style of an autonomous agent, it uses and manipulates explicit representations of its own actions.
- Through its explicit representation of information about fellow individuals and of actions, it is able to communicate action requests to other agents, as well as to respond to incoming action requests.
- The individual and each agent in it retains a history of its own past sessions and the significant actions in them.
- The characteristic information in the individual is accessible to the user via the Session GUI. This GUI can also be extended so as to support the needs of specific applications.

The following sections will describe these aspects in more detail.

2. HIGHLIGHTS OF THE SYSTEM DESIGN

The Leonardo system design is analogous to the design of a Lisp system insofar as it contains interpreter and compiler for a programming language where programs are represented as data structures, and the very same data structures can also be used for expressing application data and a variety of other languages in the system, besides the one that defines its actions. However, there are also significant differences from how Lisp systems are built. This section will describe and discuss those differences.

2.1 Representation and Execution of Actions

It has already been mentioned that the Leonardo system uses its own knowledge representation language. It is based on the use of KR-expressions (Knowledge Representation Expressions) which are somewhat similar to S-expressions, but with a more expressive syntax in particular through the use of several types of brackets. The language of evaluable KR-expressions is called the Common Expression Language, CEL.

Actions in Leonardo include both external and internal actions for the individual. External actions may be eg. sending an email, or moving a robot arm. Internal actions may be eg. changing some parts of its software according to a user request, or constructing a decision tree for dealing with some class of situations.

Actions are strictly different from terms. This is a major difference from Lisp, where everything is functions and terms, but some functions may have side-effects. Terms in Leonardo are always evaluated functionally. For actions, on the other hand, there is a distinction between *evaluation* and *execution,* and execution consists of several steps.

Consider for example the following action expression in CEL:

```
[repeat .v  <Smith.John  Baker.Bob>
   [send-email :to .v  :msg (quotation car-143)]]
```

where of course `.v` is a variable that ranges over the two entities in the sequence, and for each of them there is a `send-email` action. In each cycle of the loop there is an evaluation of the `send-email` expression, where the variable `.v` is replaced by its value, and the quotation of the car in question is computed or retrieved. The result of evaluation is therefore simply another action expression where parameters and terms have been replaced by their evaluated values.

After this, the evaluated actions are to be executed. Execution takes place in three steps, namely *precondition checking, core execution,* and *presentation.* Each action verb, such as `send-email` is therefore in principle associated with three separate things for its definition. The definition of the precondition may be a stub of code, but it may also (and preferably) be an expression in logic which determines whether the evaluated action is appropriate for core execution.

The definition of core execution may be written either in CEL or in Lisp, and is only concerned with doing the action as such. The outcome from the core execution routine shall be an *outcome record* that indicates whether the execution succeeded or failed, and what were the reasons for failure if applicable. If the action also has some kind of result (newly acquired information, for example) then this result is also included in the outcome record.

The definition of presentation specifies how the outcome of core execution, as described by the outcome record, is to be presented to the user.

Conventionally written programs for 'actions' usually integrate all of these aspects into one single, procedural definition, especially in dynamic programming languages where static precondition checking is not available. A definition of an action may therefore contain code for checking whether the given arguments are appropriate, and for displaying the results to the user, both in the case of success and in the case of failure. The definition of the core execution is sometimes dwarfed by this surrounding code.

The organization of action execution in Leonardo has several advantages. Specifying preconditions as logic expressions is more compact, and if preconditions are not satisfied then the system may use general-purpose routines for explaining the fault to the user, thereby reducing the burden on the display routine. When preconditions fail then it may also sometimes be possible to invoke general-purpose planning or replanning routines for finding a way around the problem.

The separation of the display routines is useful if the outcome is sometimes to be presented using the Command-Line Executive and sometimes using the Session GUI. More importantly, however, this separation is essential when tasks are sometimes to be delegated from one agent to another one. In this case the display of the results shall probably not be done in the host where the core execution takes place.

Similarly, when actions are delegated from one agent to another then the precondition checking is likely to be shared between the delegator and the delegee agent.

2.2 Included and Attached Facilities

A cognitive intelligent agent that is worthy of its name must have a number of capabilities that go beyond the mere programming language. The ability to operate a Session GUI has already been mentioned. In Leonardo this is realized using a general-purpose web server within the Leonardo software itself. This web server can be used both for the Session GUI and for any situation where an application requires the use of a web server for its specific purposes. The Session GUI server is special in the sense that it only accepts input from `localhost` clients.

The static and dynamic webpage definitions that are needed for these servers could in principle have been written in raw HTML, or using one of the many existing website design software systems. However, for Leonardo we have chosen to define a scripting language that can be used for both webpages and ordinary text and that is tightly integrated with the notation for defining and using actions. Just like S-expressions can be used both for Lisp programs and for definitions in a variety of languages such as KIF, KQML, PDDL and others, we are using KR-expressions both for terms, actions, preconditions, dynamic webpage definitions, plain text (such as the source text of the present article), and so forth. We are also beginning to use it for the display definitions of action verbs.

This is one example of a general principle: when there is a realistic choice, we prefer to define supporting facilities within the framework of the Leonardo system itself, its representation language, and its method of organizing actions and conditions. There are several other examples of this approach. For version management of the evolving software in Leonardo agents, for example, we use our own system which is integrated with all other handling of program and data files in the individual.

On the other hand, there are also some facilities that can not reasonably be reimplemented within the Leonardo system, such as a text editor, a web browser, or the Latex software to mention just a few. The Leonardo system has a systematic way of representing which software systems of these kinds are available on each particular host where a Leonardo system may execute, and how they shall be invoked.

2.3 Remote Facilities

In order to be full-fledged cognitive intelligent agents, Leonardo individuals must be able to use software resources in hosts other than their own. The Leonardo software therefore contains fairly extensive facilities for access to, and download of data sources from the Internet. This includes maintaining a register of such sources and their characteristic properties, login and password information for information sources, conversion of XML and other data languages to the knowledge representation language that is used within Leonardo, and so forth.

Moreover, since some information resources are best accessed by their API, the Leonardo system contains additional tools for defining how to obtain access to such API and how to decode the returned information.

3. PERSISTENCE AND MOBILITY

It was explained above that each Leonardo individual is a thing that exists for a period of time and usually changes during that period, and which, at each point in time, consists of a directory in a computer file system, with its contents in terms of subdirectories and files. It can only occur in one single such place at each point in time during its existence (except for backup copies), but it may move or be moved between places.

A Leonardo individual is therefore a 'mobile agent' in the sense that it can be moved from one host to another, or from one memory device to another, and if it is located on a detachable memory device then it can continue to function even if that device is moved from one host to another. Such moves are only done manually in the projects where we use the system. It would be straightforward to define an action whereby an individual can move itself to another location, but so far we have not seen any practical need for such a feature.

In order to achieve mobility for individuals in this sense, it has been important to design the Leonardo software so that the individual can have sessions under several different operating systems (Windows type and Linux type), and under different implementations of Common Lisp. The system has been developed under Allegro Common Lisp, but it is also operational under CLisp with minor restrictions. One step in the procedure for starting a session is to identify the choice of operating system and Lisp implementation at hand, and to load 'adapter' files containing definitions that are specific for these choices.

4. REPRESENTATION LANGUAGE

Just like S-expression syntax is being used as a syntactic framework for several different languages, the Leonardo system uses KR-expressions as a syntactic framework for several scripting languages. The basic layer of KR-expressions has five types of atoms, namely symbols, strings, and numbers which behave like in Lisp, but in addition there are *variables* and *tags.* Variables are symbols preceded by a full stop; tags are symbols preceded by a colon symbol.

An *entity* in a KR-expression can be either a symbol, or a composite entity which is formed using a *symbolic function* and its argument(s), surrounded by round parentheses. Composite entities behave like Prolog terms; they can have attributes and values just like atomic entities.

A *composite KR-expression* is either of the following:

```
A set, eg.        {a b c d]
A sequence, eg.   <a b c d>
A record, eg.     [travel :by john :to stockholm]
A term, eg.       (+ 4 9)
```

The *Common Expression Language* (CEL) is defined in terms of KR-expressions and contains a set of functions for use in terms, as well as a set of standard predicates. Literals are expressed as records, composite logical expressions as terms, for example as in

```
(and [is-substring "a" .x] [= (length .x) 3])
```

which has the value true in a context where `.x` is bound to the string `"abc"`.

The *standard evaluator* for CEL-expressions evaluates atoms to themselves, except for those variables that are bound in the evaluation context. Terms are evaluated by first evaluating their arguments, and then applying the leading function in the term, provided that no variable is still present in any of the arguments. Sets, sequences, and records are evaluated elementwise, so eg. the value of a sequence expression is a new sequence expression where each element has been replaced by its evaluated value, which may of course be itself.

Actions are expressed as records, and we have explained above how the performance of an action consists of standard evaluation of the action expression, followed by the three execution steps.

For additional details, please refer to the Leonardo documentation.

5. DOCUMENT SCRIPTING LANGUAGE

It was mentioned that several languages are defined in terms of KR-expressions and CEL syntax. The *Document Scripting Language* (DSL) is one such language, allowing evaluation of terms in the same was as for CEL-expressions, but with formatting commands and other document-oriented verbs for use in action expressions. This language is used for defining webpages in the servers that an individual operates, including the GUI server. The following is a simple example of an expression in DSL.

```
[itemize
   [item "First language:" [e "Lisp"]]
   [item "Second language:" [e "Scheme"]] ]
```

which if formatted into pdf will produce

- First language: *Lisp*
- Second language: *Scheme*

The following is a modified example defining a form with one field for a user id and one field for the user's password.

```
[request :to login-action :method post ^
  [table
    [row [box "User:" :charsize 20]
         [box [input text :tag user
                 :show "" :charsize 50 ]]]
    [row [box "Password:" :charsize 30]
         [box [input password :tag pw
                 :show "" :charsize 50 ]]]
    [sendbutton "Enter"] ]]
```

The subexpressions correspond almost one-to-one to HTML expressions for defining a form with two fields for user input. The following example shows how a simple loop may be formed:

```
[table
   [repeat .c (get .p has-children)
        [row [box .p]
             [box (get .p has-age)]
             [box (get .p has-birthday)] ]]]
```

When evaluated in an environment where the variable `.c` is bound to an entity representing a person, it will look up the value of the `has-children` attribute of `.c` which should be a sequence or set of other person entities, and generate a table with one row for each of the children, and with fields for the child's name, age, and date of birth, assuming that these data are also stored in the knowledgebase in the obvious way.

Notice that DSL uses variables, terms, and control operators (for repetition and conditionals, for example) in the same way as in executable CEL-expressions. For example, the expression `(get .p has-age)` might as well have occurred in the definition of a command script. It is only the reportoire of command verbs that strictly distinguishes DSL-expressions from CEL-expressions.

DSL facilitates integration between formatting and conventional computation. It is convenient for defining dynamic webpages, and also for tables and other generated structures in documents. It would be possible, but inconvenient to use it for running text, for example in an article such as this one. For this purpose there is a similar *Text Scripting Language,* TSL, which can be characterized by the following example.

```
[itemize
   [item The first language is [e Lisp]]
   [item The second language is [e Scheme]] ]
```

Thus TSL has very few reserved characters for the sake of formatting, effectively only the left and right square brackets, and the occurrence of a colon symbol at the beginning of a word.

TSL and DSL have the same internal representation and the same interpreter; it is only the surface syntax and the corresponding parsers that differ. DSL is appropriate for source documents that embed computation and interaction; TSL is appropriate for static text. The use of a common internal representation makes it easy to switch between the two language variants within the same source text. The Leonardo system contains translators from their common internal representation to LaTeX and to HTML. All research articles and reports in this project since late 2008 have been authored using TSL, and all webpages have been defined using DSL.

6. LEONARDO, LISP, AND LISP-LIKE LANGUAGES

The relation between Leonardo and Lisp may be discussed on several levels. First there is the comparison between S-expressions and KR-expressions with respect to convenience, readability, backward compatibility, and so forth. Next there is the comparison between Lisp function definitions and CEL-expressions. Finally one may also relate CEL and other S-expression-based languages, such as KIF and PDDL.

Beginning with the comparison between Lisp programming and CEL, the most important differences are as follows.

1. Lisp specifies that variables are represented as ordinary symbols, and non-evaluation is signalled using the quote operator. CEL treats variables as a separate syntactic type, and specifies that symbols evaluate to themselves. This is in line with standard mathematical notation, and with languages such as Prolog and KIF. It also turns out to be very convenient in practical programming.

Several S-expression-based languages use the question-mark as a prefix for variables, writing for example `?X` where CEL would write `.x` . The dot was preferred for being more discrete, but the idea is the same.

2. Lisp does not distinguish between functions that have side-effects and those that do not. CEL makes a strict and syntactically expressed distinction between terms and actions, where terms are evaluated functionally and actions are treated quite differently.

3. The separation of the different parts of the definition of an action verb has already been explained.

4. Since the outcome of performing an action is represented as an outcome record in Leonardo, it becomes possible to characterize the failure of an action and to deal with it in a procedural way, instead of by throwing and catching errors.

5. Leonardo has defined a markup language (DSL and TSL) that is closely related to the language for defining terms and actions (CEL). Lisp has, at best, a facility for writing HTML as S-expressions.

It should be noted that most of the detailed code of Leonardo applications is being written in Lisp, and the same applies for the implementation of Leonardo itself. This may change in the future, but it is the case today.

7. KR-EXPRESSIONS AS PUBLISHABLE NOTATION

With respect to KR-expressions and comparing them with S-expressions, one major consideration in the Leonardo project has been the integration of the software system with KR-related teaching materials. I am developing an on-line textbook for KR-based Artificial Intelligence [1] in which I want to have and use an easily readable notation for the variety of knowledge structures that are currently in use in AI. This includes logic formulas, of course, but it also includes other structures, such as decision trees, causal nets, hierarchical task networks, and others.

It is important, I think, to have a notation for these structures that can be used both in textbooks, in software documentation and other technical reports, and in the software system itself. In this way it becomes as easy as possible to transfer methods from the textbook to their actual use in the software system. The differences between CEL and Lisp notation that were described in the previous section go well with the goals of the textbook, in particular in the sense that CEL is closer to standard mathematical notation with respect to the use of variables and quotation. The somewhat richer notation of KR-expressions compared to S-expressions serve the same purpose. Some earlier authors have tried using S-expressions as the standard notation in their textbooks, but this has not become common practice.

Comparing ease of reading for different notations is a treacherous topic. I can only say that I have been a user of Lisp and done almost all my programming in this language since 1966, and I have often defended its single-parenthesis-based syntax in sceptical environments. However, even while being so used to the S-expression syntax, I still find the richer KR-expressions much easier to read and to work with. The use of a few different kinds of brackets, instead of just a single one, turns out to be a much bigger advantage for legibility than what one might think at first.

[1]http://www.ida.liu.se/ext/aica/

8. CURRENT USE

The Leonardo system is presently only used by the present author. However, during the previous years it has also been used by the students in my university course on Artificial Intelligence and Knowledge Representation. Each student in the course was given his or her own Leonardo individual and used it for doing and reporting the assignments that came with the course. The individual was enabled to download the definitions of assignments, support the user while doing them, pre-screen the results for correctness, and upload the results to the course administration individual. The teaching assistants for the course were therefore active Leonardo users.

The Leonardo system has also been used as the platform for a number of research projects in our group, including the development of the Common Knowledge Library (CKL) [2] Furthermore its services for document preparation and webpage definition are in continuous use.

An new project is presently about to start where Leonardo individuals with a facility for Hierarchical Task Management (HTM) are used for intelligent processing of large-scale datasets in bioinformatics.

9. A PERSPECTIVE ON SOFTWARE

We shall conclude with a somewhat philosophical note about the character of systems software in general. Contemporary software technology identifies some major types of software artifacts: operating systems, programming language implementations, database systems, web servers, email servers, and so forth. The roles of each of these types and the relationships among them are well established.

There have been some noteworthy experiments with organizing the entire software structure differently. In particular, the Lisp machines that were built in the late 1970's integrated the operating system and the implemented programming language into one coherent artifact. However it has not been followed by other similar efforts.

A Leonardo-style software individual cuts across the borders between these traditional types of software artifacts, but in a new way. Since it has a command-line executive and a capability for defining new commands in terms of existing ones, it has all the characteristics of an incremental programming system in the Lisp tradition. However, it also maintains a model of its computational environment in terms of computer hosts, memory devices, local area networks, as well as available software packages in each host, available servers on various hosts that can be accessed using a standardized http interface, and so forth. In this way it performs some of the functions that are otherwise considered as pertaining to an operating system.

Furthermore, the knowledgebase within the Leonardo individual performs many of the functions of a database system, in particular with respect to long-term preservation. A facility for working with very large scale databases is presently being developed.

[2] `http://piex.publ.kth.se/ckl/`

The present Leonardo system is implemented on top of the large Allegro and Clisp systems, which run on top of the large Windows or Linux operating systems. However, it has been designed with the long-term goal of moving it to a platform consisting of a minimal operating system and a minimal implementation of Lisp or another functional language, where major parts of the previous platform have been realized within the Leonardo system itself. Our hypothesis is that the resulting software architecture would be more effective and contain much fewer duplications of similar services.

10. A PERSPECTIVE ON INDIVIDUALITY IN SOFTWARE

The notion of software individuals has been fundamental in the design of the Leonardo system. It goes back to the notion of *biological software* [1] where the idea is that since human intelligence is "implemented" in biological tissue, and since machine intelligence must necessarily be implemented on software structures, it is reasonable to try to reproduce some of the characteristics of the biological substrate in the software medium. However, this should *not* be done by trying to simulate low-level biological processes in software; it should be done instead in ways that are characteristic of, and natural for software, and therefore as a kind of software engineering in the broad sense of the word.

Transferring the notion of an organism, or an individual to the frameworks of software was then an important first step, and the design and use of software individuals is our experiment with this idea. It turns out to have interesting connections to other issues in information technology, and in particular for digital rights management (DRM). The issue there is by what means may it be possible to identify one copy of a software system or a video recording, in such a way that it can be moved and backup-ed freely, but duplication is made impossible?

The present implementation of Leonardo assumes that the entire structure in a software individual is represented as a file directory with its subdirectories, and that the structures of different individuals are entirely disjoint. One may well ask whether it should be possible for several individuals to share the most basic parts of their code, so that update of that code can happen at a single time, instead of by repeated updates. One may also observe that current developments, including aspects of the "cloud" technology, tend to weaken the notion of what is the file system of a particular host or memory device. These developments add more possibilities to the space of possible designs, and makes it even more complicated. It is important I think not to be overwhelmed by all the new options, and to retain focus on what is important here, namely the notion of an individual as such, while at the same time being open to the new ways of implementing an individual that are becoming available.

11. DOCUMENTATION AND AVAILABILITY

The Leonardo website is located at

`http://www.ida.liu.se/ext/leonardo/`

This website contains documentation for how to start using the system and for several of its more specific aspects. It also contains links for the download of the current version of the system. All aspects of the software and its documentation is open-source under a GNU development license.

12. REFERENCES

[1] Erik Sandewall. Biological software. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1979.

[2] Erik Sandewall. On the design of software individuals. *Electronic Transactions on Artificial Intelligence*, 5B:143–160, 2001.

[3] Erik Sandewall. A software architecture for AI systems based on self-modifying software individuals. In *Proceedings of International Conference on LISP*, 2003.

[4] Erik Sandewall. The Leonardo Computation System. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science: Essays in memory of Gilles Kahn*. Cambridge University Press, 2008.

Session II

Tutorial: Typed Racket

Sam Tobin-Hochstadt
Northeastern University
samth@ccs.neu.edu

ABSTRACT

This tutorial proposal describes Typed Racket, a statically-typed dialect of Racket developed over the past 6 years. The tutorial would cover the basics of Racket and Typed Racket for anyone with a Lisp programming background and continue with a detailed example describing the implementation of an HTTP server implemented in Typed Racket.

1. INTRODUCING TYPED RACKET

Typed Racket [3] was originally developed to address the desire for optional type checking in Racket programs. Therefore, Typed Racket supports two key features not found in other languages with static types: (a) a type system designed to work with existing Racket programs [4], and (b) seamless interoperability between dynamically and statically typed portions of the program [2].

Typed Racket now provides support for a wide variety of Racket's distinctive features, ranging from dynamic type tests to delimited continuations; numeric types to first-class classes. Based on this support, Typed Racket is now used in many Racket systems, and recently two large systems have been built entirely using Typed Racket: an extensive library for mathematical computation[1] and a compiler from a subset of Racket to JavaScript[2].

The Typed Racket system itself has also been extended with numerous tools, including an optimizing compiler that leverages types to improve performance, with substantial performance gains on many programs [5]. The compiler is also able to provide recommendations to the programmer as to how they can improve the performance of their programs.

Recently, Typed Racket has been adapted to Clojure by Ambrose Bonnaire-Sergeant, in a system now being integrated into the main Clojure distribution as `core.typed` [1].

[1]http://docs.racket-lang.org/math/
[2]http://hashcollision.org/whalesong/

Submission to ELS '13

2. THE PROPOSED TUTORIAL

The proposed tutorial has three parts, and will last 120 minutes.

Introduction to Racket and the Racket environment.

This part will introduce Racket to a broad Lisp audience, with emphasis on key features such as the module system, how to use the IDE and command line tools, and available libraries.

Introduction to Typed Racket.

This segment, which will make up the bulk of the tutorial, will cover the basics of Typed Racket: the syntax, how typechecking works, what features are supported by the type system, and how integration with dynamically-typed Racket libraries works. It focuses on helping tutorial participants write small programs in Typed Racket.

A server in Typed Racket.

This portion will incrementally develop a simple dynamic web server, implemented entirely using Typed Racket and building on Racket's networking libraries. Participants will be able to write almost all the code necessary for the program.

A previous version of this tutorial was given at RacketCon 2012 in Boston in October 2012. Sample material as well as a video recording from that version are available at con.racket-lang.org.

3. REFERENCES

[1] Ambrose Bonnaire-Sergeant. A practical optional type system for Clojure. Honours Thesis, University of Western Australia, 2012.

[2] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS '06*, pages 964–974, 2006.

[3] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL '08*, pages 395–406, 2008.

[4] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP '10*, pages 117–128, 2010.

[5] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI '11*, pages 132–141, 2011.

System for functional and formal evaluation of Lisp code

Arturo de Salabert
Universidad Autónoma de Madrid, Escuela Politécnica Superior, Computer Science Dept.
Francisco Tomás y Valiente 11, 28049 Madrid, Spain
+34-914975531
arturo.desalabert@uam.es

ABSTRACT

This paper introduces Corrector, a code reviewing and grading tool designed to verify not only the functional accuracy of functions returning complex results like structures, networks or trees, but also to estimate the formal elegance and simplicity or the compliance to requirements of a solution.

Categories and Subject Descriptors

I.2.1, I.2.2 [**Computing Methodologies**]: Artificial Intelligence – *Games, Automatic analysis of algorithms, Program verification* D.2.4 [**Software**]: Software Engineering - *Software/Program Verification*

General Terms

Algorithms, Design, Experimentation, Languages, Verification.

Keywords

Unit testing, Common Lisp, Education, Plagiarism.

1. INTRODUCTION

Teaching a computer language can be a gratifying experience, but it is necessarily linked to the arduous task of reviewing and grading code. Not only functional correctness, but also style, elegance and creativity of the code can and should be evaluated. Furthermore, the grading must be systematic (kept constant along multiple reviews) and transparent (open to scrutiny by students). This is especially important when teaching Lisp not as yet another language, but as a tool to learn functional programming. Unlike Haskell, Lisp allows procedural, functional and OO programming paradigms, and does not enforce one programming style over another. Most students tend to think and code procedurally, or do not quite grasp the difference of approaches to be able to choose the most appropriate in each case. Therefore, part of the arduous task of reviewing code is not just checking that it produces the expected results, but that it is coded in a certain way. With this considerations in mind was developed Corrector, a software verification and evaluation tool that has been used at the UAM since 2007 to grade all students' code in two different AI courses.

2. SYSTEM OVERVIEW

Corrector does quite more than unit testing, but even for just unit testing it is able to be adjusted for flexibility, accepting multiple solutions and grading them according to their respective quality. Unlike unit testing, the objective of Corrector is not to accept or reject some function, but to *grade* its quality based on several criteria like functional correctness, performance and even style.

The behavior of Corrector is controlled by the instructor using a Corrector Script, based on a command language able to specify one or more solutions and their corresponding scores, to request multiple coding requisites or coding paradigms, or to penalize some others. Furthermore, since the parsing performed by these functions requires stripping down the code to some primary form, it can also be used to compare solutions and to detect plagiarism between them.

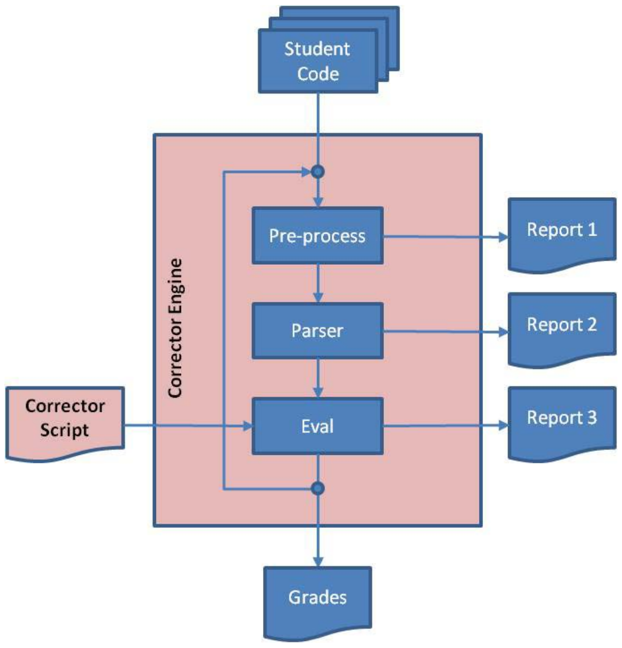


Figure 1. Corrector overview

Corrector processes a batch of files containing student solutions (algorithms in Lisp code) to exercises in various AI subjects like logic, search, logical programming, classification, machine learning, etc., and performs a sequence of evaluations. In addition, as explained earlier, Lisp is used with the intention of introducing students to functional programming; therefore the use of recursion, parallelism and iteration is also analyzed and graded.

3. SYSTEM COMPONENTS

The two major system components of the Corrector system are the engine and the script to define and control its execution. Figure 1 shows a block diagram of Corrector.

3.1 The Corrector Engine

The Corrector engine processes a selected range of text files containing Lisp code and applies a sequence of tests as specified by the script. Each step in the process produces a log detailed enough to provide students with explanation when required, similarly, each test in the script is graded and its results documented and saved. At the end of the process a final grade is given and a shortlist of the mistakes is compiled. The main modules of Corrector are described in the following sections.

3.1.1 Pre-process

This module is responsible for two tasks. The first one is performing initial format related checks and preparing the received files for further processing. Recoverable deviations from required standards like incorrect names or unnecessary functions are documented and penalized. Unrecoverable errors like unbalanced parenthesis or missing key functions abort the current correction and the process continues with the next file.

The second and most important task is to initialize the Corrector environment for the next evaluation process. Initial versions of the corrector required to close and restart the Lisp session to ensure that each correction starts with a clean system. Unbinding the known functions and symbols followed by a full garbage collection is not enough, since students may have defined auxiliary functions with unpredictable names, which not only reduces the available resources, but may also interfere with what other students try to do. Since all tests have timeouts and some of the tests may involve accurate timing of execution, it is very important that each battery of tests runs under the same conditions. The approach taken has been to parse the code to be evaluated (see next section) and keep track of all function and symbol definitions by the student to remove them after the Evaluator has finished. We could have used other techniques to accomplish this objective, like examining the symbols table, but considering that parsing the code is required anyway to evaluate programming style, this seems to be the simplest and most efficient solution.

3.1.2 Parser

Parenthesis-based closures and recursive compliance to S-expressions (sexps) are despised by the majority of students (and some teachers) as the main exponent of Lisp ugliness. However, the rare few who manage to overcome their initial repulsion may discover that the advantages widely overweight the strangeness. One of the considerable advantages of sexps is that both defining the language and parsing it are remarkably easy… as long as it keeps its lispness. Unfortunately not all Lisp is lispy, precisely, macros and functions that don't follow the S-expression definition must be treated like exceptions and are the difficult part to parse, e.g.: `defun`, `let`, `cond`, `do`… This is the reason why students are not allowed to define their own macros.

The Parser module analyzes student code, ignoring everything except function definitions. Student code is not allowed to have either assignments or symbol definitions or executions. Once a `defun` in found, it still has to be accepted by the Parser. Some functions may be protected against redefinition because they are part of the software that the student must complete. One of the components of the Pre-processor is a function that loads a file and protects all its functions (exceptions can be specified in the parameter list). The code of the Corrector itself is protected in the same way.

If the function definition is acceptable, the Parser analyzes recursively all the functions used within the user function. The recursion tree stops when either a Lisp primitive or a protected function is found. The first don't need further parsing for obvious reasons and the second are considered primitives at all effects. The fully expanded list of primitives or pseudo-primitives used by each student function is kept for future use. This procedure allows enforcing programming styles, such as recursion or parallelism, or detecting errors in the student's code, such as missing definitions or use of forbidden functions. These are primitives specifically banned by the professor either at global level or specifically for a particular exercise in order to force students to code in a particular programming style. The same mechanism can be used for the opposite objective, to enforce that certain functions are used.

3.1.3 Evaluator

Only when the student's code succeeds in the previous tests is it allowed to run. The Evaluator module performs a controlled evaluation of the student code. Evaluation is secured in two ways: execution time is limited by a timeout and errors are ignored. Both precautions protect the Evaluator against hang-ups, infinite loops or execution errors in the student code. In most cases, before running student code, some preparation is needed: assign values to global variables, define structures, etc. All this is normally done by Corrector Script commands. Once the environment of the student function is fixed as required, they are evaluated with specific arguments as specified by the script.

Results are compared with the expected results and grades are given accordingly to the closeness to the desired result. Comparing student results with expected results is not trivial, since there are many types of data that may need to be compared like lists, sets, arrays, trees, networks, or structures containing a combination of them. The type of comparison desired is specified by the script and can be different for each exercise.

3.2 The Corrector Script

Corrector runs a number of tasks autonomously, but when it comes to the functional evaluation of the code, it needs to be guided by a script created specifically for the exercises to be corrected. The Corrector Script specifies what tests must be run on which functions, in what context, under what conditions, what is the expected result, how many points is the test worth and how will the test be graded. If there is more than one acceptable result, each of the valid outcomes is given its corresponding value.

The script is a list formed by triplets belonging to two types of instructions, commands and tests. A script command is an instruction to Corrector to perform some action. It could be seen as a programming language because it has loops and conditions. A test is the specification of two things: the execution of a student function on some arguments and the expected result. The syntax of the two types of instructions is:

```
<cid>  <param>     <comment>
<test> <separator> <result>
```

Where `<cid>` is the command id; `<param>` is a list that will be used by the command as parameter; `<comment>` is just a string containing a comment that will be displayed in the log every time that this instruction is executed by the Evaluator; `<test>` is an execution of some student function; `<separator>` is any sequence of characters except the semicolon with the only objective of being a visual to identify what is a test and what is a result, it must be present to fill its place but it is ignored by the parser; `<result>` is the expected result or results of the test, can be any print format valid in lisp. Most of these items have multiple options. Describing the whole syntax would require more space than we have available. Figure 2 shows a simplified but complete case of script.

3.3 Main technical characteristics

3.3.1 360-degree evaluation

Running unknown student code in a dynamic interpreter has many risks but also interesting advantages. The major risk is that the foreign code may interfere (purposely or not) with the controlling process. A careful analysis of the source code and the banning of potentially dangerous primitives can help reduce this risk. The major advantage is that the student code can be evaluated, controlled or modified, from within. Furthermore, the CMND command allows the full power or Lisp from the script.

3.3.2 Robustness against code errors

The environment is refreshed after each correction, including all functions and symbols defined or modified by the student code.

3.3.3 Security, accountability

The parsing function prevents against Trojans or other attacks. The simplicity of Lisp S-expressions is a very strong advantage to facilitate code inspection. Use of macros is therefore not allowed in the student code. A detailed log provides accountability and reproducibility of the results.

4. ACKNOWLEDGMENTS

This work was partially funded by the Spanish National Plan of R+D, project# TIN2008-02081 and by the Autonomous Community of Madrid, CAM project# S2009/TIC-1650. Special thanks to Franz Inc. for their invaluable and reliable support since 2010.

5. ANNEX

Simplified example of a Corrector Script.

```
;;; Tests Battery for AI P3
(
 INIT (filtro '(setq 0 setf 0 sort 0) 0.1) "Penalizes each use of setq|setf|sort in all exercises"
 INIT (setq *max-bad-ifs*  2)              "Allows a max. of 2 bad-ifs, rest are penalized with *pena-filt-i*"
 INIT (setq *pena-filt-i* 0.02)            "2% penaliz. for each style or func. misuse"

 EJER (setq *pts-apdo* (/ 0.5 2))          "1. f-obj-galaxy [0.5 pts]"
 (f-obj-galaxy 'Urano '(Davion ares))      -> NIL
 (f-obj-galaxy 'Davion '(Davion ares))     -> T

 EJER (setq *pts-apdo* (/ 0.5 2))          "2. f-h-galaxy [0.5 pts]"
 (f-h-galaxy 'Sirtis *sensors*)            -> 0
 (f-h-galaxy 'Proserpina *sensors*)        -> 4

 EJER (setq *pts-apdo* (/ 1 4) *test* #'strsetequal *parms* '(act nom e-origen e-destino coste))  "3. nav-galaxy [1 pts]"
 (nav-galaxy 'Urano *holes*) -> NIL
 (nav-galaxy 'ares *holes*)  -> (#s(act :nom nav :org ares :dst avalon :cost 3) #s(act :nom nav :org ares :dst katril :cost 2)
                                 #s(act :nom nav :org ares :dst proserpina :cost 1))

 EJER (setq *pts-apdo* (/ 1 2) *test* #'equal)  "4. h-monot-p [1 pts]"
 (h-monot-p *sensors* *holes*) -> T
 (h-monot-p (mapcar #'(lambda (x) (list (first x) (* 2 (second x)))) *sensors*) *holes*)  -> NIL

 EJER (setq *pts-apdo* (/ 0.5 2))               "5. probl-estados [0.5 pts]"
 (probl-estados *galaxy-M35*)    -> (avalon mallory ares davion proserpina katril sirtis)
 (probl-e-inicial *galaxy-M35*)  -> ares

 CMND (setq *test* #'strsetequal *parms* '(fn nom args))   "Prepare structs comparison"
 (probl-fn-h *galaxy-M35*)   -> #s(fn :nom f-h-galaxy :args (((avalon 5)(mallory 7)(ares 4)(davion 1)(proserpina 4)(katril
3)(sirtis 0))))
 (probl-fn-obj *galaxy-M35*) -> #s(fn :nom f-obj-galaxy :args ((sirtis)))

 EJER (setq *pts-apdo* (/ 0.4 2) *test* #'equal) "6b. magnitud & cost comparison [0.4 pts]"
 (magnitud-de-nodo-<= nodo-0 nodo-01)  -> T
 (compara-costes-rama nodo-01 nodo-0)  -> NIL

 EJER (setq *pts-apdo* (/ 2 1) *test* #'strsetequal *parms* '(nodo estado ptr nom-act prof g f))  "7. expandir-nodo [2 pts]"
 (expandir-nodo (make-nodo :estado 'a :prof 1 :g 2 :f 10) *g01*)  ->
  (#s(nodo :estado b :ptr #s(nodo :estado a :ptr nil :nom-act nil :prof 1 :g 2 :f 10) :nom-act nav :prof 2 :g 3 :f 3)
   #s(nodo :estado f :ptr #s(nodo :estado a :ptr nil :nom-act nil :prof 1 :g 2 :f 10) :nom-act nav :prof 2 :g 7 :f 7)
   #s(nodo :estado g :ptr #s(nodo :estado a :ptr nil :nom-act nil :prof 1 :g 2 :f 10) :nom-act nav :prof 2 :g 12 :f 12))

 ;; ======== STYLE CORR. ========
 EJER (setq *pts-apdo* (/ 1 3))  "E1. unnecesario use of sort (earlier penalized as destructive, here for ineficient)"
 (chk-fl 'inserta-nodo-f '(sort 0) 1)  -> Returns-Grade
 (chk-fl 'busqueda-a '(sort 0) 1)      -> Returns-Grade
 (chk-fl 'expandir-nodo '(sort 0) 1)   -> Returns-Grade

 ;; ======== END ========
 FIN () "Fin Corrector"
 )
```

Figure 2. Simplified example of Corrector Script.

Platforms for Games and Adversarial Search

Arturo de Salabert
Universidad Autónoma de Madrid, Escuela Politécnica Superior, Computer Science Dept.
Francisco Tomás y Valiente 11, 28049 Madrid, Spain
+34-914975531
arturo.desalabert@uam.es

ABSTRACT

This paper introduces two platforms for the experimentation in games strategies within the study of Adversarial Search, part of the practical curriculum in Artificial Intelligence at the UAM. The first one (Thor) is a platform for performing tournaments amongst a large number of players. The second (NetGame) is a platform for executing P2P games in real time between remote players. It is oriented to performing individual matches and includes social network characteristics. Both platforms are fully implemented in Lisp. This paper focuses on technical and Lisp related issues. The educational potential of these systems has been described in previous work [1].

Categories and Subject Descriptors

I.2.1, I.2.2 [**Computing Methodologies**]: Artificial Intelligence – *Games, Automatic analysis of algorithms, Program verification* D.2.4 [**Software**]: Software Engineering - *Software/Program Verification*

General Terms

Algorithms, Design, Experimentation, Languages, Verification.

Keywords

Common Lisp, Adversarial Search, Education.

1. INTRODUCTION

Thor is a platform for performing tournaments. AI students are asked to send players (Lisp code) for a given game, which are confronted against each other and ranked according to performance. The platform is being used since 2010 by over 200 students per year, performing over 100,000 matches each term.

Thor checks periodically for new players and confronts them with a selection of other players to place them in the existing ranking. Before being allowed to execute, the code must pass some quality and security controls in order to ensure a minimum performance and to detect potential attacks to the integrity of the system. The code is also analyzed syntactically to insure that it follows some specific guidelines.

NetGame facilitates one-to-one games between players in real time by publishing a board of available players who accept being challenged by others. The social network dimension of the system is enabled by the ability of both sides to select their adversaries. Players higher in the ranking attract more attention and receive more challenges, but they can also decide to be more selective, so the system develops quickly a preferential attachment behavior [2]. Status, scores, historical data and other relevant information are compiled on a report and published on the web on real time.

Both platforms are independent, but they share common components. Thor involves three major functions, which for stability and resilience run in different servers: the upload server, where the students send their code, the publications server, where the results are published, and the proper tournament server, where players are evaluated and confronted to each other. NetGame does not need an upload function, since the code is never sent and runs on remote machines, but involves 2*N remote machines for each game played concurrently, where N is the number of players involved in a game.

2. THOR

2.1 Physical Architecture

To warrantee its availability, security and scalability the system functionality has been divided into three servers: a delivery server, where students upload their players, a publication server, where the ranking and other information is continuously updated, and the proper tournament server, not visible to the world. Thor periodically downloads players from the delivery server (plain text files with Lisp code), runs the tournament and publishes results uploading html files to the publication server. All three servers run Linux distributions. The tournament server runs Allegro 9 on CentOS. The delivery and publication servers run open-source applications over Ubuntu, the first an ftp with write but no read rights, the second a web server. This article focuses only on the tournament server.

2.2 Software Architecture

Although the system is available 24x7, its default status is sleeping, waiting for a new group of players to be uploaded by the students. The `start-timer` function is used to wake it up periodically. The incoming group of players is then processed as a batch. Figure 1 shows the main modules of the Thor tournament platform: Loader, Checker and Arbiter, and the control flow once it has been awakened. Their main components are described in the following sections. Each component writes on a common activity log with a configurable level of detail, including a timestamp and enough information to trace its activity along the 3 weeks that the tournament is normally active (Annex Figure 3).

2.2.1 *Loader*
The Loader module is responsible for downloading the players (via Unix shell commands) from the delivery server and for performing a number of authentication and validations. Since grades are given based on the performance of the players and students are very sensitive in this aspect, for accountability purposes files are never deleted. Downloading is selective; only files within a certain time span are retrieved and the rest are ignored. This approach allows reproducing the tournament situation at any desired time. The downloaded files are authenticated using private keys and validated against several format and content requirements. Files not conforming are silently ignored; they never reach the stage of being published, a disincentive to possible anonymous attacks or vandalism.

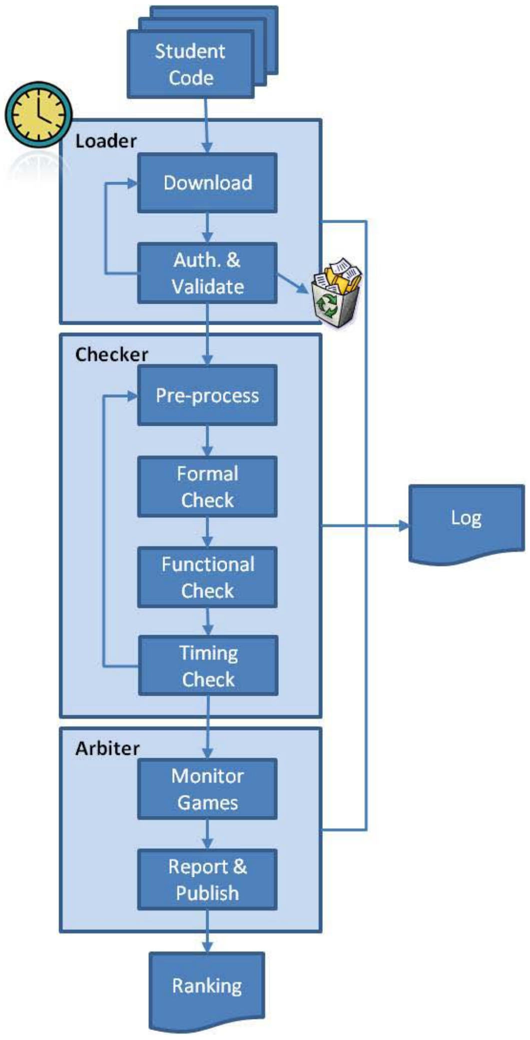


Figure 1. Thor overview.

2.2.2 *Checker*
The Checker is a much more complex module, responsible for analyzing the code and ensuring that it fulfills a number of formal, functional and performance requirements. All checks must result positive before the student code is evaluated, i.e. given control. Before entering the sequence of checks the student code goes through a Pre-processing phase, in which its functions are renamed to allow them to coexist with any other possible student code, some necessary but potentially dangerous primitives are renamed and substituted by secure versions, and the constructs for managing and reloading the player in the future are created. The later is fundamental to make the system resilient to failure, as explained later on.

Formal-checks analyze aspects like: error-free compilation, type and number of functions defined, programming style, use of recursive or iterative techniques, mandatory use of certain primitives, abstention of use of forbidden or dangerous primitives that could be used to interfere with the game mechanics (students have its full code), etc. Functional-checks control that the player can sustain a certain level of play, that it returns the expected type of data, etc. Performance-checks time the player when confronted to some fixed situations to make sure that it performs faster than a certain reference. Only players that pass all these tests are allowed to proceed to the next module.

2.2.3 *Arbiter*
The Arbiter is in charge of performing the individual games, creating and maintaining the ranking and reporting and publishing the results in html format. Maintaining the ranking involves assigning points to the players after each match. The tournament can operate in two modes, round-robin (RRT) and divisions (DT), each one of them with its corresponding point's mechanism. The final tournament is played in RRT, but there is no time to perform it between periodic updates, therefore the DT is normally used.

2.3 Main technical characteristics

2.3.1 *Failure resiliency*
Thor is designed to be able to run unattended 24x7 and to recover from most failure situations like a system crash, a power shutdown or a network failure. The status of the system is saved after each cycle of operations, including all relevant information related to the student code. After a reboot the system rebuilds itself up to the status of the last backup. Only games under the granularity of the periodic backup would be missing, but would be replayed automatically at the next round. Apart from the obvious lack of response during the duration of the failure, Thor is able to resume the tournament without human intervention, unnoticed by the users.

2.3.2 *Robustness, security, accountability*
Thor distributed architecture ensures its availability in case of failures of the web or ftp servers. A secret key procedure ensures authenticity. The parsing function prevents against Trojans or other attacks. The simplicity of Lisp S-expressions is a very strong advantage to facilitate code inspection. The use of macros is therefore not allowed in the student code. A detailed log provides accountability and reproducibility of the results.

2.3.3 *Dynamicity, scalability, game independence*
A big advantage of Lisp is its dynamic nature. Not only can the code be modified during run time, but the identity of the servers, the nature of the game, the level and verbosity of the log can be modified at any time. Furthermore, the decoupled architecture of the system allows multiple tournament servers to be running simultaneous and asynchronously, each of them publishing re-

sults either as a backup of each other or as a technique to obtain more throughput if the game becomes too heavy or there are too many users.

3. NETGAME

3.1 Physical Architecture

The server provides only a blackboard service called Majordomo (see Figure 4). The game is running simultaneously on the multiple machines of the remote contenders, and they send each other just their moves. For any one-to-one confrontation, the network topology is a triangle, with bidirectional communications between the players and unidirectional between the players and the server. Due to its distributed philosophy, the system is highly scalable. An unlimited number of games can be played simultaneously. Furthermore, the unavailability of the server does not affect the individual games. The players exchange moves and just report status to the server. The server produces a general report that is sent to the UAM web server, as in the case of Thor. Figure 2 shows a block diagram of the system. For 2 player games like checkers the total number of linked machines is 2*N remotes + 2 servers, where N is the number of simultaneous games, with Majordomo at the hub.

3.2 Software Architecture

Students participate in two roles, boaster or challenger. Boasters announce their availability to Majordomo, who publishes it. A game takes place when a request from a challenger is accepted by the chosen boaster. Everything is coded in Common Lisp but here, unlike the tournament, there is no code exchange between players; they could run on any OS and code in any language, as long as they can establish a P2P communication over TCP/IP sockets.

3.2.1 *Majordomo*

Majordomo is available 24x7. Normally its role is merely compiling and publishing, via an external web server, the status received from all players, and keeping a log of the main events. The log provides interesting data for a social network analysis of the evolution of the games. This will be addressed on a separate paper. All communications from the players are one-way *datagrams*. Once the challenger knows the contact data of his/hers desired partner, none of the players needs nor waits for any response from Majordomo. However, Majordomo has another level of operation, the security modus, where the contact data of the boasters is not published, just their alias. Any interested challenger needs to request from Majordomo the IP of its chosen partner. Based on its own rules (challenger preregistration, booster preferences, and black list) Majordomo may answer the request or remain silent. The secure modus requires two-way communication with the players, therefore is less scalable and it is dependent on the availability of Majordomo. However, all communication is asynchronous, so scalability limitations could slow down but not freeze the system.

3.2.2 *Players: boaster and challenger*

The player code can run in two modes: boaster and challenger. The only difference between them is in the P2P operation, the first is permanently listening through an open socket while the second just opens a temporary socket for the duration of one game. Therefore, students wanting to play as boasters must leave a computer on-line for as long as they announce in their initial commitment. They can also play as challengers, running a second session of the game either on the same machine or on another machine. Students playing in challenger role only need to choose from the published list of available boasters, configure the game parameters to use it as contender and run the game. The boaster will receive the challenge and may decide to accept it or to ignore it. If the game is running in secure level, the configuration of the boaster is not public and will not be known, not even to the challenger.

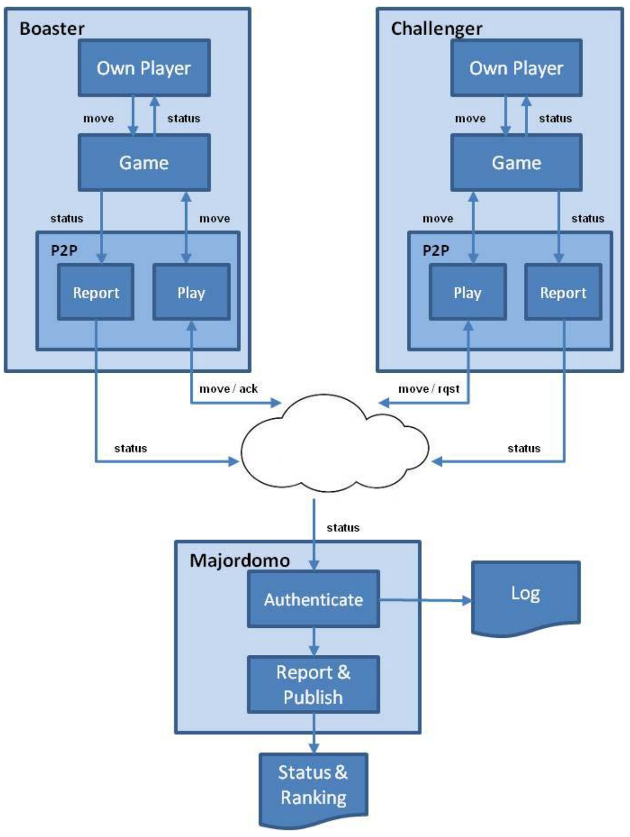


Figure 2. NetGame overview.

The game runs on both players simultaneous but independently, the local moves are supplied by the player's own code and communicated to the adversary via the P2P module, the adversary moves are received the same way. Together with the move also the resulting configuration is sent. If any of the players detects a difference between its own status and that received from the adversary the game is stopped and the situation reported to Majordomo.

3.3 Main technical characteristics

3.3.1 *Failure resiliency*

NetGain server is designed with the same failure resiliency characteristics than Thor; therefore we won't repeat them here.

3.3.2 *Robustness, security, traceability*

NetGain runs on multiple machines in a detached fashion. Failure of any or several of them will not affect the rest of the system. Due to the exchange of just moves, there is no risk of Trojans or code attacks. Dual execution of the game and the exchange of status ensure against code tampering, and the security mode, provides anti-intrusion measures like black lists or pre-registration of players IP's, although at the cost of higher bandwidth. Finally, the log maintained by Majordomo provides valuable data for social network analysis.

3.3.3 *Scalability, soft- and hardware independence*

The system is highly scalable because both bandwidth and processing requirements on the server are extremely low. Most of the processing and communications occur in and between the remote players. Since only moves are exchanged, every player could be running on different hardware and software.

4. ACKNOWLEDGMENTS

These platforms were partially funded by grants of the European Convergence Office in 2009 and 2010 and by the Spanish National Plan of R+D, project# TIN2008-02081 and by the Autonomous Community of Madrid, CAM project# S2009/TIC-1650. We are also grateful to Franz Inc. for their reliable support.

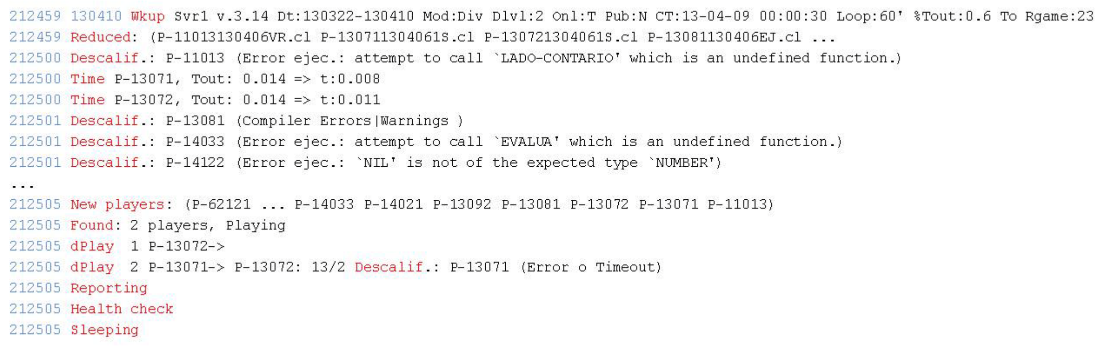
5. REFERENCES

[1] de Salabert, A., Díez, F., Cobos, R. & Alfonseca, M. 2011. An interactive platform for AI Games. In *Proceedings of Research in Engineering Education Symposium* (Madrid, Spain, October 04 – 07, 2011). 736-957. http://innovacioneducativa.upm.es/rees

[2] Price, D. D. S. 1976. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science* **27** (5): 292–306. doi:10.1002/asi.4630270505

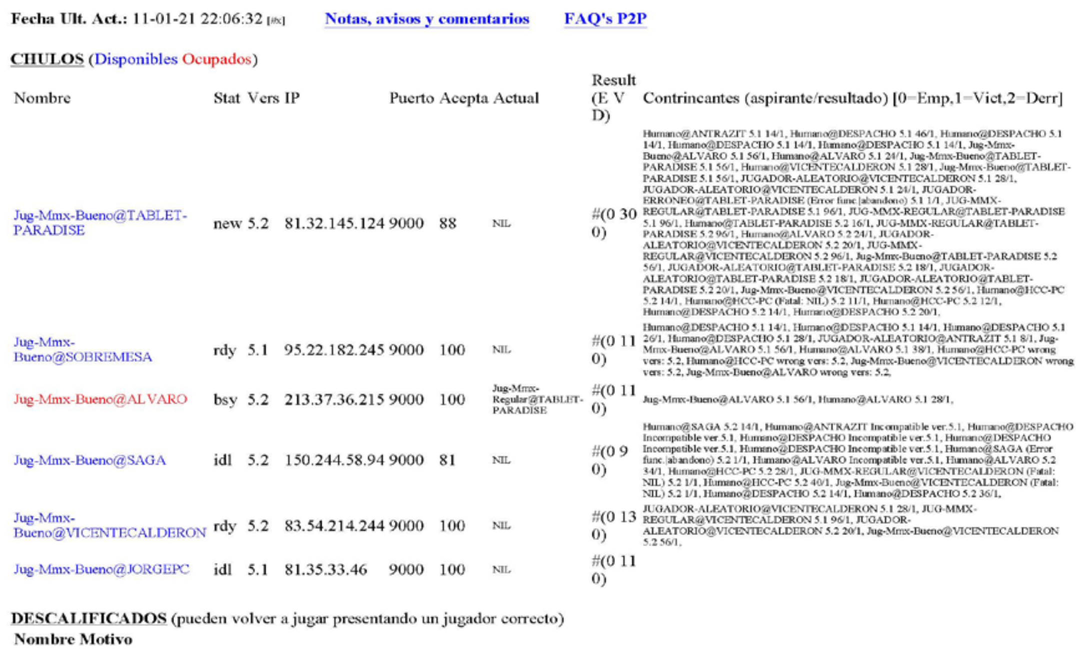
6. ANNEX

Two examples of output: A selection of Thor's log for a small batch of new players and a NetGame blackboard.



```
212459 130410 Wkup Svr1 v.3.14 Dt:130322-130410 Mod:Div Dlvl:2 Onl:T Pub:N CT:13-04-09 00:00:30 Loop:60' %Tout:0.6 To Rgame:23
212459 Reduced: (P-11013130406VR.cl P-130711304061S.cl P-130721304061S.cl P-13081130406EJ.cl ...
212500 Descalif.: P-11013 (Error ejec.: attempt to call `LADO-CONTARIO' which is an undefined function.)
212500 Time P-13071, Tout: 0.014 => t:0.008
212500 Time P-13072, Tout: 0.014 => t:0.011
212501 Descalif.: P-13081 (Compiler Errors|Warnings )
212501 Descalif.: P-14033 (Error ejec.: attempt to call `EVALUA' which is an undefined function.)
212501 Descalif.: P-14122 (Error ejec.: `NIL' is not of the expected type `NUMBER')
...
212505 New players: (P-62121 ... P-14033 P-14021 P-13092 P-13081 P-13072 P-13071 P-11013)
212505 Found: 2 players, Playing
212505 dPlay  1 P-13072->
212505 dPlay  2 P-13071-> P-13072: 13/2 Descalif.: P-13071 (Error o Timeout)
212505 Reporting
212505 Health check
212505 Sleeping
```

Figure 3. Example of Thor's log for a small batch of new players.



Fecha Ult. Act.: 11-01-21 22:06:32 [#x]   Notas, avisos y comentarios   FAQ's P2P

CHULOS (Disponibles Ocupados)

Nombre	Stat	Vers	IP	Puerto	Acepta	Actual	Result (E V D)	Contrincantes (aspirante/resultado) [0=Emp,1=Vict,2=Derr]
Jug-Mmx-Bueno@TABLET-PARADISE	new	5.2	81.32.145.124	9000	88	NIL	#(0 30 0)	Humano@ANTRAZIT 5.1 14/1, Humano@DESPACHO 5.1 46/1, Humano@DESPACHO 5.1 14/1, Humano@DESPACHO 5.1 14/1, Humano@DESPACHO 5.1 14/1, Jug-Mmx-Bueno@ALVARO 5.1 56/1, Humano@ALVARO 5.1 24/1, Jug-Mmx-Bueno@TABLET-PARADISE 5.1 56/1, Humano@VICENTECALDERON 5.1 28/1, Jug-Mmx-Bueno@TABLET-PARADISE 5.1 56/1, JUGADOR-ALEATORIO@VICENTECALDERON 5.1 28/1, JUGADOR-ALEATORIO@VICENTECALDERON 5.1 24/1, JUGADOR-ERRONEO@TABLET-PARADISE (Error func.|abandono) 5.1 1/1, JUG-MMX-REGULAR@TABLET-PARADISE 5.1 96/1, JUG-MMX-REGULAR@TABLET-PARADISE 5.1 96/1, Humano@TABLET-PARADISE 5.2 16/1, JUG-MMX-REGULAR@TABLET-PARADISE 5.2 96/1, Humano@ALVARO 5.2 24/1, JUGADOR-ALEATORIO@VICENTECALDERON 5.2 20/1, JUG-MMX-REGULAR@VICENTECALDERON 5.2 96/1, Jug-Mmx-Bueno@TABLET-PARADISE 5.2 56/1, JUGADOR-ALEATORIO@TABLET-PARADISE 5.2 18/1, JUGADOR-ALEATORIO@TABLET-PARADISE 5.2 18/1, JUGADOR-ALEATORIO@TABLET-PARADISE 5.2 20/1, Jug-Mmx-Bueno@VICENTECALDERON 5.2 56/1, Humano@HCC-PC 5.2 14/1, Humano@HCC-PC (Fatal: NIL) 5.2 11/1, Humano@HCC-PC 5.2 12/1, Humano@DESPACHO 5.2 14/1, Humano@DESPACHO 5.2 20/1,
Jug-Mmx-Bueno@SOBREMESA	rdy	5.1	95.22.182.245	9000	100	NIL	#(0 11 0)	Humano@DESPACHO 5.1 14/1, Humano@DESPACHO 5.1 14/1, Humano@DESPACHO 5.1 26/1, Humano@DESPACHO 5.1 28/1, JUGADOR-ALEATORIO@ANTRAZIT 5.1 8/1, Jug-Mmx-Bueno@ALVARO 5.1 56/1, Humano@ALVARO 5.1 38/1, Humano@HCC-PC wrong vers: 5.2, Humano@HCC-PC wrong vers: 5.2, Jug-Mmx-Bueno@VICENTECALDERON wrong vers: 5.2, Jug-Mmx-Bueno@ALVARO wrong vers: 5.2,
Jug-Mmx-Bueno@ALVARO	bsy	5.2	213.37.36.215	9000	100	Jug-Mmx-Regular@TABLET-PARADISE	#(0 11 0)	Jug-Mmx-Bueno@ALVARO 5.1 56/1, Humano@ALVARO 5.1 28/1,
Jug-Mmx-Bueno@SAGA	idl	5.2	150.244.58.94	9000	81	NIL	#(0 9 0)	Humano@SAGA 5.2 14/1, Humano@ANTRAZIT Incompatible ver.5.1, Humano@DESPACHO Incompatible ver.5.1, Humano@DESPACHO Incompatible ver.5.1, Humano@DESPACHO Incompatible ver.5.1, Humano@DESPACHO Incompatible ver.5.1, Humano@SAGA (Error func.|abandono) 5.2 1/1, Humano@ALVARO Incompatible ver.5.1, Humano@ALVARO 5.2 34/1, Humano@HCC-PC 5.2 28/1, JUG-MMX-REGULAR@VICENTECALDERON (Fatal: NIL) 5.2 1/1, Humano@HCC-PC 5.2 40/1, Jug-Mmx-Bueno@VICENTECALDERON (Fatal: NIL) 5.2 1/1, Humano@DESPACHO 5.2 14/1, Humano@DESPACHO 5.2 36/1,
Jug-Mmx-Bueno@VICENTECALDERON	rdy	5.2	83.54.214.244	9000	100	NIL	#(0 13 0)	JUGADOR-ALEATORIO@VICENTECALDERON 5.1 28/1, JUG-MMX-REGULAR@VICENTECALDERON 5.1 96/1, JUGADOR-ALEATORIO@VICENTECALDERON 5.2 20/1, Jug-Mmx-Bueno@VICENTECALDERON 5.2 56/1,
Jug-Mmx-Bueno@JORGEPC	idl	5.1	81.35.33.46	9000	100	NIL	#(0 11 0)	

DESCALIFICADOS (pueden volver a jugar presentando un jugador correcto)

Nombre Motivo

Figure 4. Example of the NetGame status blackboard.

Session III

DBL

A Lisp-based Interactive Document Markup Language

Mika Kuuskankare
Department of Doctoral Studies in Musical Performance and Research
Sibelius Academy
mkuuskan@siba.fi

ABSTRACT

DBL (Document Builder Language) is a novel, Lisp-based document markup language designed for document preparation. It is realized using the LispWorks Common Lisp Environment. Some of the functionality assumes the presence of a UNIX command-line, which is used to automate tasks, such as converting between image formats, when necessary, or launching build scripts. The DBL system consists of the DBL language, a parser, an interactive viewer/editor, and several backends for translating the internal representation of DBL into other formats, such as HTML.

DBL is a work in progress. This paper describes its current state.

Categories and Subject Descriptors

I.7.2 [**Document Preparation**]: Markup languages; I.7.1 [**Document and Text Editing**]: Document management

General Terms

Applications

1. INTRODUCTION

Markup languages specify code for formatting, both the layout and the style of a document. This is done within a text file by annotating the document using pieces of code called tags. Well-known and widely used markup languages include LaTeX and HTML. Some of the languages are primarily intended to be human readable. These are sometimes called lightweight markup languages. A lightweight markup language is a markup language that has a simple syntax and is easy for humans to enter and read. Markdown[6] is an example of such a language.

Approximately half of the dozen or so Lisp-based markup languages, found in the CLiki[4] website, deal with generating HTML or XHTML code. CL-WHO[3], CL-Markdown[1], and yaclml[8] are examples of such libraries. Apart from CL-Markdown, these libraries are also based on the Lisp s-expression syntax. CL-Markdown provides an interface to Markdown, which, in turn, is a text-to-HTML conversion tool. Other markup languages include, for example, Scribble[7]. Scribble extends the Common Lisp reader with the text markup syntax of the Scheme-based Scribble document preparation systems. cl-typesetting[2] is an extensive and a more general purpose typesetting and document preparation application. It is able to generate beautifully laid-out PDF documents directly from a description written in Lisp. According to the author, it is intended to be an alternative to the TEX-like typesetting systems.

The Document Builder Language (DBL) is a novel, Lisp-based document markup language for typesetting interactive and static multimedia documents. The DBL system consists of the DBL language, a parser, an interactive viewer/editor, and several backends. The DBL parser is implemented with the help of Esrap[5], a Lisp-based packrat parser by Siivola. However, we are using our own fork[1] which implements an extension, making it possible to use the Lisp reader as a lexer. Unlike most other Lisp-based packages, the DBL language is not based on S-expressions. Instead, for the most part, it is based on the concept of keyword-value pairs. The idea is that a DBL document should look, as much as possible, like normal text.

DBL is written in LispWorks Common Lisp and realized as a standalone ASDF[2] system. Primarily, DBL has been developed for the needs of PWGL[14]. PWGL, in turn, is a visual music programming language written on top of Common Lisp, CLOS and OpenGL. Within PWGL, DBL has two primary purposes. First, it provides the basis for its internal documentation system, making it possible to document the code, the system itself, and the user-libraries. Second, it is intended as an interface for producing interactive teaching material. PWGL specific extensions make it possible to mix interactive PWGL components, such as the musical score editor ENP[12], with standard User Interface (UI) components, such as text and images. Currently, the interactive PWGL Tutorial is written using a combination of DBL and normal PWGL patches. A document called 'PWGL-Book'[3] can be automatically produced using the tu-

[1]https://github.com/kisp/esrap
[2]Another System Definition Facility, `http://common-lisp.net/project/asdf/`
[3]PWGL Book can be downloaded from `http://www2.siba.fi/PWGL/downloads/PWGL-book.pdf.zip`

torial as a source. Furthermore, DBL can also be used as a documentation tool for MacSet[10], which is, along with PWGL, one of the music-related software packages developed at Sibelius Academy.

The rest of the paper is structured as follows. First, we briefly introduce the DBL markup language. Next, we describe in detail the collection of currently available DBL components. Finally, we present a framework for a music-related application realized with the help of DBL and offer some concluding remarks and suggestions for future developments.

2. DBL

DBL has three main components, the language, the parser, and the backends. The DBL parser translates source code written in the DBL language into an internal object representation called DBL document. The DBL document is a container object that holds a list of DBL components. The backends, then, work with the internal representation to produce output in different formats, such as CAPI[4], HTML, and PDF (see Figure 1).

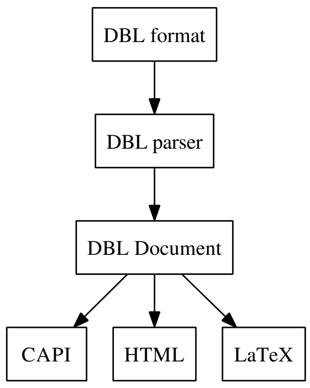


Figure 1: The DBL workflow

2.1 DBL Language

The DBL language, for the most part, is based on keyword-value pairs instead of s-expressions. However, Lisp can be used to programmatically generate documents or content. The motivation behind this design is to create a language that:

(1) could easily be written by hand, for instance when using a text processing software, and

(2) would be close enough to Lisp to make it possible to algorithmically generate both the format and content.

Appendix A shows an example where pieces of 'lorem ipsum', the de facto nonsensical placeholder text used in publishing, are generated automatically by calling a Lisp function `lorem-ipsum`.

[4]Common Application Programmer's Interface is a library by LispWorks for the LispWorks Common Lisp programming environment for implementing Graphical User Interface (GUI) elements and their interaction. It provides a standard set of GUI elements and their behaviors, and provides for a CLOS-based way of defining new ones.

DBL components are created by naming the component using an appropriate tag. The tags are Lisp keywords, such as :figure or :table. The tag can be followed by one or more mandatory attributes and any number of optional attributes. The attribute values themselves are usually strings, numbers, or keywords. All DBL components can also have an ID. The ID can be used for message passing between the components in interactive applications. An example of a DBL definition for a component of type figure is given below:

DBL Code

```
:figure "myimage.png"
:caption "My image caption"
```

The :figure tag requires one mandatory attribute, a string naming the file of the image. The image file has to be in the same directory as the DBL document referring to it (in case no matching file is found, a special 'image not found' image is displayed instead). The :caption is optional and takes a string as a value.

2.2 DBL Parser

The DBL parser is written with the help of Esrap, a Common Lisp packrat[9] parsing library. A packrat parser guarantees linear parse time through the use of memoization. Previous parse attempts for a particular grammar expression at a particular location are cached. If the grammar involves retrying the same expression at that particular location again, the parser skips the defined parsing logic and just returns the result from the cache. Packrat parsers also lend themselves well to designing dynamic and extensible languages. The following sample code shows the parser rule for the *figure* component:

DBL Code

```
(defrule figure (and (figure-p dbl-token)
                     (or (stringp dbl-token)
                         (pathnamep dbl-token))
                     (? dbl-id)
                     (* figure-options))
        (:destructure (token filename id args)
         (declare (ignore token))
         (apply #'make-instance 'figure
                :id id
                :filename filename
                (apply #'append args))))
```

The rule states that a parsable figure definition consists of a proper figure tag, a string or a pathname naming an image file, an optional ID and any number of optional arguments. If the form is successfully parsed an object of a type figure is created. Some of the DBL components, such as the figure, require only an opening tag. Others, however, require both opening and closing tags to create an environment or a hierarchical structure. The section component, for example, is defined by enclosing DBL item definitions between :section-start and :section-end tags.

Normally, a parser performs some sort of lexical analysis, by which the input character stream is split into meaningful symbols defined by the grammar. In our case, we don't look at the input stream on a character by character basis; instead, we have extended the Esrap parser to use the Lisp reader as a lexer. In this way, we are able to read Lisp objects, such as keywords and strings, and, at the same, take an advantage of the flexibility of the Esrap parser to define the logic of the DBL language as a whole. The dbl-token rule is at the center of the DBL parser. It reads from the input stream one Lisp form at a time while consuming all of the excess whitespace characters. The dbl-token rule is defined as follows:

DBL Code

```
(defrule dbl-token (and read (* (or #\space #\newline)))
  (:destructure (token whitespace)
   (declare (ignore whitespace))
   token))
```

2.3 DBL Backends

Currently, there are four official backends: (1) CAPI, (2) HTML (3) LaTeX, and (4) DBL. The DBL backend reverses the process of the DBL parser by converting the internal representation into text. Eventually, this backend will be used to build an automatic test suite.

In principle, all the backends work in a similar manner. They create a project folder where the translated document and all of the supporting files, such as images and template files, are copied. The template files normally include a collection of default images, such as the 'image missing' image and, in the case of the HTML backend, also a CSS style sheet. The LaTeX template, in turn, includes a defaults.tex file and several style files defining special formats and commands. When needed, the backbends convert the original images into formats better suited for the backend in question. For example, in the case of HTML, EPS files are converted into PNG's with the help of ImageMagick. Finally, in the case of the LaTeXbackend, a special build script is also executed, which, in turn, calls pdflatex and makeindex to typeset the document.

New DBL backends can be implemented by defining a backend and a set of dbl-export methods. A minimal DBL backend definition would be as follows:

Lisp Code

```
(def-dbl-backend html ())

(defmethod dbl-export ((self document) (backend html))
  ;;; do backend specific inits here
  (dolist (item (items self))
    (dbl-export item backend))))
```

In addition, for every DBL component, a corresponding dbl-export method would need to be defined.

Appendix A shows a sample document typeset with the CAPI backend. The CAPI backend translates the DBL components into CAPI-based interface elements. The CAPI-based DBL documents may contain interactive parts, such as buttons and, when used in tandem with PWGL, also music notation editors, patches, etc.

2.4 Supporting Tools

The DBL package also incorporates the following tools for viewing and editing DBL documents:

- ***DBL Viewer*** is used to render DBL documents using the CAPI backend. It accepts both individual DBL files and directories containing DBL documents and accompanying data. This is the primary front-end for the users.
- ***DBL Editor*** is a tool that makes it possible to edit and view DBL documents. It supports update while typing and syntax validation. Furthermore, it can be connected to, for example, 'aspell' for rudimentary spell checking.

3. DBL COMPONENTS

This section describes the current set of DBL components. Figure 3 shows the DBL component class hierarchy. The two base classes are dbl-item and dbl-container. Every DBL component inherits from a dbl-item. DBL-container objects can hold an arbitrary number of dbl-items. In general, container objects can hold dbl-items of any kind. There are, however, some exceptions. For example, the items environment (see section 3.9) cannot contain sections (see section 3.1).

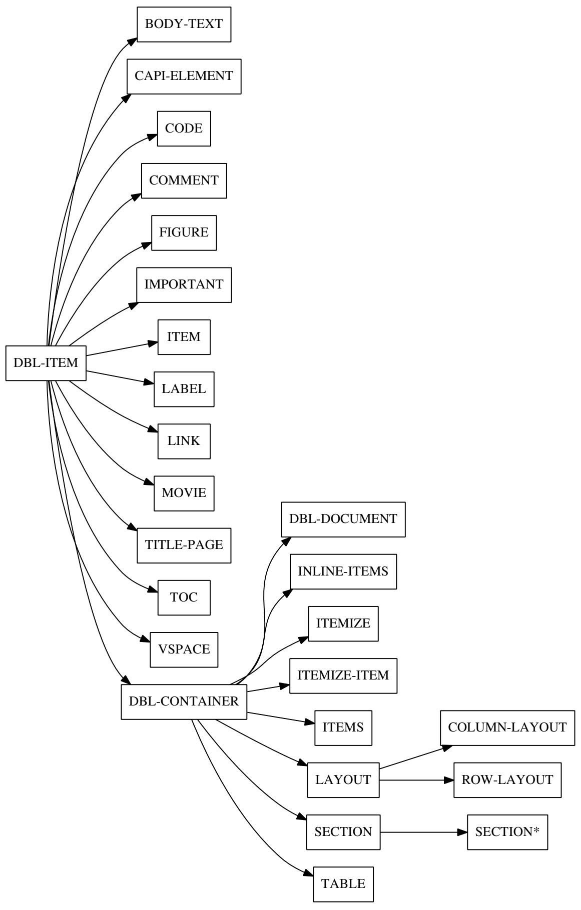


Figure 2: The DBL object hierarchy.

The following paragraph lists the DBL tags. Most of the time, the name of the component is the same as that of the tag. However, when this is not the case, the component name is also shown: (1) :section-start/:section-end for section, (2) :body-text, (3) :figure, (4) :code, (5) :items/:inline-items, (6) :link, (7) :important, (8) :table, (9) :comment, (10) :items-start/:items-end for itemize, (11) :item-start/:item-end for itemize-item, (12) :column-layout/:row-layout, (13) :movie, (14) :toc, (15) :vspace, (16) :capi, and (17) :title-start/:title-end for title page.

The following sections discuss the most important DBL components.

3.1 section

A component of type *section* is created using the :section-start and :section-end tags. Sections can be arbitrarily nested to create subsections. The sections are automatically numbered. The :section-start tag must be followed by a string naming the section title. The :section-end tag is not followed by any arguments.

3.2 body-text

The *body-text* component defines a piece of 'bread' text. It is indicated using the :body-text tag and it takes a list of strings as arguments. Each string begins a new paragraph.

3.3 figure

The *figure* component is defined using the :figure tag and it has one required attribute, the pathname of the image. The pathname is usually given as a relative pathname, e.g., 'piechart.png'. In this case, an image named 'piechart.png' must be located in the same directory as the document that refers to it. Absolute pathnames can be used, too. However, this makes the DBL documents less portable across different machines. The :figure component accepts two optional attributes – :caption and :adjust. The :adjust can have one of three values: :left, :right, or :center).

3.4 code

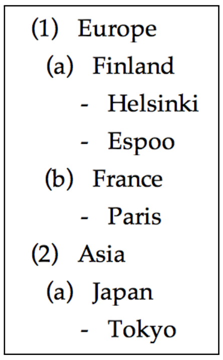
The *code* component is used to define text that is displayed using a fixed-width font. It preserves both spaces and line breaks. The *code* components inherits from the body-text. It has one optional attribute – :caption.

3.5 items

The *Items* component is used for simple ordered or unordered lists. The list is defined using the :items tag followed by any number of list items. The list items are entered using a combination of hyphens, i.e., '-', '- -', or '- - -', and a string. The number of hyphens defines the list item level. The items cannot contain other DBL components. For more complicated list structures use the *items environment* (section 3.9). The DBL definition given below was used to produce the list shown in Figure 3:

DBL Code

```
:items
- "Europe"
-- "Finland"
--- "Helsinki"
--- "Espoo"
-- "France"
--- "Paris"
- "Asia"
-- "Japan"
--- "Tokyo"
```



(1) Europe
  (a) Finland
    - Helsinki
    - Espoo
  (b) France
    - Paris
(2) Asia
  (a) Japan
    - Tokyo

Figure 3: An items component typeset with the help of the CAPI backend.

3.6 link

The *link* component defines a hyperlink, which is used to link to a local file or directory, or to a web page. It has one mandatory attribute: a string naming a URL. Furthermore, the :link tag can be followed by two optional attributes: (1) :link-text defines the piece of text that is displayed instead of the URL, and (2) :protocol specifies how to open the linked document and can be either :http or :file. Clicking on the link text opens the linked resource in all of the official backends.

3.7 important

The *important* component defines a piece of text that is typeset so that it stands out from the bread text. It inherits from the body-text component and has the same attributes and arguments. The following example shows how the *important* component appears in the final document:

> In the beginning the Universe was created.
> This has made a lot of people very angry
> and been widely regarded as a bad move.
> – Douglas Adams

3.8 table

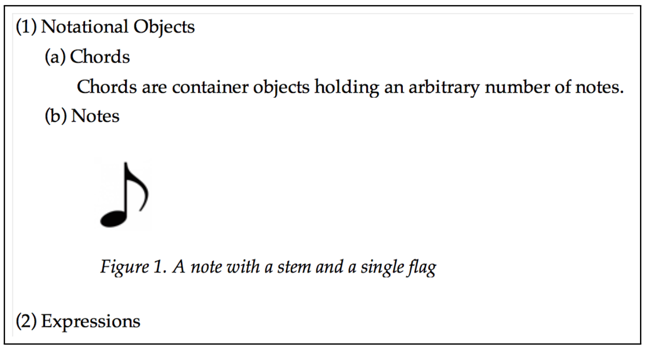
The *table* component defines a simple table for arranging data in rows and columns. The cell content is defined by entering a required number (a multiple of the number of columns in the table) of strings in row-major order. The table has 4 optional attributes: (1) :columns; (2) :title, which is used to give the table a title string; (3) :caption, which is used to give the table a caption string; and (4) :has-title-column. The :has-title-column attribute is a boolean value specifying whether the columns have titles. If the value is true then the first row of strings represents the titles for each column.

3.9 items environment

As opposed to its simpler counterpart presented in section 3.5, the *items environment* can be used to define more complicated lists. The items environment must be surrounded by :items-start/:items-end tags. Inside the environment, the items can be nested within one another; they can be up to four levels deep and must be enclosed within the :item-start and :item-end tags. The most important difference between this and the simple *items* component is that the components of the items environment can contain any valid DBL elements (except for sections). The following figure gives an example of a complex list, where the items contain not only text but also images.

DBL Code

```
:items-start
  :item-start "Notational Objects"
    :item-start "Chords"
      :body-text "Chords are container objects holding an
            arbitrary number of notes."
    :item-end
    :item-start "Notes"
      :figure "note.png"
      :caption "A note with a stem and a single flag"
    :item-end
  :item-end
  :item-start "Expressions"
  :item-end
:items-end
```



(1) Notational Objects

(a) Chords

Chords are container objects holding an arbitrary number of notes.

(b) Notes

Figure 1. A note with a stem and a single flag

(2) Expressions

Figure 4: An items component typeset with the help of the CAPI backend.

3.10 row/column layouts

The *row-layout* and *column-layout* can be defined using the :row-start/:row-end and :column-start/:column-end tag pairs. These layouts can contain any valid DBL components and the layouts can be nested. The layouts can be used for, among other things, placing several images side by side.

3.11 toc

In the case of DBL projects (defined as a folder of DBL documents) the *toc* component inserts the table of contents into a DBL document. It takes one optional argument, the section depth. The depth refers to the directory hierarchy starting from the top-level project pathname. Only directories are considered.

3.12 capi-element

The *capi* component can be used to generate normal UI components, such as buttons. The following CAPI classes are supported by the backends: (1) capi:push-button, (2) capi:radio-button-panel, and (3) capi:check-button-panel. The components can only be used within the CAPI backend. In terms of the other backends, if possible, these components are exported only for completeness and they don't have any function other than a visual one.

3.13 movie

The *movie* component is specific to the Mac OS X operating system. It uses Apple QuickTime to insert a component, in any of the supported multimedia formats, into a DBL document. The CAPI and HTML backends are able to export most of these components. Also, the LaTeX backend uses an experimental multimedia class for embedding movies into the resulting PDF document.

4. DBL DOCUMENTS

DBL documents are plain text files containing DBL code. They can be written and edited using any text editor. A folder containing DBL documents and supporting files is called a DBL project.

DBL documents can have several document options, such as font, font size, and language. The options can be set on file-by-file basis, or, in the case of DBL projects, they can be saved in a special file located in the root folder of the project. The file must be named '_settings' and it must contain valid DBL document options.

One of the more interesting document options is :language. This option makes it possible to select the primary language and any number of secondary languages. The following piece of DBL code establishes that the primary language of the document is English and that the secondary language is French. The order and the number of the languages can be changed at any time.

DBL Code

```
:language :en :fr
```

Now, a paragraph of text can be assigned to any of the given languages using a special DBL language tag that names the language. Appendix B shows an example of a paragraph where the same text is entered in two different languages. The secondary language is typeset in a slightly different manner to make it easy to distinguish it from the primary one. If there is no paragraph text corresponding to the specified language then the text for the primary language is used. In this case, the primary language is English, and the secondary language is French. However, if the ':language' property would name only French language, then the document would show only the paragraphs with French text using the default paragraph style.

5. A DBL EXAMPLE

In this section we give a brief example of a music pedagogical application realized with the help of DBL running on top of the PWGL environment. Here, we implement a DBL document that at the same time teachers basic harmonization and incorporates components for realizing interactive exercises as a part of the document. Using the combination of DBL and PWGL, the whole process of teaching the subject matter, realizing the pertinent exercises, and checking the answers given by the users, can be automated.

The explanatory text and images are realized as standard DBL components. ENP, our music notation program, is used here as a CAPI-based GUI component for representing the exercise and inputting the students solution. The harmonization rules are implemented using our music scripting system ENP-script[11]. We implement a rule that checks for correct harmonic functions, as well as a rule that checks for parallel movement.[5] These rules are not only used to check the correctness of the assignment, but also to present visual clues and textual advice for the user using the elaborate expression system provided by ENP (see [12] for examples).

The assignment is relatively straightforward: the student is given an overview of the rules of harmonization. In the interactive component, a melodic line (the top voice) and three other voices are provided. The students are then required to harmonize the melody by modifying the notes in the other three voices.

Appendix C shows the DBL definition of our sample application which consists of a text component, a music notation component and a push button. The music notation editor is used to input the solution. The `:enp` tag shown here is a PWGL specific DBL component that creates an ENP score from a special s-expressions-based format[13]. The push button, at the bottom of the document, is used by the student to request that the software evaluate the given solution. The comments and corrections are rendered directly as a part of the music notation making it easy to relate the textual instructions with the relevant parts of the solution.

This document can be exported in HTML as well as in PDF format. The interactive part is simply rendered as an image (EPS for PDF, and a PNG for HTML). Appendix D shows the application in the Safari Web browser after being exported as a HTML page.

6. FUTURE DEVELOPMENTS

The current version of DBL can't typeset or represent 'rich text'. Paragraphs are written in one style only. In the future, the paragraph parser should be extended so that users would be able insert formatting instructions alongside the bread text. Also, even though the HTML and LaTeX backends would be able to export the new 'rich text' format without problems, a new component would have to be implemented for the CAPI backend. The following shows a hypothetical syntax for indicating formatting instructions:

DBL Code

```
:body-text :ref "figure:species1" "shows a typical
example of" :italic "Species Counterpoint."
```

[5]Parallel movement is a theoretical concept that every student of music theory and composition is supposed to master and, which, to put it simply, is considered paramount for writing good sounding common practice music

The above would translate to the following piece of formatted text:

Figure **1** shows a typical example of *Species Counterpoint.*

Also, the same limitations apply to the table component. Currently, it accepts plain strings as rows and columns. A more comprehensive table component should be able to typeset not only rich text but also be able to include other components, such as images, in its definition.

Finally, currently, the syntax validation only indicates if the parsing was successful or not, therefore better syntax validation and visualization are needed. Ideally, the offending position would be shown in both the textual and visual parts.

7. CONCLUSIONS

This paper presents a Lisp-based document markup language. DBL is a standalone Lisp library that can be used for creating rich documents within the LispWorks Common Lisp Environment. DBL is comprised of the DBL language, the parser, and several backends. The purpose of the backends is to convert the DBL representation into formats such as PDF or HTML. DBL also makes it possible to define interactive documents with the help of the CAPI backend.

The primary purpose of DBL is to replace the various documentation systems used by PWGL. Eventually, DBL will be used not only for documenting the PWGL system itself, but also for documenting the source code. An automatic reference document could then be compiled by scanning the source code, compiling a DBL document and exporting it using one of the available backends.

8. ACKNOWLEDGMENT

The work of Mika Kuuskankare has been supported by the Academy of Finland (SA137619). The author would also like to thank CCRMA at Stanford University, for hosting this research project.

9. REFERENCES

[1] Cl-markdown. http://common-lisp.net/project/cl-markdown/.

[2] cl-typesetting. www.fractalconcept.com/asp/cl-typesetting.

[3] CL-WHO. http://weitz.de/cl-who/.

[4] CLiki. http://www.cliki.net.

[5] Esrap. http://nikodemus.github.com/esrap/.

[6] Markdown. http://en.wikipedia.org/wiki/Markdown.

[7] Scribble. http://www.cliki.net/Scribble.

[8] yaclml. http://common-lisp.net/project/bese/yaclml.html.

[9] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming*, 2002.

[10] M. Kuuskankare, M. Castrén, and M. Laurson. MacSet: A Free Visual Cross-Platform Pitch-Class Set Theoretical Application. In *Proceedings of International Computer Music Conference*, volume I, pages 51–54, Copenhagen, Denmark, 2007.

[11] M. Kuuskankare and M. Laurson. Annotating Musical Scores in ENP. In *International Symposium on Music Information Retrieval*, pages 72–76, London, UK, 2005.

[12] M. Kuuskankare and M. Laurson. Expressive Notation Package. *Computer Music Journal*, 30(4):67–79, 2006.

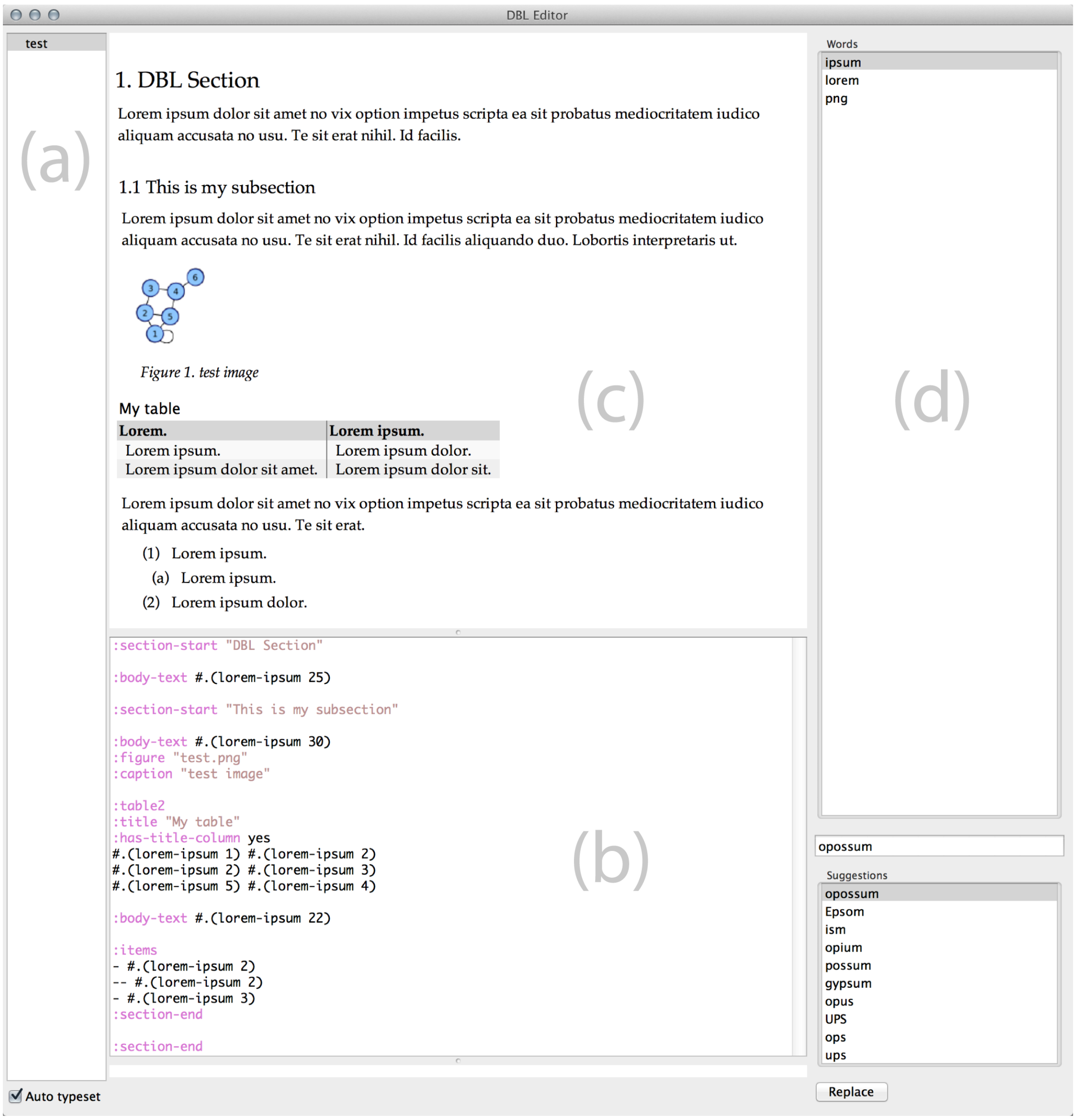
[13] M. Laurson and M. Kuuskankare. From RTM-notation to ENP-score-notation. In *Journées d'Informatique Musicale*, Montbéliard, France, 2003.

[14] M. Laurson, M. Kuuskankare, and V. Norilo. An Overview of PWGL, a Visual Programming Environment for Music. *Computer Music Journal*, 33(1):19–31, 2009.

APPENDIX

A.

The DBL Editor components: (a) the document browser, (b) the DBL code input view, (c) the active document preview typeset with the CAPI backend, and (d) an optional spell checker panel. When the 'Auto typeset' option (see the bottom left corner) is selected, then the preview is updated whenever the piece DBL code is successfully parsed.



1. DBL Section

Lorem ipsum dolor sit amet no vix option impetus scripta ea sit probatus mediocritatem iudico aliquam accusata no usu. Te sit erat nihil. Id facilis.

1.1 This is my subsection

Lorem ipsum dolor sit amet no vix option impetus scripta ea sit probatus mediocritatem iudico aliquam accusata no usu. Te sit erat nihil. Id facilis aliquando duo. Lobortis interpretaris ut.

Figure 1. test image

My table

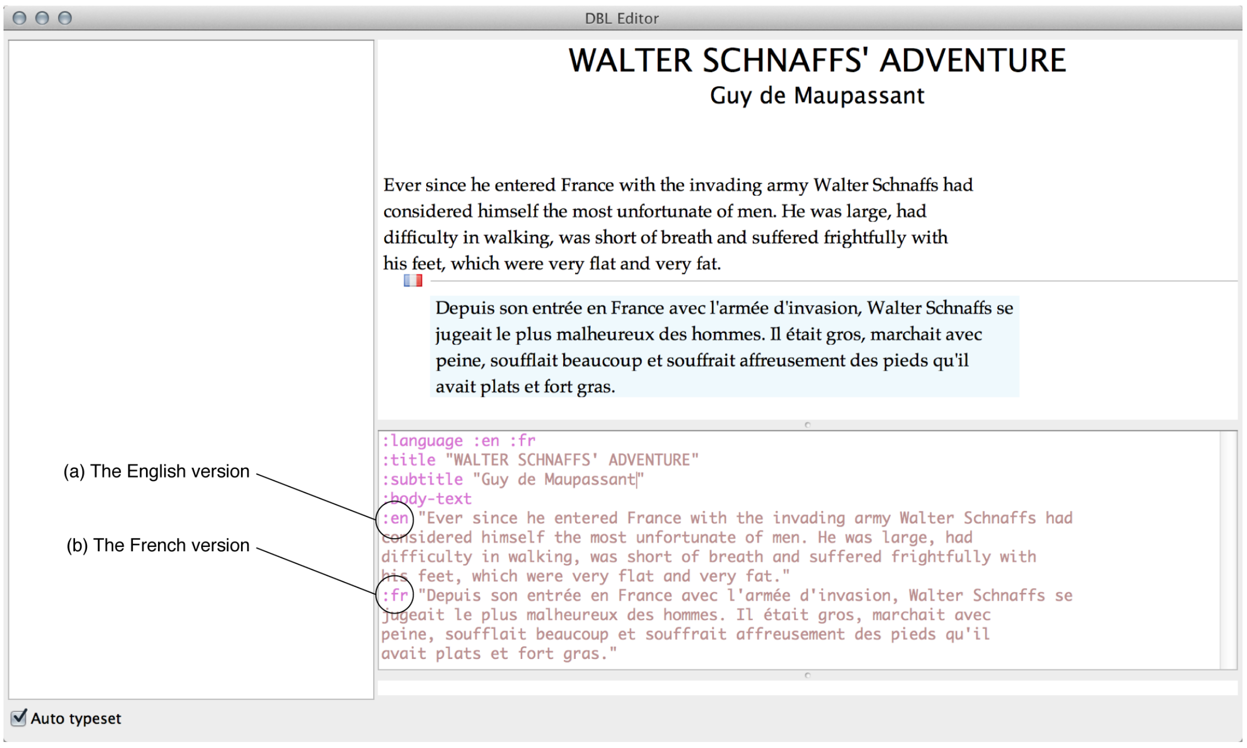
Lorem.	Lorem ipsum.
Lorem ipsum.	Lorem ipsum dolor.
Lorem ipsum dolor sit amet.	Lorem ipsum dolor sit.

Lorem ipsum dolor sit amet no vix option impetus scripta ea sit probatus mediocritatem iudico aliquam accusata no usu. Te sit erat.

(1) Lorem ipsum.
  (a) Lorem ipsum.
(2) Lorem ipsum dolor.

```
:section-start "DBL Section"

:body-text #.(lorem-ipsum 25)

:section-start "This is my subsection"

:body-text #.(lorem-ipsum 30)
:figure "test.png"
:caption "test image"

:table2
:title "My table"
:has-title-column yes
#.(lorem-ipsum 1) #.(lorem-ipsum 2)
#.(lorem-ipsum 2) #.(lorem-ipsum 3)
#.(lorem-ipsum 5) #.(lorem-ipsum 4)

:body-text #.(lorem-ipsum 22)

:items
- #.(lorem-ipsum 2)
-- #.(lorem-ipsum 2)
- #.(lorem-ipsum 3)
:section-end

:section-end
```

B.
A paragraph of text written in two different languages: (a) in English using the :en tag, and (b) in French using the :fr tag.



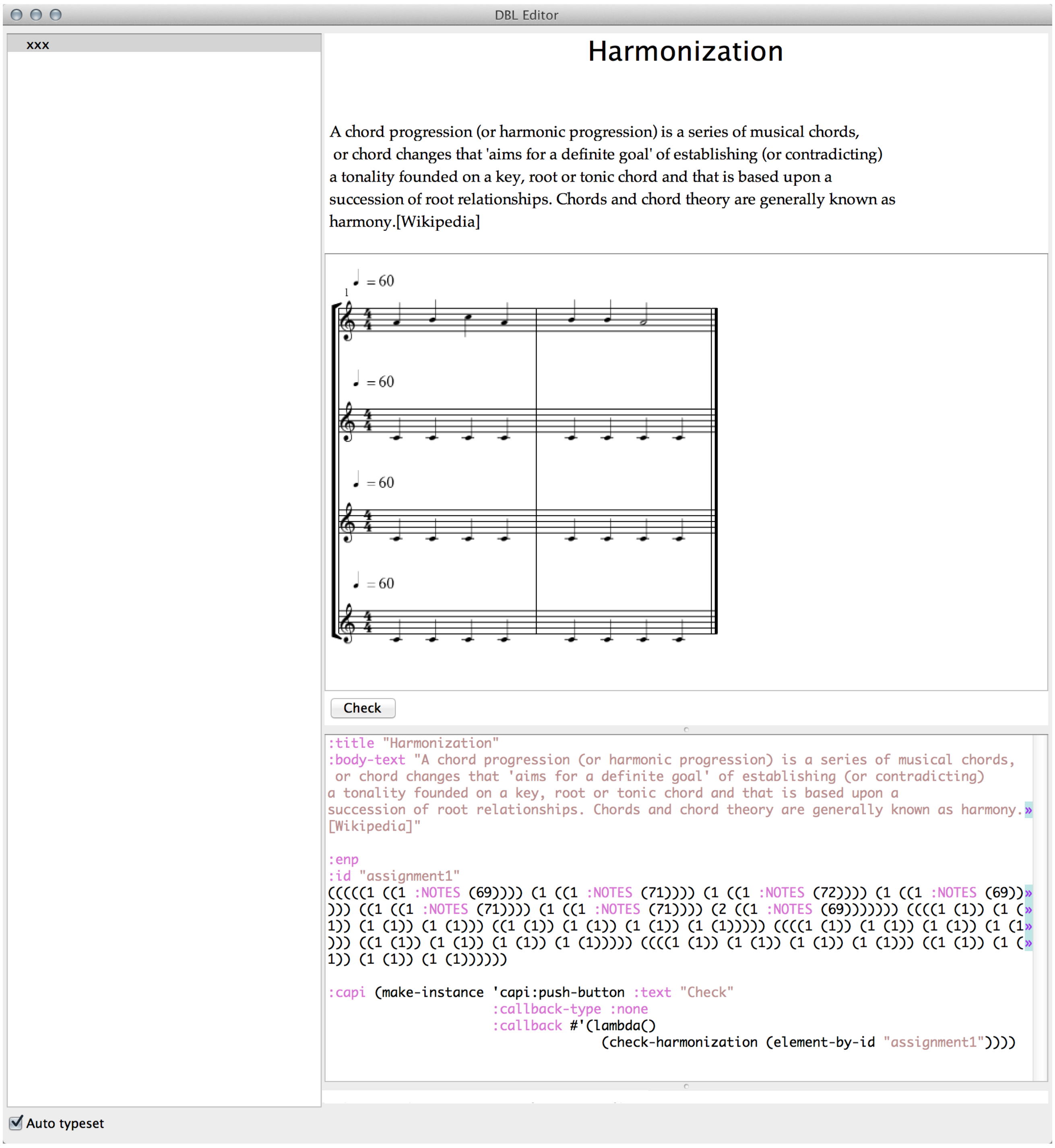
WALTER SCHNAFFS' ADVENTURE
Guy de Maupassant

Ever since he entered France with the invading army Walter Schnaffs had considered himself the most unfortunate of men. He was large, had difficulty in walking, was short of breath and suffered frightfully with his feet, which were very flat and very fat.

Depuis son entrée en France avec l'armée d'invasion, Walter Schnaffs se jugeait le plus malheureux des hommes. Il était gros, marchait avec peine, soufflait beaucoup et souffrait affreusement des pieds qu'il avait plats et fort gras.

```
:language :en :fr
:title "WALTER SCHNAFFS' ADVENTURE"
:subtitle "Guy de Maupassant"
:body-text
:en "Ever since he entered France with the invading army Walter Schnaffs had
considered himself the most unfortunate of men. He was large, had
difficulty in walking, was short of breath and suffered frightfully with
his feet, which were very flat and very fat."
:fr "Depuis son entrée en France avec l'armée d'invasion, Walter Schnaffs se
jugeait le plus malheureux des hommes. Il était gros, marchait avec
peine, soufflait beaucoup et souffrait affreusement des pieds qu'il
avait plats et fort gras."
```

C.

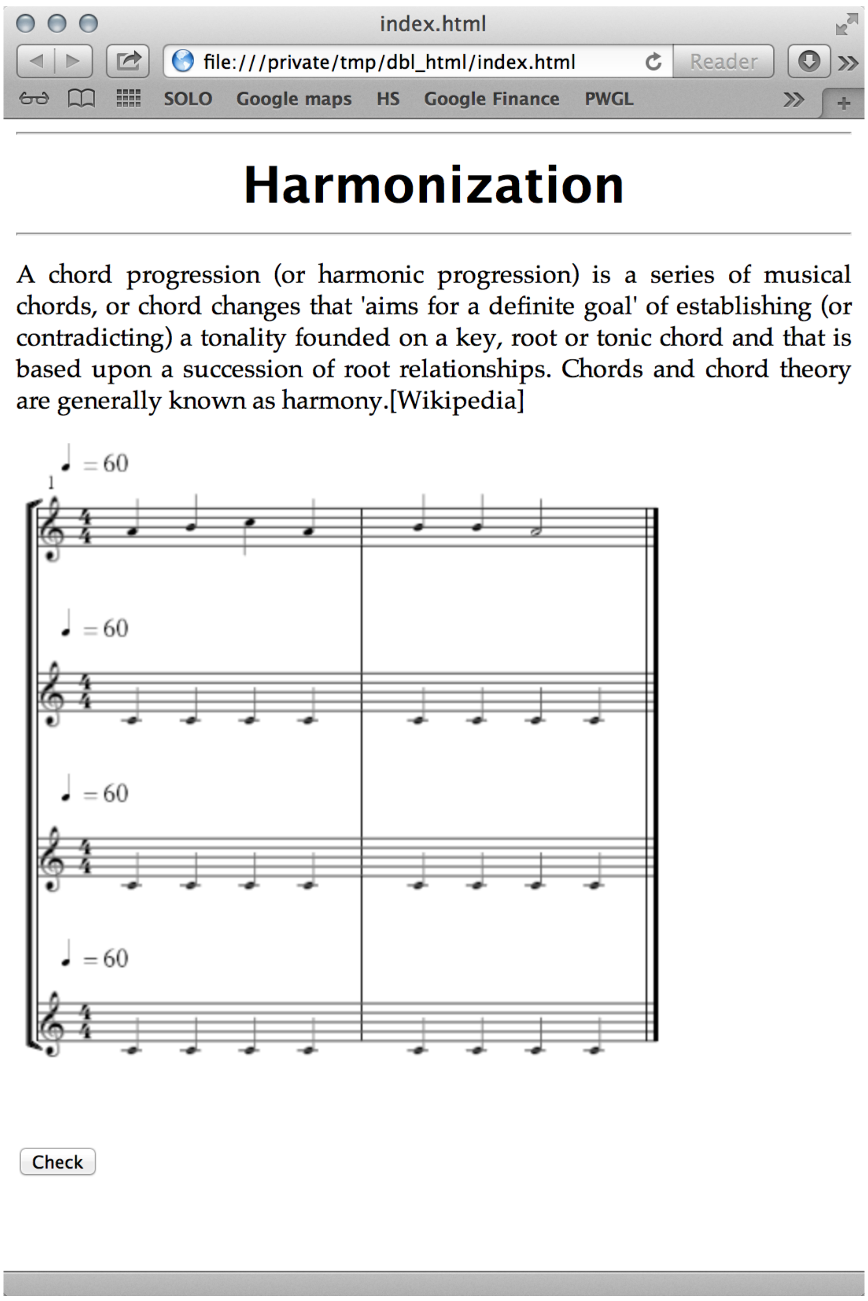


Harmonization

A chord progression (or harmonic progression) is a series of musical chords,
or chord changes that 'aims for a definite goal' of establishing (or contradicting)
a tonality founded on a key, root or tonic chord and that is based upon a
succession of root relationships. Chords and chord theory are generally known as
harmony.[Wikipedia]

```
:title "Harmonization"
:body-text "A chord progression (or harmonic progression) is a series of musical chords,
 or chord changes that 'aims for a definite goal' of establishing (or contradicting)
a tonality founded on a key, root or tonic chord and that is based upon a
succession of root relationships. Chords and chord theory are generally known as harmony.»
[Wikipedia]"

:enp
:id "assignment1"
(((((1 ((1 :NOTES (69)))) (1 ((1 :NOTES (71)))) (1 ((1 :NOTES (72)))) (1 ((1 :NOTES (69))»
))) ((1 ((1 :NOTES (71)))) (1 ((1 :NOTES (71)))) (2 ((1 :NOTES (69))))))) ((((1 (1)) (1 (»
1)) (1 (1)) (1 (1))) ((1 (1)) (1 (1)) (1 (1)) (1 (1))))) ((((1 (1)) (1 (1)) (1 (1)) (1 (1»
))) ((1 (1)) (1 (1)) (1 (1)) (1 (1))))) ((((1 (1)) (1 (1)) (1 (1)) (1 (1))) ((1 (1)) (1 (»
1)) (1 (1)) (1 (1))))))

:capi (make-instance 'capi:push-button :text "Check"
                     :callback-type :none
                     :callback #'(lambda()
                                  (check-harmonization (element-by-id "assignment1"))))
```

D.



Harmonization

A chord progression (or harmonic progression) is a series of musical chords, or chord changes that 'aims for a definite goal' of establishing (or contradicting) a tonality founded on a key, root or tonic chord and that is based upon a succession of root relationships. Chords and chord theory are generally known as harmony.[Wikipedia]

Infinite transducers on terms denoting graphs

Bruno Courcelle
courcell@labri.fr

Irène A. Durand
idurand@labri.fr

LaBRI, CNRS, Université de Bordeaux, Talence, France

ABSTRACT

Starting from Courcelle's theorem which connects the problem of verifying properties of graphs of bounded clique-width with term automata, we have developed the Autograph Lisp library[1] which provides automata for verifying graph properties [2]. Because most of these automata are huge, fly automata have been introduced in the underlying library Autowrite and have become the default type of automata [11]. By default, the operations on automata are now performed on fly automata.

This article shows how fly automata can be generalized to *attributed fly automata* and finally to *transducers*.

We apply these concepts in the domain of graph theory. We present some computations performed by transducers on terms representing graphs. This transducer approach for computing graph values is an alternative to the classical algorithms of graph theory.

Categories and Subject Descriptors

D.1.1 [**Software**]: Programming Techniques, Applicative (Functional) Programming; F.1.1 [**Theory of Computation**]: Models of Computation, Automata; G.2.2 [**Mathematics of Computing**]: Graphs Theory, Graph Algorithms

Keywords

Term automata, Term transducers, Lisp, graphs

1. INTRODUCTION

At ELS2010 [2], we showed that bottom-up term automata could be used to verify monadic second order properties on graphs of bounded clique-width which can be represented by terms. Although finite, these automata are often so large that their transitions table cannot be built.

To solve this problem, we have introduced automata called *fly automata* which were first presented at ELS2011 [11]. In such automata, instead of the transition function being represented as a set of values, it is represented as the code of a computable function. In this setting, the states of an automaton need not be listed; a finite subset of the whole set of states is produced *on the fly* by the transition function during the run of the automaton on a term.

[1]itself based on the Autowrite Lisp library

Fly automata solve the problem of representing huge finite automata but also yield new perspectives as they need not be finite; they may be infinite in two ways: they may have an infinite denumerable signature and an infinite denumerable set of states.

Fly automata are nicely implemented in Lisp, the core of the automaton being its transition function. Also Lisp conditions are used to detect early failure.

In the graph framework, we may obtain fly automata working for any clique-width with a signature containing an infinite number of constant symbols. Also we may use natural integers as states to count for instance the number of vertices of a graph.

Usually, a term automaton has a set of transitions and a set of states, a subset of which are the final states. A fly automaton has a transition function and a final state predicate which says whether a state is final or not. When running an automaton on a term, one gets a *target* which may be just one state if the automaton is deterministic or a set of states otherwise; the target is final if it is a final state (deterministic case) or contains a final state (non deterministic case). The term is recognized when the target is final.

As we have no limit on the number of states, we may extend our fly automata to compute *attributed states* which are states associated with an attribute which is computed along the run.

Attributes may be of many different types: they are often integers, sets of positions, symbols, terms and tuples, sets, and multisets of the previous types.

In the framework of graphs, attributes may be color assignments (colorings) or subset assignments, number of colorings, number of subset assignment, etc. An *attributed automaton* is just an automaton whose transition function is completed with an *attribute function* which synthetizes the attribute of the states. The attribute function computes the new attribute from the attribute of the arguments and the function symbol. When the automaton is non deterministic, the same state may be obtained by several computations but with different attributes; in that case, we must provide a function to *combine* the attributes for this state. We shall refer to this function as the `combine-fun`.

In this paper, we extend the concept of fly automaton to *attributed fly automaton* and finally to *fly transducer*. A fly transducer has an output function instead of a final state predicate. As for an automaton, the run of a transducer on a term produces a target. The output function is applied to the final target for the final result. In our case, a transducer will be an attributed automaton with an output function that will be applied to the attribute of the final target.

A fly automaton may be seen as a particular case of a transducer where the output function is the final state predicate.

The `Autowrite` software [13] entirely written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [9]. For this purpose, it implements classical finite term (tree) automata. In the first implementation, just the emptiness problem (does the automaton recognize the empty language) was used and implemented.

In subsequent versions [10], the implementation was continued in order to provide a substantial library of operations on term automata. The next natural step was to try to solve concrete problems using this library and to test its limits.

Starting from Courcelle's theorem [7] which connects the problem of verifying properties of graphs of bounded clique-width with term automata, we have developed the `Autograph` library [14] (based on `Autowrite`) which provides automata for verifying graph properties [2].

Because most of these automata are huge, we introduced fly automata into `Autowrite` and made them the default type of automata [11]. By default, the operations on automata are performed on the fly automata. The traditional table-automata are just compiled versions of finite fly automata.

The purpose of this article is:

- to show how fly automata can be generalized to attributed fly automata and finally to transducers,
- to describe part of the implementation,
- and to present some computations performed by such transducers on terms and terms representing graphs.

This transducer approach for computing graph values is an alternative to the classical algorithms of graph theory.

One advantage of the use of automata or transducers is that, using inverse-homomorphisms, we can easily get algorithms working on induced subgraphs from the ones working on the whole graph which is most often not feasible with classical algorithms of graph theory.

Some graph coloring problems will be used as examples throughout the paper.

2. PRELIMINARIES

We recall some basic definitions concerning terms and how terms may be used to represent graphs of bounded clique-width.

2.1 Signature and terms

We consider a signature $\mathcal{F}$ (set of symbols with fixed arity).

EXAMPLE 2.1. *Let $\mathcal{F}$ be a signature containing the symbols* `{a, b, add_a_b, ren_a_b, ren_b_a, oplus}` *with*

$$\boxed{\begin{array}{l}\mathsf{arity}(\mathtt{a}) = \mathsf{arity}(\mathtt{b}) = 0 \quad \mathsf{arity}(\mathtt{oplus}) = 2\\ \mathsf{arity}(\mathtt{add\_a\_b}) = \mathsf{arity}(\mathtt{ren\_a\_b}) = \mathsf{arity}(\mathtt{ren\_b\_a}) = 1\end{array}}$$

In Section 3, we show that this signature is suitable for writing terms representing graphs of clique-width at most 2.

We denote the subset of symbols of $\mathcal{F}$ with arity n by $\mathcal{F}_n$. So $\mathcal{F} = \bigcup_n \mathcal{F}_n$. By $\mathcal{T}(\mathcal{F})$, we denote the set of (ground) terms built upon the signature $\mathcal{F}$.

EXAMPLE 2.2. *t_1, t_2, t_3 and t_4 are terms built with the signature $\mathcal{F}$ of Example 2.1.*

```
t1  =  oplus(a,b)
t2  =  add_a_b(oplus(a,oplus(a,b)))
t3  =  add_a_b(
         oplus(a,oplus(a,oplus(b,b))))
t4  =  add_a_b(
         oplus(a,
           ren_a_b(add_a_b(oplus(a,b)))))
```

In Table 1, we see their associated graphs. The connection between terms and graphs will be described in Section 3.2.

3. APPLICATION DOMAIN

Part of this work will be illustrated in the framework of graphs of bounded clique-width. In this section, we present the connection between graphs and terms. First we define the graphs.

3.1 Graphs as a logical structure

We consider finite, simple, loop-free undirected graphs (extensions are easy)[2]. Every graph can be identified with the relational structure $\langle \mathcal{V}_G, edg_G \rangle$ where $\mathcal{V}_G$ is the set of vertices and edg_G the binary symmetric relation that describes edges: $edg_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ and $(x, y) \in edg_G$ if and only if there exists an edge between x and y.

Properties of a graph G can be expressed by sentences of relevant logical languages. Monadic Second order Logic is suitable for expressing many graph properties like k-colorability, acyclicity (no cycle), k-acyclic-colorability, ...

3.2 Term representation of graphs of bounded clique-width

DEFINITION 1. *Let $\mathcal{L}$ be a finite set of vertex labels also called* ports *and let us consider graphs G such that each vertex $v \in \mathcal{V}_G$ has a label $label(v) \in \mathcal{L}$. The operations on graphs are:*

- `oplus`*: the union of disjoint graphs,*

for every pair of distinct vertex labels $(a, b) \in \mathcal{L} \times \mathcal{L}$:

[2]We consider such graphs for simplicity of the presentation but we can also work with directed graphs, loops, labeled vertices and edges. A loop is an edge connecting one single vertex.

- *unary edge addition operations* `add_a_b`[3] *that add the missing edges between every vertex labeled* a *to every vertex labeled* b*,*
- *unary relabeling operations* `ren_a_b` *that rename* a *to* b*, and*

for every vertex label $a \in \mathcal{L}$*,*

- *constants* `a` *such that the term* `a` *denotes a graph with a single vertex labeled by* a *and no edge.*

Let $\mathcal{F}_{\mathcal{L}}$ *be the set of these operations and constant symbols.*

Every term $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ *defines a graph* $G(t)$ *whose vertices are the leaves of the term* t*. Note that because of the relabeling operations, the labels of the vertices in the graph* $G(t)$ *may differ from the ones specified in the leaves of the term and that several terms may represent the same graph up to isomorphism.*

A graph has clique-width *at most* k *if it is defined by some* $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ *with* $|\mathcal{L}| \leq k$*. The clique-width of a graph is the minimal such* k*. We shall abbreviate clique-width by* cwd.

Examples of terms with their associated graph are given in Table 1.

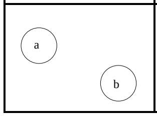
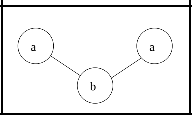
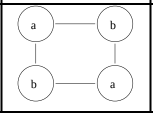
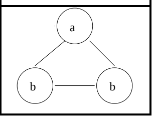
t_1	t_2	t_3	t_4
			

Table 1: The graphs corresponding to the terms of Example 2.2

3.3 Clique-width of some well-known graphs

The problem of finding a *decomposition* of a graph (*i.e.* a term representing the graph) with a minimal number of labels (so the clique-width) is NP-complete [15].

However, an approximation can always be found, the worst approximation being the one using as many labels as vertices in the graph and as many `add_a_b` operations as edges in the graph.

For instance, the graph corresponding to t_3 of Table 1 can be decomposed as

```
add_a_d(
 add_c_d(
  add_b_c(
   add_a_b(
    oplus(a,oplus(b,oplus(c,d))))))
```

which uses 4 ports labels.

However, the clique-width parameter being crucial in our algorithm, it is important to minimize the number of port labels. So term t_3 which uses 2 labels would be preferable to the term above.

For some classical family of graphs, the clique-width is known. For instance, cliques K_n have clique-width 2, for all $n > 1$. A term k_n representing K_n is recursively given by:

$$\begin{cases} k_1 = \texttt{a} \\ k_n = \texttt{ren\_b\_a(add\_a\_b(oplus(}k_{n-1}\texttt{,b)))} \end{cases}$$

P_n graphs (chains of n nodes) have clique-width 3 for $n > 3$. A term p_n representing a graph P_n is recursively given by:

$$\begin{cases} p_1 = \texttt{b} \\ p_n = \texttt{ren\_c\_b(ren\_b\_a(add\_b\_c(oplus(}p_{n-1}\texttt{,c)))} \end{cases}$$

Rectangular grids $n \times m$ with $m < n$ have clique-width $m + 2$. Square grids $n \times n$ have clique-width $n + 1$; the latest decomposition is a bit tricky [17].

3.4 Representation of colored graphs

To deal with colored graphs, we use a modified constant signature. If we are dealing with k colors then every constant `c` yields k constants `c~1`, ..., `c~k`. In a term, the constant `c~i` means that the corresponding vertex is colored with color i.

EXAMPLE 3.1. *For instance, the term*
`add_a_b(oplus(a~1,oplus(b~2,oplus(a~1,b~2))))`
represents a proper 2*-colored version of term* t_3 *of Example 2.2.*

3.5 Representation of sets of vertices

To deal with graphs with identified subsets of vertices, we also use a modified constant signature. If we are dealing with m subsets $V_1, \ldots, V_m$ then every constant `c` yields 2^m constants of the form `c^w` where w is a bit vector $b_1 \ldots b_m$ such that $b_i = 1$ if the corresponding vertex belongs to V_i, $b_i = 0$ otherwise.

4. TERM AUTOMATA

4.1 Finite term automata

We recall the definition of finite term automaton. Much more information can be found in the on-line book [1].

DEFINITION 2. *A (finite bottom-up)* term automaton[4] *is a quadruple* $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ *consisting of a finite signature* $\mathcal{F}$*, a finite set* Q *of states, disjoint from* $\mathcal{F}$*, a subset* $Q_f \subseteq Q$ *of final states, and a set of transitions rules* Δ*. Every transition is of the form* $f(q_1, \ldots, q_n) \to q$ *with* $f \in \mathcal{F}$*,* $\mathsf{arity}(f) = n$ *and* $q_1, \ldots, q_n, q \in Q$*.*

Term automata recognize *regular* term languages[20]. The class of regular term languages is closed under the Boolean operations (union, intersection, complementation) on languages which have their counterpart on automata.

EXAMPLE 4.1. *A graph is* stable *if it has no edges. The automaton* `2-STABLE` *of Figure 1 recognizes stable graphs of clique-width* 2*. The states* `<a>`*,* `<b>`*,* `<ab>` *mean that the graph contains no edge and respectively at least a vertex labeled* `a`*, at least a vertex labeled* `b`*, at least a vertex labeled* `a` *and a vertex labeled* `b`*; they are all final states. The state* `error` *is the only non final*

[3]for the oriented case both `add_a_b` and `add_b_a` are used; for the unoriented case, we may assume a total order on the port labels and use the `add_a_b` such that `a` < `b`.

[4]Term automata are frequently called tree automata, but it is not a good idea to identify trees, which are particular graphs, with terms.

```
Automaton 2-STABLE
Signature: a b ren_a_b:1 ren_b_a:1 add_a_b:1 oplus:2*
States: <a> <b> <ab> error
Final States: <a> <b> <ab>

Transitions  a -> <a>                    b -> <b>
  add_a_b(<a>) -> <a>                    add_a_b(<b>) -> <b>
  ren_a_b(<a>) -> <b>                    ren_b_a(<a>) -> <a>
  ren_a_b(<b>) -> <b>                    ren_b_a(<b>) -> <a>
  ren_a_b(<ab>) -> <b>                   ren_b_a(<ab>) -> <a>
  oplus*(<a>,<a>) -> <a>                 oplus*(<b>,<b>) -> <b>
  oplus*(<a>,<b>) -> <ab>                oplus*(<b>,<ab>) -> <ab>
  oplus*(<a>,<ab>) -> <ab>               oplus*(<ab>,<ab>) -> <ab>
  add_a_b(<ab>) -> error                 ren_a_b(error) -> error
  add_a_b(error) -> error                ren_b_a(error) -> error
  oplus*(error,q) -> error for all q
```

Figure 1: Automaton recognizing stable graphs

state; it means that an edge has been found. The rule which triggers the first `error` *state is the* `add_a_b(<ab>) -> error` *rule; such operation adds an edge between vertices labeled* `a` *and vertices labeled* `b`*. We shall see later that this automaton is in fact the compiled version of a finite fly automaton.*

From this automaton, we can derive another automaton for deciding whether a subgraph induced by a subset of vertices V_1 is stable. We have seen that membership of a vertex to a subset of vertices V_1 is expressed with a bit added to the constants. `a^1` *represents a vertex labeled* `a` *belonging to V_1 while* `a^0` *represents a vertex labeled* `a` *not belonging to V_1. The term*
`add_a_b(oplus(a^1,oplus(b^0,oplus(a^1,b^0))))`
represents the same graph as t_3 in Example 2.2 but with the two vertices labeled `a` *in V_1.*

To the previous automaton, we add the symbol `@` *for representing an empty graph (so a stable graph), the state* `#f` *for representing a neutral final state (which will be used as long as no vertex in V_1 has been found) and the rules*

```
@ -> #f
zzz_x_y(#f) -> #f
oplus(q,#f) -> q
```

for every `zzz` *$\in \{add, ren\}$, every $x, y \in \{a, b\}$ such that $x \neq y$ and every* `q` *state of* `2-STABLE` *and consider the homomorphism h such that*

$$h(\texttt{a^1})=\texttt{a}$$
$$h(\texttt{a^0})=\texttt{@}$$
$$h(\texttt{b^1})=\texttt{b}$$
$$h(\texttt{b^0})=\texttt{@}$$

and $h(f) = f$ for every non constant symbol.

Applying h^{-1} to the automaton `2-STABLE` *yields an automaton which recognizes graphs such that V_1 is stable. Only the constant rules differ*

```
a^1 -> <a>
b^1 -> <b>
a^0 -> #f
b^0 -> #f
```

To distinguish these finite automata from the fly automata defined in Subsection 4.2 and as we only deal with terms in this paper we shall refer to the term automata defined in Definition 2 as *table-automata*.

4.2 Fly term automata

DEFINITION 3. *A* fly term automaton *(*fly automaton *for short) is a triple $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{fs})$ where*

- *$\mathcal{F}$ is a countable signature of symbols with a fixed arity,*
- *δ is a computable transition function,*

$$\begin{array}{rccc} \delta: & \bigcup_n \mathcal{F}_n \times Q^n & \to & Q \\ & f q_1 \ldots q_n & \mapsto & q \end{array}$$

  where Q is a countable set of states, disjoint from $\mathcal{F}$,
- *fs is the final state predicate*

$$\mathsf{fs}: \quad Q \quad \to \quad Boolean$$

  which indicates whether a state is final or not.

Note that, both the signature $\mathcal{F}$ and the set of states Q may be infinite. A fly automaton is *finite* if both its signature and its set of states are finite.

Operations on term languages like Boolean operations, homomorphisms and inverse-homomorphisms have their counterpart on fly automata [3, 4]. For instance, the union of two fly automata recognizes the union of the two languages recognized by the automata.

We use the term *basic* for fly automata that are built from scratch in order to distinguish them from the ones that are obtained by combinations of existing automata using the operations cited in the above theorem. We call the latter *composed* fly automata.

The *run* of an automaton on a term labels the nodes of the term with the state(s) reached at the corresponding subterm. The run goes from bottom to top starting at the leaves.

In `Autowrite`, this is implemented via the
`compute-target(term automaton)`
operation which, given a term and an automaton, returns the target (a single state if the automaton is deterministic or a container of states otherwise).

A term is *recognized* by the automaton when after the run of the automaton on the term, a final state is obtained at the root.

```
(defmethod stable-transitions-fun
    ((root constant-symbol) (arg (eql nil)))
  (let ((port (port-of root)))
    (when (or (not *ports*)
      (member port *ports*))
     port
     (make-stable-state
      (make-ports-from-port port)))))

(defmethod stable-transitions-fun
    ((root abstract-symbol) (arg list))
  (common-transitions-fun root arg))
```

Figure 2: Transition function for constants

In Autowrite, this is implemented by the
`recognized-p(term automaton)`
operation which returns true if at least one state in the target is final according to the final state predicate of the automaton.

In fact, we have an intermediate operation
`compute-final-target(term automaton)` which returns the *final target* that is the target without non final states. Then the `recognized-p(term automaton)` operation is implemented by checking whether the final target is empty.

4.3 Examples with the stability property

We can create an infinite fly automaton that verifies that a graph is stable for *any* clique-width.

We create it as a basic automaton (in the sense given in Section 4.2). This means that we must define the structure of the states for this automaton and the transition function that computes the states. This automaton is deterministic.

Its states are of uniform type; the state computed at the root of a term represents the set of port labels encountered so far.

```
(defclass stable-state (graph-state)
  ((ports :type ports
          :initarg :ports
          :reader ports)))
```

The transition function is `stable-transitions-fun`.

For a constant symbol a, a `stable-state` is created with `ports` being the singleton $\{a\}$. This is shown in Figure 2.

For the non constant symbols, the transition function calls
`common-transitions-fun`
which switches to a call to `graph-oplus-target`,`graph-ren-target` or
`graph-add-target` according to the symbol (see Figure 3).

If we fixed the clique-width cwd, then we could compile this fly automaton to a minimal table-automaton with 2^{cwd} states. The automaton 2-STABLE of Figure 1 is in fact the compiled version of the fly version with $cwd = 2$.

Figure 4 shows that the graph corresponding to the term t_3 is not stable. Figure 5 that the subgraph induced by the vertices with port `a` is stable. The automaton used is obtained by inverse homomorphism as described in Example 4.1.

```
(defmethod graph-ren-target
    (a b (so stable-state))
  (make-stable-state
   (ports-subst
    b a
    (ports so))))

(defmethod graph-add-target
    (a b (so stable-state))
  (let ((ports (ports so)))
    (unless (and
     (ports-member a ports)
     (ports-member b ports))
      so)))

(defmethod graph-oplus-target
    ((s1 stable-state) (s2 stable-state))
  (make-stable-state
   (ports-union (ports s1) (ports s2))))
```

Figure 3: Transition function for stability

```
AUTOGRAPH> (recognized-p
    *t3*
    (stable-automaton))
NIL
NIL
AUTOGRAPH> *t3*
add_a_b(oplus(a,oplus(a,oplus(b,b))))
AUTOGRAPH> (recognized-p
    *t3*
    (stable-automaton))
NIL
NIL
```

Figure 4: Examples with stability

4.4 Examples with graph-colorings

In graph theory, a graph is k-*colored* if its vertices are colored with k colors. The coloring is *proper* if two adjacent vertices do not have the same color. The graph represented by the term given in Example 3.1 has a proper 2-coloring.

4.4.1 Graph coloring verification

For a fixed number of colors k, we can create an infinite fly automaton which verifies that a graph has a proper k-coloring for any clique-width.

```
AUTOGRAPH> *s3*
add_a_b(oplus(a^1,oplus(a^1,oplus(b^0,b^0))))
AUTOGRAPH> (recognized-p
    *s3*
    (nothing-to-x1
     (stable-automaton)))
T
!<{a}>
```

Figure 5: Stability of induced subgraph

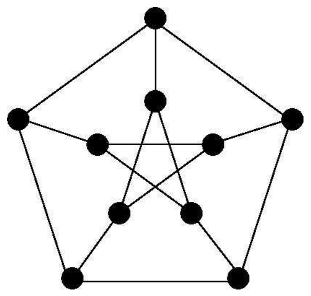


Figure 6: Petersen's Graph

The constants have colors which are positive integers in $[1, k]$; the constant `a~i` means that this vertex has color `i` for $i \in [1, k]$.

We create it as a basic automaton (in the sense given in Section 4.2). This means that we must define the structure of the states for this automaton and the transition function that computes the states. This automaton is deterministic.

Its states are of uniform type; the state computed at the root of a term represents a function which, given a constant name `c`, gives the set of color numbers appearing on leaves `c~i` in the term.

```
(defclass colors-state (graph-state)
  ((color-fun :initarg :color-fun
              :reader color-fun)))
```

For instance, the `color-fun` of the state reached at the root of term `oplus(a~1,oplus(b~1,a~2))` should return `{1,2}` when applied to `a`, `{1}` when applied to `b`, and the empty set when applied to any other constant label.

The transition function of the automaton is described a little further.

If we fixed the clique-width cwd, then we could compile this fly automaton to a table-automaton with $2^{2^{cwd}} - 1$ states. That would give $2^{2^{\times 6}} - 1$ states in order to get an automaton able to work on Petersen's graph (see Figure 6) for which our best decomposition has $cwd = 6$. The term representing Petersen's graph has 37 nodes and depth 28.

We can use that automaton, to verify that some graphs are properly k-colored. We shall see later that by doing a color-projection (erasing the colors) of this automaton, we obtain a non deterministic automaton which recognizes graphs that are k-colorable. The two rules `a~1 -> q1`, `a~2 -> q2` would become the non deterministic rule `a -> o{q1,q2}` via the color-projection.

In Figure 7, we show the function `colored-automaton` which returns such an infinite automaton. When the optional `cwd` (clique-width) parameter is omitted, the resulting automaton has an infinite signature and an infinite number of states. The crucial part to implement is the operation `colored-transition-fun` which corresponds to the transition function of the automaton. When the optional parameter cwd is zero, then the automaton has an infinite signature and works on terms of any clique-width. Otherwise ($cwd > 0$), there is a finite number of port labels so that the signature has a finite number of constants and the automaton should not recognize constants with a port $>= cwd$. The `*ports*` special variable records the list of autorized labels when finite and is `NIL` otherwise; it is used when the transition function is applied to a constant.

```
(defun colored-automaton
    (k &optional (cwd 0))
  (make-fly-automaton
   (setup-color-signature cwd k)
   (lambda (root states)
     (let ((*ports* (port-iota cwd))
           (*colors* (color-iota k)))
       (colored-transitions-fun
        root states)))
   :name (format
          nil
          "~A-COLORED-~A" cwd k))))
```

Figure 7: Automaton for verifying the coloring of a graph

For a colored constant `c~i`, the transition function returns a `colors-state` whose `color-fun` gives the singleton $\{$`i`$\}$ for `c` and the empty set for every other constant.

```
(defmethod colored-transitions-fun
    ((root color-constant-symbol)
     (arg (eql nil)))
  (let ((port (port-of root)))
    (when (or (endp *ports*)
              (member port *ports*))
      (let* ((color (symbol-color root))
             (color-fun
               (add-color-to-port
                 color
                 port
                 (make-empty-color-fun))))
        (make-colors-state color-fun)))))
```

For the non constant symbols, as for the stable case, the transition function calls the method `graph-oplus-target`,
`graph-ren-target` or
`graph-add-target`
according to the symbol.

For the disjoint union operation `oplus`, the `color-fun` of the new state returns the union of the `color-fun` of the children. There may be no failure. The function `graph-oplus-target` implements this operation.

```
(defmethod graph-oplus-target
    ((s1 colors-state) (s2 colors-state))
  (make-colors-state
   (merge-color-fun
    (color-fun s1)
    (color-fun s2))))
```

The only operations which may lead to a failure are the `add_a_b` operations, because they may connect two vertices which have the same color; in that case, the transition function returns `NIL`. This failure should be transmitted directly to the root of the term via `Lisp` conditions.
The function `graph-add-target` implements this operation.

```
(defmethod graph-add-target
    (a b (colors-state colors-state))
  (let ((color-fun
         (color-fun colors-state)))
    (unless (intersection
              (get-colors a color-fun)
              (get-colors b color-fun))
      colors-state)))
```

In Figure 8, we call the function to obtain an infinite automaton that verifies whether a graph of any clique-width has a proper 2-coloring.

```
AUTOGRAPH> (setf *2-colored*
             (colored-automaton 2))
0-COLORED-2 ;; deterministic
AUTOGRAPH> (compute-target
             (input-term "a~1")
             *2-colored*)
<a:1> ;; one state
```

Figure 8: Automaton for coloring verification

Finally, Figure 10 shows the use of this automaton on a 2-colored graph. We use term t_3 of Table1 (which corresponds to a cycle of size 4) with two different colorings:
one is proper (`*t3_1*` see Figure 9), the other one is not (`*t3_2*`).

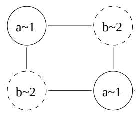


Figure 9: A proper 2-coloring of t_3

```
AUTOGRAPH> *t3_1*
add_a_b(
 oplus(a~1,oplus(a~1,oplus(b~2,b~2))))
AUTOGRAPH> (recognized-p
            *t3_1*
            *2-colored*)
T
<a:1 b:2>
AUTOGRAPH> *t3_2*
add_a_b(
 oplus(a~1,oplus(a~2,oplus(b~2,b~1))))
AUTOGRAPH> (recognized-p
            *t3_2*
            *2-colored*)
NIL
NIL
```

Figure 10: Verification of the coloring of a graph

4.4.2 *Graph k-colorability*

To obtain an automaton for deciding whether an uncolored graph is k-colorable, one must apply a projection (inverse homomorphism) which removes the colors from the constants to the previous automaton. The result is a non deterministic automaton.

```
AUTOGRAPH> (setf
             *2-colorability*
             (color-projection-automaton
              *2-colored* 2))
fly-asm(0-COLORED-2) ;; non deterministic
```

Now, we run the automaton on some terms.

```
AUTOGRAPH> (compute-target
             (input-term "a")
             *2-colorability*)
o{<a:1> <a:2>} ;; 2states
```

We verify that a clique of size 3 is not 2-colorable:

```
AUTOGRAPH> (recognized-p
             (graph-kn 3) ;; clique of size 3
             *2-colorability*)
NIL
NIL
```

but that the graph corresponding to t_3 is 2-colorable:

```
AUTOGRAPH> *t3*
add_a_b(oplus(a,oplus(a,oplus(b,b))))
AUTOGRAPH> (recognized-p
             *t3* *2-colorability*)
T
o{<a:1 b:2> <a:2 b:1>} ;; 2 states
```

It is nice to know that a graph is k-colorable but it would be even nicer to effectively find a proper coloring (or all proper colorings) of a graph. A simple fly automaton is not enough for that, as it just gives a boolean answer. In the next section, we shall show how a fly automaton may be enhanced in order to compute more interesting answers than boolean values. In particular, we shall be able to compute or enumerate the proper colorings of a graph.

5. FLY TRANSDUCERS

Because, the number of states of a fly automaton may be infinite, we may associate *attributes* to the states of the fly automata in order to compute more complicated information than just states.

5.1 Attributed fly automata

An *attributed* fly automaton $\mathcal{B}$ is a fly automaton which is based on another automaton $\mathcal{A}$.

The transition function of $\mathcal{B}$ is the one of $\mathcal{A}$ enhanced in order to compute an attribute associated with each state. So the automaton computes attributed states instead of states. Attributed states are a particular kind of state. They are states that contain states. In `Autowrite` we already had states that contain a state; for instance *indexed-states* for computing disjoint unions of automata. The `in-state` class captures that behaviour.

```
(defclass in-state-mixin ()
  ((in-state :initform nil
     :initarg :in-state
             :accessor in-state)))

(defclass in-state
    (in-state-mixin abstract-state) ())
```

Then the class for attributed-states is derived from the `in-state` class.

```
(defclass attributed-state (in-state)
  ((state-attribute
    :initarg :state-attribute
    :accessor state-attribute)
   (combine-fun :initarg :combine-fun
                :reader combine-fun)))
```

In the deterministic case, instead of computing just the state q, it computes an *attributed-target* which is just an *attributed-state* $[q, a]$ where q is the state computed by $\mathcal{A}$ and a the attribute.

In the non-deterministic case, instead of computing a set of states $\{q_1, \dots, q_p\}$, it computes an attributed target which is a set of attributed states

$\{[q_1, a_1] \dots, [q_p, a_p]\}$.

The final state predicate must be extended to work on attributed states: an attributed state $[q, a]$ is final if the state q is.

At each node of the term, the attribute (or the attributes in the non deterministic case) are computed from the ones obtained at the child nodes.

In the deterministic case, just one function must be provided which for each symbol $f \in \mathcal{F}_n$ returns a function of n arguments to be applied to the n attributes computed at the child node. We refer to it as the *symbol-fun*.

Suppose we have a term $t = f(t_1, \dots, t_n)$ and already recursively computed the attributed state $[q_i, a_i]$ for each child t_i. Let $g =$ `symbol-fun`(f). The attribute for t is given by $g(a_1, \dots, a_b)$.

In the non deterministic case, we may obtain the same state using different applicable rules. In that case, we need a function in order to combine the attributes into a single one. We refer to it as the *combine-fun*.

An instance of the class `afuns` contains all what is needed to attribute an automaton.

```
(defclass afuns ()
  ((symbol-fun :reader symbol-fun
               :initarg :symbol-fun)
   (combine-fun :reader combine-fun
                :initarg :combine-fun)))

(defun make-afuns (symbol-fun combine-fun)
  (make-instance 'afuns
                 :symbol-fun symbol-fun
                 :combine-fun combine-fun))
```

We can for instance count the number of runs (which is interesting for the non deterministic case). Here is the attribution mechanism for counting runs.

```
(defgeneric count-run-symbol-fun (symbol))
(defmethod count-run-symbol-fun
    ((s abstract-symbol))
  #'*)
(defparameter
    *count-afun*
  (make-afuns #'count-run-symbol-fun #'+))
```

The following `attribute-transitions-fun` operation transforms the transitions of a non attributed fly automaton into attributed transitions according to the attribution mechanism `afun`.

```
(defmethod attribute-transitions-fun
    ((transitions abstract-transitions) afun)
  (lambda (root attributed-states)
    (compute-attributed-target
     root
     (apply-transition-function
      root
      (mapcar #'in-state attributed-states)
      transitions)
     afun
     attributed-states)))
```

An attributed state $[q, a]$ is final for the attributed automaton if q was final for the non attributed automaton.

```
(defmethod attribute-final-state-fun
    ((automaton abstract-automaton))
  (lambda (attributed-state)
    (final-state-p
     (in-state attributed-state)
     automaton)))
```

With the two previous operations, we can define the operation which transforms a non attributed automaton into an attributed one according to the attribution mechanism `afun`.

```
(defmethod attribute-automaton
    ((automaton abstract-automaton) afun)
  (let ((transitions
         (transitions-of automaton)))
    (make-fly-automaton-with-transitions
     (make-fly-transitions
      (attribute-transitions-fun
       transitions afun)
      (deterministic-p automaton)
      (complete-p automaton)
      (completion-state-final transitions)
      :transitions-type
      'fly-casted-transitions)
     (signature automaton)
     (attribute-final-state-fun automaton)
     (format nil "~A-att" (name automaton)))))
```

5.2 Fly transducers

A fly transducer is just a fly automaton with an output function which may be applied to the targets.

If the fly automaton is not attributed, by default, the output function is the final state predicate which returns a Boolean value.

If the fly automaton is attributed, by default, the output function returns the attribute of the final target: in the deterministic case, the target is just a state $[q, a]$ and the result is just the attribute a; in the non deterministic case, the target is a container of attributed states $[q_1, a_1], \dots, [q_p, a_p]$ and the result is computed by applying the `combine-fun` of the attribution mechanism to the attributes $a_1, \dots, a_p$.

The main operations applicable to a term and a fly transducer are `compute-value (term fly transducer)` and `compute-final-value (term fly transducer)`.

5.2.1 Counting graph-colorings

We would like a fly transducer for counting the number of k-colorings of a graph. Note that the problem is $\#P$-complete for $k = 3$.

We start with the automaton `*2-colored*` computed previously (see Figure 8) and recognizing graphs having a proper k-coloring, then we attribute it with `*count-afun*`.

```
(setf *2-colored-counting*
      (attribute-automaton
       *2-colored*
       *count-afun*))
```

Then we do the color-projection as before.

```
(setf *count-2-colorings*
      (color-projection
       *2-colored-counting*
       2))
```

The resulting automaton computes attributed states each one containing the number of runs leading to the state.

```
AUTOGRAPH> (compute-final-target
             *t3*
             *count-2-colorings*)
o{[!<a:1 b:2>,1] [!<a:2 b:1>,1]}
```

Used as a transducer, it computes the number of k-colorings of the graph:

```
AUTOGRAPH> (compute-final-value
             *t3*
             *count-2-colorings*)
2
T
```

For some well-known graphs of graph theory like Petersen's graph, the chromatic polynomial has already been computed: it gives the number of colorings for each k. We could verify experimentally that our method gives the same values as the chromatic polynomial.

The chromatic polynomial for Petersen's graph is

$$k(k-1)(k-2)$$
$$(k^7 - 12k^6 + 67k^5 - 230k^4 + 529k^3 - 814k^2 + 775k - 352)$$

For instance, we may verify that Petersen's graph has 12960 4-colorings. Note that the problem of deciding whether a graph is k-colorable is NP-complete for $k > 2$.

```
AUTOGRAPH> (compute-final-value
     (petersen)
     (color-projection-automaton
      (attribute-automaton
       (coloring-automaton 4)
       *count-afun*)
      4))
12960
T
AUTOGRAPH> (petersen-chromatic-polynomial 4)
12960
```

5.2.2 Computing graph-colorings

The coloring of a graph described by a term may be given by a function which, given a constant position in the term, gives the assigned color number. The positions are denoted by Dewey's words which are words in $[0, m[^*$ where m is the maximal arity of a symbol in the signature.

For representing graphs, the maximal arity is 2 (`oplus`), so the positions will contain only zeros or ones.

The root position is denoted by the empty word. In outputs, it will be denoted by `E`.

For instance, the set of positions of the term `a` is { `E` } and the set of positions of `add_a_b(oplus(a,b))` is
{ `E, 0, 00, 01` }.

It is not diffcult to compute such colorings as an attribute on the automaton which verifies that a graph has a proper k-coloring.

This attribution mechanism is accessible via the variable `*assignment-afun*`.
The code implementing this mecanism is presented in Figure 12 at the end of the paper. It works both for color assigment and subset assignment. The `combine-fun` is just a union; The `symbol-fun` is the `assignment-symbol-fun` function; for every non constant operation it returns the function for computing the attribute: it extends the positions computed so far with the correct child number; in the case of colored constant symbols, if returns a zeroary function that will initiate the attribute as a list containing the empty position associated with the color of the constant.

```
AUTOGRAPH> (setf *af2*
             (attribute-automaton
              *2-colored*
              *assignment-afun*))
0-COLORED-2-att
```

The following example shows how to obtain all the possible proper colorings for the graph t_3.

```
AUTOGRAPH> (setf *f2*
                 (color-projection-automaton
                  *af2* 2))
fly-asm(0-COLORED-2-att)
AUTOGRAPH> (compute-final-value *t3* *f2*)
(([00:1] [010:1] [0110:2] [0111:2])
 ([00:2] [010:2] [0110:1] [0111:1]))
T
```

Although being finite, the set of possible proper colorings of a graph may be of exponential size. We do not necessarily need all proper colorings. If that is the case, then the enumeration mechanism described in ELS2012 [12] is just what we need.

We construct an enumerator of the final values of the fly transducer whose values are colorings. Then we just enumerate these values in order to obtain as many proper colorings as we need.

```
AUTOGRAPH> (defparameter *e*
             (final-value-enumerator
              (petersen)
              *compute-4-colorings*))
*E*
```

Then we call the enumerator to get the colorings one by one.

```
AUTOGRAPH> (call-enumerator *e*)
(([0000:3]
  [000100000:2]
  [0001000010000:3]
  [000100001000100000000000000:2]
  [000100001000100000000000001:1]
  [00010000100010000000000001:1]
  [0001000010001000000000001:1]
  [000100001000100000000001:2]
  [00010000100010000000001:3]
  [0001000010001000000001:2]))
T
```

Given the size and complexity of the value computed by our transducers, we obtain different classes of complexity (FPT, XP) [16, 8]. This has been studied and submitted to CAI2013 [5].

6. EXPERIMENTS

We have worked on many graph properties, many of which are described in [11, 4, 5] in particular on acyclic-colorings [18].

In graph theory, an acyclic-coloring is a (proper) vertex coloring in which every 2-chromatic subgraph is acyclic. The acyclic chromatic number $A(G)$ of a graph G is the least number of colors needed in any acyclic-coloring of G. It is NP-complete to determine whether $A(G) \leq 3$ (Kostochka 1978). So acyclic-colorability is not a trivial matter.

McGee's graph is shown in Figure 11; it is regular (degree 3); it has 24 vertices and 36 edges; we have a decomposition yielding a term of clique-width 10, size 76 and depth 99. This graph is 3-acyclic-colorable (but not 2-acyclic-colorable). We may verify this last fact in less than three hours and compute the number of 3-acyclic-colorings (57024) in less than six hours.

7. CONCLUSION AND PERSPECTIVES

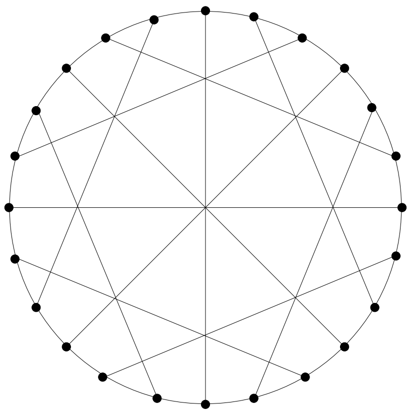


Figure 11: McGee's Graph

We have defined fly transducers on terms which can compute information about terms. When terms represent graphs, we can compute information about graphs.

One advantage of fly automata and fly transducers are their flexibility and the possibility to transform or combine them in order to obtain new ones. This can be elegantly done through the functional paradigm of Lisp. The CLOS layer is also heavily used both in `Autowrite` and `Autograph`.

In this paper we did not address the difficult problem of finding a clique-width decomposition of a graph (so the clique-width) of a graph. This problem was shown to be NP-complete in [15]. [19] gives polynomial approximated solutions to solve this problem. More can be found in [6].

In some applications, the graphs may be given by their decomposition. Some methods exist for specific graphs like cliques, grids, square grids, trees, ... For other graphs like Petersen's or McGee's, we had previously done hand decompositions. In most cases, we do not know whether we have the best decomposition (the smallest clique-width).

Recently, we have started developping a system `DecompGraph` for approximated clique-width decomposition of graphs. *Approximated* means, that it will not necessarily give the smallest possible clique-width.

The `DecompGraph` is independent from `Autograph`.
With `DecompGraph`, we can at least decompose the graphs we had already worked on and were pleased to improve the hand decomposition of Petersen's graph from $cwd = 7$ to $cwd = 6$. For McGee's graph however, we did not find a better decomposition ($cwd = 10$) with `DecompGraph`.

The decomposition of a graph is a kind of preprocessing phase. Once the graph is decomposed into a term, we may keep the decomposition and apply as many fly automata or fly transducers as we want on the term.

The fact that we can now decompose graphs (although may be not very big ones) means that we are able to effectively prove graph properties and compute graph properties starting from the graph itself which was not possible before.

Any domain using terms for representing objects (language processing, protocol verification, . . .) could benefit from fly transducers. We are looking for applications in graphs or any other domain using terms.

Acknowledgements

The authors thank the referee for his very accurate and constructive remarks.

8. REFERENCES

[1] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Draft, available from `http://tata.gforge.inria.fr`.

[2] B. Courcelle and I. Durand. Verifying monadic second order graph properties with tree automata. In *Proceedings of the 3rd European Lisp Symposium*, pages 7–21, May 2010.

[3] B. Courcelle and I. Durand. Fly-automata, their properties and applications. In B. B.-M. et al., editor, *Proceedings of the 16th International Conference on Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 264–272, Blois, France, July 2011. Springer Verlag.

[4] B. Courcelle and I. Durand. Automata for the verification of monadic second-order graph properties. *Journal of Applied Logic*, 10(4):368 – 409, 2012.

[5] B. Courcelle and I. Durand. Automata for monadic second-order model-checking. In *Proceedings of the 5th International Conference on Algebraic Programming*, Porquerolles Island, Aix-Marseille University, France, 2013. To appear in September 2013.

[6] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic, a Language Theoretic Approach*. Cambridge University Press, 2012.

[7] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23 – 52, 2001.

[8] R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49, pages 49–99. AMS-DIMACS Proceedings Series, 1999.

[9] I. Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.

[10] I. Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronics Notes in Theorical Computer Science*, 124:29–49, 2005.

[11] I. Durand. Implementing huge term automata. In *Proceedings of the 4th European Lisp Symposium*, pages 17–27, Hamburg, Germany, March 2011.

[12] I. Durand. Object enumeration. In *Proceedings of the 5th European Lisp Symposium*, pages 43–57, Zadar, Croatia, May 2012.

[13] I. Durand. Autowrite. Software, since 2002.

[14] I. Durand. Autograph. Software, since 2010.

[15] M. Fellows, F. Rosamond, U. Rotics, and S. Szeider. Clique-width minimization is NP-hard. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 354–362, Seattle, 2006.

[16] M. R. Fellows, F. V. Fomin, D. Lokshtanov, F. Rosamond, S. Saurabh, S. Szeider, and C. Thomassen. On the complexity of some colorful problems parameterized by treewidth. *Information and Computation*, 209(2):143 – 153, 2011.

[17] M. C. GOLUMBIC and U. ROTICS. On the clique-width of some perfect graph classes. *International Journal of Foundations of Computer Science*, 11(03):423–443, 2000.

[18] B. Grünbaum. Acyclic colorings of planar graphs. *Israel Journal of Mathematics*, 14:390–408, 1973.

[19] S.-I. Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):1–20, 2008.

[20] J. Thatcher and J. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

```
(defun union-fun (&key (fun #'union) (test #'equalp))
  (lambda (&rest attributes)
    (reduce (lambda (a1 a2) (funcall fun a1 a2 :test test))
            attributes :initial-value '())))

(defgeneric position (position-assignment))
(defgeneric assignment (position-assignment))

(defclass position-assignment ()
  ((position :initarg :position :reader position)
   (assignment :initarg :assignment :reader assignment))
  (:documentation "position with color or subset assignment"))

(defun make-position-assignment (position assignment)
  (make-instance 'position-assignment
                 :position position
                 :assignment assignment))

(defgeneric left-extend-position-assignment (position-assignment i))
(defmethod left-extend-position-assignment
    ((position-assignment position-assignment) (i integer))
  (make-position-assignment
   (left-extend-position (position position-assignment) i)
   (assignment position-assignment)))

(defgeneric assignment-fun (attributes1 attributes2))
(defmethod assignment-fun ((attributes1 list) (attributes2 list))
  (loop
     with attributes = '()
     for a1 in attributes1
     do (loop for a2 in attributes2
              do (push (append a1 a2) attributes))
     finally (return attributes)))

(defgeneric assignment-symbol-fun (s))
(defmethod assignment-symbol-fun ((s vbits-constant-symbol))
  (lambda ()
    (list (list (make-position-assignment (make-position '()) (vbits s))))))

(defmethod assignment-symbol-fun ((s color-constant-symbol))
  (lambda ()
    (list (list (make-position-assignment
                (make-position '())
                (symbol-color s))))))

(defmethod assignment-symbol-fun ((s abstract-parity-symbol))
  (lambda (&rest attributes)
    (setf attributes
          (loop
             for attribute in attributes
             for i from 0
             collect
               (loop
                  for position-assignments in attribute
                  collect
                    (loop
                       for position-assignment in position-assignments
                       collect (left-extend-position-assignment
                                position-assignment i)))))
    (if (endp (cdr attributes))
        (car attributes)
        (reduce #'assignment-fun attributes))))

(defparameter *assignment-afun* (make-afuns #'assignment-symbol-fun (union-fun)))
```

Figure 12: The attribute mecanism for computing position assignment

Session IV

Lazy Signal Combinators in *Common Lisp*

Max Rottenkolber
Rottenkolber Software Engineering
Karlstraße 15
53115 Bonn, Germany
max@mr.gy

ABSTRACT

This demonstration explores an intuitive approach to signal synthesis using *Higher-order functions* to compute *lazy* representations of signals. Given a few primitives which adhere to said representation, ad hoc combinations of signals can be formed to be used to synthesize new signals intuitively. Results from an experimental implementation of this approach in an embedded[1] signal synthesis language called `SOUNDLAB` prove the approach to be powerful and extensible.

Categories and Subject Descriptors

J.5 [**Arts and Humanities**]: Performing Arts;
H.5.5 [**Information Systems**]: Information Interfaces and Presentation—*Sound and Music Computing*

General Terms

Demonstration

Keywords

Signal synthesis, combinatorial higher-order functions, Common Lisp

1. INTRODUCTION

The described approach is mainly inspired from experience gained by using analogue sound synthesizers. While every analogue synthesizer has its own unique sound based on the physical parts it is made of, most do share their key concepts. Usually a limited number of oscillators generate signals resembling—more or less—sine waves which are then modulated by being combined with each other in different ways.

`SOUNDLAB`—an experimental implementation of the presented approach—is designed to enable the user to explore ways of signal combination. It does so by defining an embedded *domain specific language* which provides axioms that generate primitive signals and axioms that combine arbitrary signals into new signals. The semantics of the language are based on a signal interface agreed on by every component. Furthermore `SOUNDLAB` allows the use of *Common Lisp's* means of abstraction to define compound signals and signal combinators. Primitive as well as compound parts of the system form a homogeneous group of objects defined by their shared interfaces, which grant the system power and flexibility of a *Lisp* system.

There are of course many free software implementations[2] of signal synthesis systems with programming language interfaces. `SOUNDLAB` is—when compared to others—much simpler and entirely written and embedded in *Common Lisp*.

`SOUNDLAB` is free software licensed under the *GNU AGPL* and can be obtained at `http://mr.gy/software/soundlab/`.

2. RENDERING SIGNALS

Before discussing signal synthesis, we must define ways for consuming the synthesized signal as well as for verification of our results. Because our domain is music, we need to be able to play back signals as sound. Furthermore visualizing a signal can be useful for debugging since some properties of a signal are better conceived visually than aurally.

For both forms of presentation a technique called *sampling* is used—which will not be described in detail here. All that is needed to know for this approach, is that the sampling routine records a sequence of linear amplitude values according to a time span and a function—or signal—which maps values of time to values of amplitude. The resulting sequence resembles the kind of data that can be fed into standard digital sound adapters or plotting applications.

```
;;; Approximate type of a sampling function.

(FUNCTION ((FUNCTION (REAL) REAL) REAL)
          (SEQUENCE REAL))
```

`SOUNDLAB` derives its signal type from this rationale. It also exports two functions which record signals to standard *WAVE* audio files and *Gnuplot* compatible data files respectively. `SOUNDLAB` also chooses arbitrary but sensible units and

[1]Embedded in *Common Lisp*, that is.

[2]See for instance Overtone (`http://overtone.github.io`) and Csound (`http://www.csounds.com`).

scales for time and amplitude. Time is chosen to be a number in seconds greater than zero and amplitude is chosen to be a number ranging from -1 to 1. Results of inputs to the sampling routine exceeding these bounds are undefined.

3. SIGNAL SYNTHESIS

3.1 Signals as functions

As discussed in the previous section, functions are the natural way to model a signal. Furthermore signals as functions encourage lazy operations without enforcing them—which can later be useful for aggressive optimizations.

```
;;; Type of a signal.

(FUNCTION (REAL) REAL)
```

A crucial type of signal is the sine wave—since in theory, all signals are sums of sine waves. *Common Lisp* provides us with a sine function `SIN` which serves our purpose well. We could pass `#'SIN` to a sampling routine as is, which would produce a very low frequency signal below the human hearing threshold. In order to specify other frequencies a constructor `SINE` is defined which accepts a frequency in Hz and returns the respective sine signal.

```
;;; Constructor for primitive sine signals.

(defun sine (frequency)
  (lambda (x) (sin (* 2 pi frequency x))))
```

Additionally a constructor for chorded signals could be defined as a function that takes two signals as arguments and returns a function that sums and normalizes them according to the boundaries we defined in the previous section.

```
;;; Constructor for a chord of two signals.

(defun chord-2 (signal-1 signal-2)
  (lambda (x) (* (+ (funcall signal-1 x)
                    (funcall signal-2 x))
                 1/2)))
```

The `CHORD-2` function demonstrates the important traits of signals as functions. A new signal in form of an anonymous function is being compiled whenever we call `CHORD-2`. Because the actual processing of the arguments is postponed until sampling occurs, operation on signals is cheap. Furthermore calls to `CHORD-2` can be combined to create chords with an arbitrary number of voices.

3.2 Signal combination

As seen in the previous section, modeling signals as functions enables us to write small, cheap and powerful signal combinators which can be chained to arbitrary extent. When chosen carefully, a small set of primitive combinators and signals can be used to create infinitely complex sounds.

```
;;; Type of a signal combinator.

(FUNCTION (&REST (FUNCTION (REAL) REAL))
          (FUNCTION (REAL) REAL))
```

While building `SOUNDLAB`, some primitives turned out to be especially useful. `FLATLINE`—a constant signal constructor—serves a simple but important purpose. It takes a number as its only argument and returns a flat signal with a constant amplitude. When passed to a signal combinator its purpose is usually to scale combinations of signals. `ADD` is a general signal adder. It takes an arbitrary number of signals and sums them. Likewise, `MULTIPLY` multiplies signals. The `CHORD-2` combinator of the previous section can be defined more generally using these primitives.

```
;;; Implementation of FLATLINE.

(defun flatline (amplitude)
  (lambda (x)
    (declare (ignore x))
    amplitude))
```

```
;;; Generic implementation of CHORD.

(defun chord (&rest signals)
  (multiply (apply #'add signals)
            (flatline (/ 1 (length signals)))))
```

Note that—due to the normalization performed by `CHORD-2`—the equivalent of `(chord a b c)` is

```
(chord-2 (chord-2 a b) (chord-2 c (flatline 1)))
```

as opposed to

```
(chord-2 (chord-2 a b) c)
```

which would produce the chord of `C` and the chord of `A` and `B` instead of the chord of `A`, `B` and `C`.

Furthermore, using signals as arguments to operations where constants would suffice whenever possible has proven to be feasible and powerful. Whenever a component is being modeled that would be controlled by a knob or fader in an analogue synthesizer, then its digital counterpart should be controlled by a signal. Take for instance a signal combinator `MIX*` whose purpose is to merge two signals—just like `CHORD`—while additionally providing a way to control how much each input signal amounts to the mixed signal. So what would have been a *Dry/Wet* knob on an analogue synthesizer becomes a signal in our case. Our `MIX*` takes three signals as arguments, two to be mixed and a third to control their amounts. For ease of implementation we also introduce `SUBTRACT`—the counterpart to `ADD`.

```
;;; Implementation of MIX*.

(defun mix* (signal-a signal-b ratio-signal)
  (add (multiply signal-a
                 (subtract (flatline 1)
                           ratio-signal))
       (multiply signal-b
                 ratio-signal)))
```

Staying within closure of the signal representation—that is trying hard to define our operations on a uniform signal representation only—grants the system a lot of power and flexibility. All of the presented signal combinators can be plugged into each other without restriction. As of now some care has to be taken to not produce signals exceeding the defined boundaries—see *Rendering signals.* Additionally, some combinators make use of non-audible signals. For instance `MIX*` expects `RATIO-SIGNAL` to return values ranging from zero to one and `MULTIPLY` is used in combination with `FLATLINE` to moderate signals. `SOUNDLAB` fails to address the issue of having multiple informal subtypes of signals. As of now the user has to refer to the documentation of a combinator to find out if it expects certain constraints—as is the case with `MIX*`. Nevertheless, our few examples can already be used to produce complex sounds. The code snippet below works in `SOUNDLAB` as is and produces a rhythmically phasing sound.

```
;;; Possible usage of the presented combinators.

(defun a-4 () (sine 440))
(defun a-5 () (sine 220))

;; Normalize a sine to 0-1 for use as RATIO-SIGNAL.
(defun sine-ratio ()
  (multiply (add (sine 1)
                 (flatline 1))
            (flatline 1/2)))

;; Produce a WAVE file.
(export-function-wave
  ;; A complex signal.
  (mix* (chord (a-4) (a-5))
        (multiply (a-4) (a-5))
        (sine-ratio))
  ;; Length of the sampling in seconds.
  4
  ;; Output file.
  #p"test.wav")
```

4. THE STATE OF *SOUNDLAB*

As of the time of this writing `SOUNDLAB` consists of roughly 500 lines of source code. It depends on a minimal library for writing *WAVE* files and is written entirely in *Common Lisp*. The source code is fairly well documented and frugal.

While being compact `SOUNDLAB` provides basic routines for working with western notes and tempo, a few primitive waveforms, *ADSR* envelopes with customizable slopes and the ability to form arbitrary waveforms from envelopes, a good handfull of signal combinators and last but not least an experimental lowpass filter. A stable API is nowhere near in sight but some trends in design are becoming clear.

On the roadmap are classic sound synthesis features like resonance, routines for importing signals from *WAVE* files and many small but essential details like bezier curved slopes for envelopes.

5. CONCLUSIONS

`SOUNDLAB`—even in its immature state—presents an opportunity to explore abstract signal synthesis from scratch for engineers and artist alike. Its simplicity encourages hacking and eases understanding. While many complex problems surrounding signal synthesis remain unsolved, its lazy combinatorial approach forms a powerful and extensible framework capable of implementing classic as well as uncharted sound synthesis features.

The demonstrated approach proved to be especially suited to exploratory sound engineering. Ad-hoc signal pipelines can be built quickly in a declarative way, encouraging reusability and creativity. In comparison to other tools in the domain the line between using and extending the system is blurry. Where *Csound* lets the user declaratively configure instruments and controls using *XML*, `SOUNDLAB` emphasizes the user to use its built-in primitives and all of *Common Lisp* to stack layers of signal sources and modulators on top of each other. When compared to *Overtone*—a *Clojure* front-end to the *SuperCollider* audio system—`SOUNDLAB`'s back-end independency and simplicity make it seem more suited for exploration and hacking. Its core concepts are few and simple and its codebase is tiny and modular despite some advanced features like envelopes, musical scales and tempo, a lowpass filter and many kinds of signal combinators being implemented.

Many of *Common Lisp's* idioms proved to be an ideal fit for the domain of signal synthesis. Furthermore, embedding a signal synthesis language in *Common Lisp* provides the system with unmatched agility. While the core approach is mainly built on top of functional paradigms, extensions like signal subtype checking—as mentioned in section 3.2—could be implemented using macros.

I personally had tons of fun building and playing with `SOUNDLAB`. I encourage everyone interested in computerized music to dive into the source code and experiment with the system—it really is that simple. Feedback and contributions are welcome!

6. ACKNOWLEDGMENTS

Thanks to Michael Falkenbach for teaching me a whole lot about analogue audio hacking.[3]

Thanks to Vera Schliefer for introducing me to signal theory.

Thanks to Drew Crampsie for providing the *Common Lisp* community with resources regarding the implementation of monadic combinators. [4]

[3] `http://soundpiloten.de`

[4] `http://common-lisp.net/~dcrampsie/smug.html`

CL-NLP — a Natural Language Processing library for Common Lisp

Vsevolod Domkin

vseloved@gmail.com

Abstract

CL-NLP is the new Common Lisp Natural Language Processing library the development of which has started in 2013. The purpose of the library is to assemble a comprehensive suite of NLP algorithms, models and adapters to popular NLP resources for Common Lisp. Similar projects in other languages include the Python Natural Language Toolkit NLTK [NLTK], which is the most popular starting point for educational work and academic research, Stanford CoreNLP [CoreNLP] and Apache OpenNLP [OpenNLP] libraries.

The motivations for its creation include the following:

- Lisp is very well suited for NLP projects, providing broad support for statistical calculations, symbolic computation, as well as string and tree mainpulation;
- unfortunately, a lot of work was done in the NLP area in Common Lisp before the advent of open-source, and its artifacts are scattered across various libraries, university Internet web-sites and books, not gathered under one roof, curated and supported. The idea behind CL-NLP is to provide a central repository for such artifacts in the future;
- the existing open source Common Lisp NLP tools are insufficient.

This article presents an overview of the current state of CL-NLP, a discussion of its implementation and a plan for its further development.

Keywords: Natural Language Processing, Programming Environments, Software Architectures, Reusable Software,Language Constructs and Features

1. Overview

1.1. Previous work

There is an existing NLP Lisp library — cl-langutils [langutils], which was considered as a candidate to serve as a base for this effort, but its original authors had abandoned it, and it has not seen active development for a long period of time with only occasional bugfixes. The main concern with cl-langutils is that it doesn't provide a modular foundation suitable for supporting many alternative ways to solving the same tasks which is required to assemble the suite of algorithms that should become CL-NLP.

Apart from langutils, other repositories of NLP-related Common Lisp code include:

- code from “Natural Language Processing in Lisp” book [NLPinLisp]
- code from “Natural Language Understanding” book [NLU]

- code from “Paradigms of Artificial Intelligence Programming” book [PAIP]
- code from “Artificial Intelligence Programming” book [AIProg]
- CMU AI repository [CMUAI]
- Lexiparse project [Lexiparse]
- Sparser project [sparser]
- CL-EARLY-PARSER project [CLEarly]
- Basic-English-Grammar project [BasicEngGrammar]
- Various Wordnet interfaces, including cl-wordnet [CLWordnet] and cffi-wordnet [cffiWordnet]
- Soundex project [Soundex]

1.2. Library design

The most important design goal for CL-NLP is to provide a modular extensible foundation to accumulate over time various NLP algorithms and models. To achieve it the library uses several layers of modularization facilities: systems, packages and CLOS generic functions.

At the top level the library is provided as two ASDF systems:

- `cl-nlp` implements the core functionality;
- `cl-nlp-contrib` provides various adapters to external systems, that have additional dependencies not essential to the core library.

At the namespacing level the library is split into packages of three types:

- basic packages that provide the foundational data structures and utilities to serve as the common “language” of CL-NLP – these include: `nlp.core`, `nlp.util`, `nlp.test-util`, and `nlp.corpora`;
- functional packages in `cl-nlp` system that implement a broad range of functionality in one of the areas of the libraries scope: `nlp.syntax`, `nlp.generation`, `nlp.phonetics` etc;
- functional packages in `cl-nlp-contrib` system that implement specific adapters to external NLP resources: `nlp.contrib.wordnet`, `nlp.contrib.ms-ngrams`;
- `nlp-user` package which collects all the public symbols from the other packages and provides additional facilities to enhance the usability of interactive development with CL-NLP.

Common Lisp's lack of hierarchical packages is often regarded as a shortcoming of the language. Yet, package hierarchy may be emulated by using an appropriate naming scheme, like the scheme of CL-NLP packages and the use of some utility functions. At the same time the package system is built with separation of concerns in mind that is lacking in hierarchical namespace systems of such languages as Python and Java that tie package names to the file system and make it difficult to evolve such hierarchy without the help of special tools. Besides, this coupling doesn't allow to easily aggreagate names from more than 1 file in a module that is often desirable.

At the next level each package exports a number of generic functions for major operations that serve as the API entry point: `tokenize`, `lemmatize`, `tag`, `parse` etc. The methods of such generic functions implement specific algortihms of tokenization, lemmatization etc. By convention the first argument of each generic function should be an instance of the class which specifies, what algorithm should be implemented. For instance, such classes as `regex-tokenizer`, `markov-chain-generator`, `hmm-tagger` are defined. These instances provide parametrization possibilities for the algorithms and allow for code reuse through inheritance.

The following code demonstrates this principle int the implementation of the simple regular-expression based word tokenizer.

```
(defgeneric tokenize (tokenizer string)
  (:documentation
   "Tokenize STRING with TOKENIZER. Outputs 2 values:
    - list of words
    - list of spans as beg-end cons pairs"))

(defclass regex-word-tokenizer (tokenizer)
  ((regex :accessor tokenizer-regex :initarg :regex
          :initform
          (re:create-scanner
           "\\w+|[!\"#$%&'*+,./:;<=>?@^`~…\\(\\)⟨⟩{}\\[\\|\\]——
—«»“”‘’¶-]")
          :documentation
          "A simpler variant would be [^\\s]+ —
           it doesn't split punctuation, yet sometimes it's desirable."))
  (:documentation
   "Regex-based word tokenizer."))

(defmethod tokenize ((tokenizer regex-word-tokenizer) string)
  (loop :for (beg end) :on (re:all-matches (tokenizer-regex tokenizer)
                                           string)
                       :by #'cddr
     :collect (subseq string beg end) :into words
     :collect (cons beg end) :into spans
     :finally (return (values words
                              spans))))
```

Additionally, the CLOS method-combination facilities are extensively used to factor auxiliary actions out of the main methods implementations.

```
(defmethod tokenize :around ((tokenizer tokenizer) string)
  "Pre-split text into lines and tokenize each line separately."
  (let ((offset 0)
        words spans)
    (loop :for line :in (split #\Newline string) :do
       (multiple-value-bind (ts ss) (call-next-method tokenizer line)
         (setf words (nconc words ts)
               spans (nconc spans (mapcar #`(cons (+ (car %) offset)
                                                  (+ (cdr %) offset))
                                          ss)))
         (incf offset (1+ (length line)))))
    (values words
            spans)))
```

To further simplify development with CL-NLP singleton instances of certain classes with default parameter values are defined where it is possible. For instance, there's a default `<word-tokenizer>` singleton, which is an instance of `postprocessing-regex-word-tokenizer` class. Also by convention these instances have their names in angular brackets and there is a special macro for efficiently defining them. This is how the macro is invoked:

```
(define-lazy-singleton word-chunker
    (make-instance 'regex-word-tokenizer
                   :regex (re:create-scanner "[^\\s]+"))
  "Dumb word tokenizer, that will not split punctuation from words.")
```

It uses `define-symbol-macro` in its definition:

```
(defmacro define-lazy-singleton (name init &optional docstring)
  "Define a function NAME, that will return a singleton object,
   initialized lazily with INIT on first call.
   Also define a symbol macro <NAME> that will expand to (NAME)."
  (with-gensyms (singleton)
    `(let (,singleton)
       (defun ,name ()
         ,docstring
         (or ,singleton
             (setf ,singleton ,init)))
       (define-symbol-macro ,(mksym name :format "<~A>") (,name)))))
```

One of the biggest shortcomings of the design of NLP libraries based on conventional approach to object-orientation, such as NLTK or OpenNLP, is that the algorithms are implemented inside the concrete classes. Taking into account the extensive usage of inheritance this brings to the situation in practice, when parts of the implementation are scattered around several classes and files which greatly impedes its readability and in effect clarity. Another problem is the need to properly instantiate the classes before performing the tasks which may not be straighforward, especially in multi-threaded systems, as not all of the classes are implented in a thread-safe manner. The approach taken by CL-NLP which extensively utilizes CLOS capabilities tries to solve these problems while maintaining the extensible nature of object-oriented programs.

2. Main modules

2.1. Util

The `nlp.util` package defines the basic set of utilities to handle individual characters, words, text strings, files, trees of symbols, perform common mathematical operations and some supplementary utilities. The package `nlp.util` provides specific procedures to run unit tests and test algorithms with various corpora.

2.2. Core

The `nlp.core` package defines the basic data sctructures and algorithms used in other CL-NLP modules. They are also intended for stand-alone usage. These include:

- the basic `token` data-structure and `tokenization` generic function with several basic tokenization algorithms for splitting text chunks, words and sentences;
- language modelling with ngrams, implemented with the following classes:
    - `ngrams` is a low level class which wraps access to an underlying storage of ngrams (an in-

memory hash-table, a database etc.) It supports such functions as getting ngram frequency individually or in batchs (`freq` and freqs), approximated probability (`prob`, `probs`) and log of probability (`logprob`, `logprobs`), and conditional probabilites (`cond-prob`, `cond-probs`, `cond-logprob`, `cond-logprobs`). The default ngrams implementation is the `table-ngrams` class that stores them in a hash-table.

  - `language-model` is a more high-level interface that incapsulates access to a group of ngrams of different orders to provide smoothing capabilities for the methods `freqs`, `probs`, `logprobs` and `cond-logprobs`. Two implementations are provided: `plain-lm` without any smoothing and `stupid-backoff-lm` which implements the Stupid Backoff smoothing algorithm [LargeLMinMT]. Additionally, in `cl-nlp-contrib` the implementation of access to Microsoft Web N-gram Services [MSWebNgrams]. More language models should be added in the future to support other smoothing language models [SmoothingLMs], as well as adapter for external language modelling software, such as BerkeleyLM [BerkeleyLM].

2.3. Corpora

The `nlp.corpora` package implements functions to load and access commonly used lingusitic corpora, such the Brown corpus [BrownCorpus] or Penn Treebank [PennTreebank]. The basic data-structures for corpus management are:

- `text` which holds an individual unit of the corpus' text data in the raw, clean-up and tokenized forms;
- `corpus` which holds a collection of texts with possible various groupings of them (for example by-category, by-author etc).

2.3. Syntax

The `nlp.syntax` package provides the generic functions `tag`, `parse`, and `parse-n` (which returns the N most likely parses of the sentence). The implementations of taggers include an `hmm-tagger` and a `glm-tagger`. And for parsing a common PCFG-based algorithm is implemented with the lazy algorithm for finding N best parses [kBestParsing].

2.5. Wordnet

The `nlp.contrib.wordnet` package implements the interface to Wordnet lexical database of English [Wordnet]. There are at least two other libraries which provide access to Wordnet from Common Lisp that were mentioned previously. This libraries were not used as a basis for the implementation of Wordnet interface in CL-NLP. The reason for that was that is that they both interface (directly or indirectly) with the custom Wordnet file format. There are many alternative Wordnet storage formats, including a relational database ones [WordNetSQL]. The advantages of SQL-based representation are:

- it is standard so there may be multiple ways to interact with it, including the stock SQL clients;
- SQL interaction is well-supported in client libraries;
- it is transparent;
- it is distributed as a single file;
- it is rather fast and may be optimized if necessary.

That is why Wordnet interaction in CL-NLP was implemented using SQL Wordnet representation, specifically, an sqlite database. CLSQL [CLSQL] was chosen as the client library, because it supports many SQL databases and so allows to change SQL backend in the future if necessary.

The other benefit of implementing Wordnet support this way is that it provides another alternative solution for Lisp-Wordnet interface in addition to the existing ones.

2.6. Other modules

Other modules are for such functionality as phonetics, morphology, text generation, text classification and clustering, semantic analysis etc. are under development.

3. Implementation notes

3.1. Trade-offs

There are several qualities of code for which CL-NLP can be optimized. They include:

- simplicity – how simple is the implementation of the functionality;
- performance – how effective and optimized are the library's algorithms;
- extensibility – how easy it is to add new functionality;
- uniformity – how much does the code follow a single set of standards and conventions.

Extensibility and uniformity receive top priority in the implementation of CL-NLP. As for performance, the algorithms are made as efficient as possible without compromising the aforementioned qualities. The reasons for that are:

- in many use cases optimal performance is not required;
- when optimal performance is required, it is possible to create a custom optimized implementation of the algorithm. At the same time, supporting and extending a lot of custom implementations not following a set of unifying principles is an unnecessary burden.

As for simplicity, sometimes the implementation is made more general to allow for different use cases and provide several extension points. This, in effect, adds some complexity comapred to te most straightforward solution, but the complexity is justified by the overall preference to extensibility.

3.2. Stylistic issues

As in most programming language communities, there are many debates about good and bad style of Common Lisp programs. In general, CL-NLP follows the Google Common Lisp Styleguide [GoogleCLStyleguide]. Besides, the following stylistic principl apply:

1. Using the whole language.

   Some people suggest to exclude some Common Lisp functions/macros from the program lexicon on the basis that there are better alternative versions in the standard. The classic example of this is `do` and `loop` macros that both allow to express arbitrary iteration algorithms, but in substantially different manners. So the proposal is to choose one of the two and use it consistently. The approach taken in CL-NLP is the opposite: to use the construct that allows to express the computation in the most concise and clear way regardless of the constructs used in other parts of the system. This approach is also taken in anticipation of the need to integrate many algorithms coming from different people and sources, which may rely on different coding standards.

2. Extending the language to achieve more declarative and concise code.

   The “Growing a Language” [GrowingALang] approach is characteristic to Lisp systems, but at the same time it is often proposed not to use utility extensions to the language that merely change the surface syntax, but do not add anything to the semantics. CL-NLP is built on a different premise – that overlooking syntactic convenience is detrimental to the code clarity. It uses the reasonable-utilities library [rutils] which provides a lot of extensions to the Common Lisp standard library to improve the usability of handling strings, hash-tables, files, sequences etc.

3. Providing multiple choice to the user.

   There is a broad range of use cases for CL-NLP: from experimental work to production usage. To support different usage scenarios and in effect a different interaction model with the library clients, the library should provide different interfaces. For instance, for interactive use it is convenient to have all the functions immediately available, to operate with short operation names, to have sensible defaults. The requirements for production systems are more in the areas of possibility to granuarily control the used operations, to optimize their performance and resource requirements and to maintain the resulting code. CL-NLP's development goal is to support all these modes of operations by providing many alternative usage paths with the help of packaging, aliases, default implementations and pre-configured classes and other means.

4. References

[AIProg] Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, James R. Meehan, 1987, “Artificial Intelligence Programming”

[BasicEngGrammar] http://www.cliki.net/Basic-English-Grammar

[BerkeleyLM] https://code.google.com/p/berkeleylm/

[BrownCorpus] W. N. Francis, H. Kucera, 1979, “A Standard Corpus of Present-Day Edited American English, for use with Digital Computers, Revised and Amplified”

[cffiWordnet] https://github.com/kraison/cffi-wordnet[Kakkonen] Tuomo Kakkonen, 2007, "Framework and Resources for Natural Language Parser Evaluation"

[CLEarly] http://www.cliki.net/CL-EARLEY-PARSER

[CLSQL] http://clsql.b9.com/

[CLWordnet] https://github.com/TheDarkTrumpet/cl-wordnet

[CMUAI] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/0.html

[CoreNLP] Stanford Core NLP, http://www-nlp.stanford.edu/software/corenlp.shtml

[GoogleCLStyleguide] Robert Brown, François-René Rideau, “Google Common Lisp Style Guide”, http://google-styleguide.googlecode.com/svn/trunk/lispguide.xml

[GrowingALang] Guy L. Steele Jr., 1999, “Growing a Language”

[kBestParsing] Liang Huang, David Chiang, 2005, “Better k-best Parsing”

[langutils] Ian Eslick, Hugo Liu, 2005, "Langutils: A Natural Language Toolkit for Common Lisp"

[LargeLMinMT] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, Jeffrey Dean, “Large

Language Models in Machine Translation"

[Lexiparse] Drew Mcdermott, 2005, "Lexiparse: A Lexicon-based Parser for Lisp Applications"

[MSWebNgrams] http://web-ngram.research.microsoft.com/

[NLPinLisp] Gerald Gazdar, Chris Mellish, "Natural Language Processing in Lisp/Prolog/Pop11", source code available from: https://bitbucket.org/msorc/nlp-in-lisp

[NLTK] NLTK, http://nltk.org/

[NLU] James Allen, 1994, "Natural Language Understanding (2nd Edition)"

[OpenNLP] Apache OpenNLP, http://opennlp.apache.org/

[PAIP] Peter Norvig, 1992, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp"

[PennTreebank] http://www.cis.upenn.edu/~treebank/

[rutils] https://github.com/vseloved/rutils

[SmoothingLMs] Gina-Anne Levow, "Smoothing N-gram Language Models"

[Soundex] http://www.cliki.net/Soundex

[sparser] https://code.google.com/p/sparser

[Wordnet] http://wordnet.princeton.edu/

[WordNetSQL] http://sourceforge.net/projects/wnsql/